Analyse des erreurs d'arrondi sur les nombres à virgule flottante par programmation par contraintes

Rémy Garcia

UNIVERSITÉ CÔTE D'AZUR

ÉCOLE DOCTORALE
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

THÈSE DE DOCTORAT

Analyse des erreurs d'arrondi sur les nombres à virgule flottante par programmation par contraintes

Rémy GARCIA

Laboratoire d'Informatique, de Signaux et Systèmes de Sophia Antipolis (I3S)
UMR7271 UCA CNRS

Présentée en vue de l'obtention du grade de docteur en Informatique **d'**Université Côte d'Azur

Dirigée par : Claude MICHEL, Ingénieur de Recherche, Université Côte d'Azur
Co-encadrée par : Michel RUEHER, Professeur Émérite, Université Côte d'Azur
Soutenue le : 8 Septembre 2021

Devant le jury, composé de :
Jean-Paul COMET, Professeur, Université Côte d'Azur
Sylvie PUTOT, Professeur, École Polytechnique
Andreas PODELSKI, Professeur, Université de Freiburg
Matthieu MARTEL, Professeur, Université de Perpignan Via Domitia

AGENCE NATIONALE DE LA RECHERCHE
ANR

ANALYSE DES ERREURS D'ARRONDI SUR LES NOMBRES À VIRGULE FLOTTANTE PAR PROGRAMMATION PAR CONTRAINTES

Floating-point numbers round-off error analysis by constraint programming

Rémy GARCIA

Jury :

Président du jury
Jean-Paul COMET, Professeur, Université Côte d'Azur

Rapporteurs
Sylvie PUTOT, Professeur, École Polytechnique
Andreas PODELSKI, Professeur, Université de Freiburg

Examinateurs
Matthieu MARTEL, Professeur, Université de Perpignan Via Domitia

Directeur de thèse
Claude MICHEL, Ingénieur de Recherche, Université Côte d'Azur

Co-encadrant de thèse
Michel RUEHER, Professeur Émérite, Université Côte d'Azur

Université Côte d'Azur

Rémy GARCIA

Analyse des erreurs d'arrondi sur les nombres à virgule flottante par programmation par contraintes

xi+133 p.

Ce document a été préparé avec LaTeX2e et la classe `these-ISSS` version v. 2.10.

Impression : sommaire.tex – 25/8/2021 – 19:13

Révision pour la classe : these-ISSS.cls,v 2.10 2020/06/24 14:16:37 mpelleau

Analyse des erreurs d'arrondi sur les nombres à virgule flottante par programmation par contraintes

Résumé

Les nombres à virgule flottante sont utilisés dans de nombreuses applications pour effectuer des calculs, souvent à l'insu de l'utilisateur. Les modèles mathématiques de ces applications utilisent des nombres réels qui ne sont souvent pas représentables sur un ordinateur. En effet, une représentation binaire finie n'est pas suffisante pour représenter l'ensemble continu et infini des nombres réels. Le problème est que le calcul avec des nombres à virgule flottante introduit souvent une erreur d'arrondi par rapport à son équivalent sur les nombres réels. Connaître l'ordre de grandeur de cette erreur est essentiel afin de comprendre correctement le comportement d'un programme. De nombreux outils en analyse d'erreurs calculent une sur-approximation des erreurs. Ces sur-approximations sont souvent trop grossières pour évaluer efficacement l'impact de l'erreur sur le comportement du programme. D'autres outils calculent une sous-approximation de l'erreur maximale, *i.e.,* la plus grande erreur possible en valeur absolue. Ces sous-approximations sont soit incorrectes, soit inatteignables. Dans cette thèse, nous proposons un système de contraintes capable de capturer et de raisonner sur l'erreur produite par un programme qui effectue des calculs avec des nombres à virgule flottante. Nous proposons également un algorithme afin de chercher l'erreur maximale. Pour cela, notre algorithme calcule à la fois une sur-approximation et une sous-approximation rigoureuses de l'erreur maximale. Une sur-approximation est obtenue à partir du système de contraintes pour les erreurs, tandis qu'une sous-approximation atteignable est produite à l'aide d'une procédure générer-et-tester et d'une recherche locale. Notre algorithme est le premier à combiner à la fois une sur-approximation et une sous-approximation de l'erreur. Nos méthodes sont implémentées dans un solveur, appelé FErA. Les performances sur un ensemble de problèmes communs sont compétitives : l'encadrement rigoureux produit est précis et se compare bien par rapport aux autres outils de l'état de l'art.

Mots-clés : programmation par contraintes, nombres à virgule flottante, erreur d'arrondi, analyse d'erreurs, contraintes sur les erreurs, optimisation

Floating-point numbers round-off error analysis by constraint programming

Abstract

Floating-point numbers are used in many applications to perform computations, often without the user's knowledge. The mathematical models of these applications use real numbers that are often not representable on a computer. Indeed, a finite binary representation is not sufficient to represent the continuous and infinite set of real numbers. The problem is that computing with floating-point numbers often introduces a rounding error compared to its equivalent over real numbers. Knowing the order of magnitude of this error is essential in order to correctly understand the behaviour of a program. Many error analysis tools calculate an over-approximation of the errors. These over-approximations are often too coarse to effectively assess the impact of the error on the behaviour of the program. Other tools calculate an under-approximation of the maximum error, *i.e.,* the largest possible error in absolute value. These under-approximations are either incorrect or unreachable. In this thesis, we propose a constraint system capable of capturing and reasoning about the error produced by a program that performs computations with floating-point numbers. We also propose an algorithm to search for the maximum error. For this purpose, our algorithm computes both a rigorous over-approximation and a rigorous under-approximation of the maximum error. An over-approximation is obtained from the constraint system for the errors, while a reachable under-approximation is produced using a generate-and-test procedure and a local search. Our algorithm is the first to combine both an over-approximation and an under-approximation of the error. Our methods are implemented in a solver, called FErA. Performance on a set of common problems is competitive: the rigorous enclosure produced is accurate and compares well with other state-of-the-art tools.

Keywords: constraint programming, floating-point numbers, round-off error, error analysis, constraints over errors, optimization

Remerciements

Je tiens tout d'abord à remercier Sylvie PUTOT et Andreas PODELSKI d'avoir accepté de rapporter cette thèse. Je remercie également Matthieu MARTEL et Jean-Paul COMET d'avoir participé à mon jury de thèse.

Merci à Claude MICHEL et Michel RUEHER, mes encadrants de thèse. Je tiens en particulier à les remercier pour les conseils, la patience, et la liberté qu'ils m'ont accordé depuis le début.

Je tiens aussi à remercier Laurent MICHEL pour le temps qu'il a passé à m'expliquer le fonctionnement du solveur Objective-CP et pour son enthousiasme constant dans ses explications tant théoriques que techniques.

Je remercie les permanents de l'équipe MDSC : Jean-Charles RÉGIN, Arnaud MALAPERT, Marie PELLEAU, Cinzia DI GIUSTO, Enrico FORMENTI, Adrien RICHARD, Kévin PERROT, Bruno MARTIN, Sandrine JULIA, Joëlle DESPEYROUX, et Elisabetta DE MARIA, pour leur accueil et les nombreuses discussions que nous avons pu avoir pendant ma thèse.

Merci à Benjamin MIRAGLIO, Jonathan BEHAEGEL, Heytem ZITOUN, Ophélie GUINAUDEAU, Ingrid GRENET (et Reiki), Assia KAMAL-IDRISSI, Piotr KRASNOWSKI, Laetitia LAVERSA, Nicolas ISOART, Arthur FINKELSTEIN, Laetitia GIBART, Sara RIVA, Samvel DERSARKISSIAN, Giulia ROCCO, François DORÉ, Amélie GRUEL, Loïc GERMERIE, et aux autres doctorants et stagiaires que j'aurai pu oublier d'avoir rendu ces années passées en thèse plus agréables et mémorables.

Merci à Hélène COLLAVIZZA, Erick GALLESIO, Pascal URSO, Stéphane LAVIROTTE, et Julien PROVILLARD avec qui j'ai eu le plaisir d'enseigner pendant mon doctorat.

Merci aux différentes équipes de l'ADSTIC, et en particulier à Lyes KHACEF, Adrien RUSSO, et Amina GHRISSI pour tout ce qu'ils ont fait depuis notre arrivée dans l'association. Je souhaite aussi bon courage à la prochaine équipe.

Je remercie aussi mes amis, que je ne nommerai pas, au risque d'en oublier certains, pour tous les moments partagés ensemble. Je tiens tout de même à remercier Tiphaine GILSON pour ces longues années d'amitié et Mircea MOSCU pour ses blagues inimitables.

Merci à Diana RESMERITA pour ces quatre ans de colocation extraordinaires, les apéros, les séances de sports, les discussions, les ragots, et tout le reste.

Je remercie bien sûr ma famille, et en particulier mes parents et mon frère, pour le support et le soutient inconditionnel qu'ils m'accordent depuis toujours.

Je remercie finalement toutes les personnes que j'ai croisées, à l'université et en dehors, et qui ont permis à ces années de thèse d'être inoubliables.

Table des matières

CHAPITRE 1

Introduction

De nombreuses applications en ingénierie, mathématique, ou physique utilisent les nombres à virgule flottante pour effectuer des calculs, souvent à l'insu de l'utilisateur. Ces applications vont du logiciel embarqué dans une voiture [Yamada, 1998], au machine learning [Köster *et al.*, 2017], en passant par les modèles représentant des phénomènes physiques [Lee, 2014]. Les nombres utilisés dans les modèles mathématique de ces applications sont souvent des nombres réels et ne sont pas représentable sur un ordinateur. En effet, une représentation binaire finie n'est pas suffisante pour représenter l'ensemble continu et infini des nombres réels. C'est pourquoi les nombres à virgule flottante, qui sont une approximation discrète et finie des nombres réels, sont utilisés à leur place. Tous les nombres réels n'ont pas d'équivalent exact dans l'ensemble des nombres flottants. Le nombre réel $\frac{1}{3}$ (aussi écrit $0{,}333\ldots$) avec un nombre infini de décimales, est approximé par le nombre flottant $0{,}3333333433$ [1]. Comme les nombres à virgule flottante sont basés sur une représentation binaire, un nombre fini de décimales tel que $\frac{1}{10}$ (aussi écrit $0{,}1$) est représenté par le nombre flottant $0{,}1000000015$ [1]. De plus, le résultat d'une opération sur deux nombres flottants n'est pas toujours un nombre flottant, et doit donc être approximé vers un flottant proche. Par exemple, l'opération $1 + 1 \times 10^{-10}$ donne $1{,}0000000001$ sur les réels, mais est approximé par le nombre flottant $1{,}0$ en machine [1]. Cette approximation d'un réel vers un flottant est mise en place à travers un opérateur d'arrondi et s'applique sur chaque valeur et opération élémentaire d'un programme qui concerne les nombres à virgule flottante. Cette différence entre la valeur exacte d'un nombre sur les réels ou d'un calcul sur les réels et sa valeur approximée sur les flottants est appelée une erreur. Une erreur peut être négligeable et même compensée par d'autres erreurs dans un programme, mais une accumulation d'erreurs peut avoir des conséquences désastreuses. En particulier dans un système cyber-physique : un programme informatique qui contrôle et commande des entités physiques. De telles conséquences peuvent entrainer la mort d'êtres humains [General Accounting Office, 1992] ou des pertes financières importantes [Quinn, 1983] sur les marchés boursiers.

Pendant la première guerre du Golfe, en 1991, l'armée américaine a installée des missiles Patriot [General Accounting Office, 1992] afin d'intercepter les missiles Scud ennemis. Le logiciel installé dans ces missiles utilisait un entier pour compter le temps en dixième de seconde. Cet entier était ensuite multiplié par $\frac{1}{10}$ pour obtenir le temps en secondes. Le résultat de cette opération est un nombre à virgule flottante. Mais comme $\frac{1}{10}$ n'est pas un nombre flottant, arrondir sa valeur introduit une erreur de conversion. Cette erreur, certes négligeable, a des répercussions dans la suite du programme. En effet, le flottant représentant le temps en secondes était initialement stocké sur 24 bits, mais le logiciel avait été mis à jour pour avoir une représentation plus précise du temps sur 48 bits. Ces améliorations n'ont par contre pas été effectuées dans l'ensemble du code. Ainsi, la différence entre deux mesures du temps, l'une sur 24 bits et l'autre sur 48 bits, introduit une erreur

1. pour un nombre flottant sur 32 bits, avec un arrondi au plus proche pair.

non négligeable pour l'estimation de la trajectoire du missile à intercepter. Une telle erreur est proportionnelle à l'entier qui compte le temps en dixième de seconde. En laissant fonctionner ce logiciel 100 heures, la mesure du temps a dérivé d'un tiers de seconde, sous-estimant la trajectoire du missile Scud [2] en approche de 600 mètres. L'erreur dans le calcul de la trajectoire a causé la mort de 28 soldats américains et fait une centaine de blessés.

En 1982, la bourse de Vancouver [Quinn, 1983] a lancé un index boursier accumulant les valeurs de l'ensemble des 1 400 actions listées chez eux. Le résultat de cette somme était calculé jusqu'à la quatrième décimale et affiché jusqu'à la troisième. Par contre, au lieu d'arrondir ce résultat au nombre flottant le plus proche (à 3 décimales près), il était tronqué : la quatrième décimale était supprimée et oubliée immédiatement. Cette troncation était réalisée jusqu'à 3 000 fois par jour, causant une perte mensuelle de 25 $ pendant 23 mois. À la fin, la valeur de l'index calculé était de 524,811 $ alors que sa valeur exacte était de 1 098,892 $.

Certaines de ces erreurs proviennent souvent d'une mauvaise connaissance de l'arithmétique des nombres à virgule flottante par le programmeur et d'autres des limites intrinsèques aux calculs sur les nombres à virgule flottante. Pour la sécurité des logiciels critiques il est donc indispensable de pouvoir calculer et détecter les erreurs produites, en particulier pour détecter les erreurs ayant des conséquences désastreuses. Cette thèse se situe dans cette perspective et notre objectif est de calculer la sur-approximation la plus précise possible par rapport aux erreurs actuellement produites par un programme.

1.1 Contexte

Cette thèse s'inscrit donc dans l'analyse de programmes, en particulier dans l'analyse d'erreurs d'arrondi issues de calculs sur les nombres à virgule flottante. Schématiquement, l'analyse de programmes se divise en deux branches : l'analyse statique et l'analyse dynamique. L'analyse statique consiste à vérifier des propriétés d'un programme sans l'exécuter. Cette analyse est souvent réalisée à partir du code source du programme et va travailler sur une interprétation de ce dernier. Une analyse dynamique va au contraire exécuter le programme à vérifier afin d'étudier son comportement et ses effets sur son environnement. Cette analyse travaille directement sur l'exécutable du programme sur une machine donnée.De nombreuses méthodes en analyse statique et en analyse dynamique sont appliquées à l'analyse d'erreurs. Ces méthodes calculent des approximations d'erreurs pour s'assurer que le programme respecte sa spécification.

Dans le cadre de l'analyse dynamique la programmation par contraintes offre des méthodes de résolution générique pour résoudre des problèmes combinatoires [Gotlieb *et al.*, 2000, Meudec, 2001, Sy et Deville, 2003, Gotlieb et Botella, 2003, Denmat *et al.*, 2005, Collavizza et Rueher, 2007, Collavizza *et al.*, 2010], en particulier pour le test de programme. Afin de pouvoir analyser un programme avec des nombres à virgule flottante les contraintes ont été étendues aux nombres à virgule flottante [Michel *et al.*, 2001, Michel, 2002, Marre et Michel, 2010, Zitoun, 2018]. Les contraintes sur les flottants ont principalement été appliquées à la vérification de programme, en particulier à la génération de contre-exemples violant une propriété d'un programme à vérifier.

Nos travaux se situent à l'intersection entre la programmation par contraintes et l'analyse d'erreurs en vérification de programme.

2. Un missile Scud se déplace de 1 676 mètres par seconde.

1.2 Problématique : l'analyse d'erreurs en programmation par contraintes

Cette thèse s'intéresse à l'analyse d'erreurs d'arrondi issues des calculs sur les nombres à virgule flottante. Comme expliqué précédemment, une erreur dans un programme peut avoir des conséquences désastreuses. Les méthodes existantes reposent sur une sur-approximation sûre des erreurs. Elles utilisent l'interprétation abstraite [Goubault et Putot, 2006, Goubault et Putot, 2011, Moscato *et al.*, 2017, Titolo *et al.*, 2018], les séries de Taylor [Solovyev *et al.*, 2015, Solovyev *et al.*, 2018], ou encore la programmation semi-défini positive [Magron *et al.*, 2017]. Comme ces approches sont basées sur une sur-approximation un problème majeur de ces approches est celui de *faux positif*, c'est-à-dire les erreurs qui sont détectées ne sont pas atteignable en pratique. Pour éliminer ces faux positifs, il est nécessaire de calculer l'*erreur maximale* : l'erreur la plus grande, en valeur absolue, que le programme peut atteindre. Il existe déjà des approches qui cherche une sous-approximation de cette erreur à travers un processus de génération et test [Chiang *et al.*, 2014] ou bien la programmation semi-défini positive [Magron, 2018]. Calculer exactement cette erreur est très couteux dans le cas général. L'approche que nous proposons ici consiste à relaxer ce problème en cherchant un encadrement de l'erreur maximale. Cet encadrement fournit des garanties sur la distance entre la sur-approximation (ou sous-approximation) de l'erreur et l'erreur maximale et permet ainsi de réduire les faux positifs.

La programmation par contraintes a déjà été appliquée aux nombres à virgules flottantes [Michel *et al.*, 2001, Michel, 2002, Marre et Michel, 2010, Zitoun, 2018].

Dans cette thèse, nous proposons de nouvelles méthodes de résolution en programmation par contraintes bornant les erreurs produites lors des calculs via une analyse statique de programme. D'autre part, nous utilisons la programmation par contraintes afin d'encadrer l'erreur maximale sous forme de problème d'optimisation.

1.3 Contributions

Nos contributions portent sur la définition de nouvelles méthodes de résolution en programmation par contraintes dédiées à l'analyse statique d'erreurs d'arrondi dans les calculs sur les nombres à virgule flottante. Elles peuvent être séparées en deux parties. Plus précisement, nous avons d'une part défini un système de contraintes spécifique pour l'analyse des erreurs d'arrondi, et d'autre part proposé un algorithme de branch-and-bound pour trouver l'erreur maximale qu'un programme est susceptible de produire.

Dans le système de contraintes pour l'analyse d'erreur d'arrondi, nous avons étendu la notion de problème de satisfaction de contraintes (ou CSP) aux erreurs et proposé une modélisation capable de représenter ce problème. Cette modélisation introduit des contraintes sur les erreurs. Elle offre la possibilité de raisonner sur les erreurs d'un programme et d'exprimer des relations entre erreurs. La partie résolution utilise un filtrage, à base de fonctions de projection, dédiées aux erreurs. Ce filtrage profite des propriétés de l'arithmétique des flottants pour réduire les domaines représentants les erreurs. Ces domaines sont des intervalles sur les rationnels et permettent une représentation exacte de l'erreur, si le problème à résoudre est dans $\mathbb{Q}$. Sinon, l'erreur doit être approximée.

Trouver l'erreur maximale produite par un programme est un problème difficile dans le cas général. Nous proposons un algorithme qui calcule rigoureusement un encadrement de l'erreur

maximale. Cet algorithme est basé sur un branch-and-bound et calcule deux bornes de l'erreur maximale. La borne supérieure est une sur-approximation inférée à partir de notre système de contraintes pour l'analyse d'erreurs. Elle permet de détecter certains *faux positifs* et de les supprimer. La borne inférieure est atteignable et est obtenue en combinant une instanciation aléatoire des variables d'entrées du programme avec une recherche locale. Cette borne donne une garantie, pour des cas donnés, qu'un programme s'exécute avec des erreurs pouvant causer des problèmes. L'originalité de notre approche réside dans le fait que nous calculons en même temps une borne inférieure et une borne supérieure de l'erreur maximale.

Ces méthodes sont implémentées dans un solveur de contraintes sur les flottants construit à partir de FPCS [Michel *et al.*, 2001, Michel, 2002, Marre et Michel, 2010] et d'Objective-CP [Hentenryck et Michel, 2013]. Notre prototype, FErA, est capable de résoudre à la fois des problèmes de satisfaction de contraintes, en produisant des intervalles sur-approximant les erreurs issues d'un programme, et des problèmes d'optimisation sous contraintes, en produisant un encadrement rigoureux de l'erreur maximale.

Ces contributions ont fait l'objet de diverses publications dans des conférences nationales [Garcia *et al.*, 2018b], internationale [Garcia *et al.*, 2020a], et des workshops [Garcia *et al.*, 2018a, Garcia *et al.*, 2020b].

1.4 Organisation du manuscrit

Le manuscrit est organisé de la façon suivante. Le chapitre 2 présente les notions sur les nombres à virgule flottante et leurs erreurs nécessaires à la compréhension de nos travaux. La programmation par contraintes est détaillée dans le chapitre 3, en particulier pour son utilisation sur les nombres à virgule flottante. Le chapitre 4 présente les différentes techniques et méthodes utilisées en analyse d'erreurs dans le cadre de la vérification de programme. Le chapitre 5 est consacré à la programmation par contraintes dédiées aux erreurs issues de calculs sur les nombres à virgule flottante. Le chapitre 6 introduit notre algorithme pour calculer un encadrement rigoureux de l'erreur maximale. Finalement, le chapitre 7 présente des expérimentations évaluant les performances de notre approche par rapport aux autres méthodes existantes. L'implémentation du système de contraintes et de l'algorithme pour encadrer l'erreur maximale est également détaillée dans ce chapitre.

Notations

Arithmétique

$\mathbb{R}$	ensemble des nombres réels
$\mathbb{Q}$	ensemble des nombres rationnels
$\mathbb{F}$	ensemble des nombres à virgule flottante
$\mathbb{Z}$	ensemble des nombres entiers
$\mathbb{N}$	ensemble des nombres entiers naturels
$\oplus, \ominus, \otimes, \oslash, \circledcirc$	opérations arithmétiques sur $\mathbb{F}$
$+, -, \times, \div, \sqrt{}$	opérations arithmétiques sur $\mathbb{R}$
f	fonction évaluée sur $\mathbb{R}$
$\tilde{f}$	fonction évaluée sur $\mathbb{F}$
x^-	prédécesseur du nombre flottant x
x^+	successeur du nombre flottant x
∞	Infini
NaN	*Not-a-Number*, nombre spécial dans $\mathbb{F}$
rnd	opérateur d'arrondi dans $\mathbb{F}$
$\lceil x \rceil$	arrondi au plus petit entier strictement supérieure à x
$\lvert x \rvert$	valeur absolue d'une valeur x
β	base du format d'un nombre à virgule flottante

Intervalles et formes affines

$\mathbf{a}$	intervalle d'une variable a
$\underline{\mathbf{a}}$	borne inférieure de l'intervalle $\mathbf{a}$
$\overline{\mathbf{a}}$	borne supérieure de l'intervalle $\mathbf{a}$
$[a, b[$	intervalle semi-ouvert sur la borne supérieure
$\hat{a}$	forme affine d'une variable a
α_i^x	*i*-ème coefficient, dans $\mathbb{R}$, pour une forme affine $\hat{x}$
ϵ_i	*i*-ème symbole de bruit, dans l'intervalle $[-1, 1]$, pour une forme affine $\hat{x}$

Programmation par contraintes

$\langle X, D, C \rangle$	un problème de satisfaction de contraintes (CSP)
X	ensemble des variables d'un CSP
D	ensemble des domaines d'un CSP
C	ensemble des contraintes d'un CSP
D_x, $\mathbf{x}$	domaine des valeurs de la variable x
D_{e_x}, $\mathbf{e_x}$	domaine d'erreurs de la variable x
Φ	processus de filtrage

Branch-and-bound

e	erreur à maximiser
e^*	borne inférieure atteignable de l'erreur maximale
$\overline{e}$	borne supérieure de l'erreur maximale
S	ensemble ordonné des bornes inférieures atteignables calculées
$\mathbb{B}$	boîte, *i.e.,* le produit cartésien des domaines de valeurs et des domaines d'erreurs
L	ensemble des boîtes à traiter
$\overline{e}^D$	borne supérieure des boîtes écartées

État de l’art

CHAPITRE 2

Nombres à virgule flottante

Les nombres à virgule flottante, ou nombres flottants, sont une approximation représentable en machine des nombres réels. Ils offrent un bon compromis entre plage de valeurs et précision. Dans ce chapitre, Nous introduisons les notions de base liées aux nombres flottants telles que leur représentation, leur arithmétique, ainsi que certaines de leurs spécificités. La notion d'erreur, inhérente aux calculs sur les flottants, est également présentée avec les mesures permettant de la quantifier.

Les nombres à virgule flottante ont été introduit comme une approximation discrète et finie des nombres réels. En effet, l'ensemble des nombres réels étant continu et infini, il est impossible de le représenter dans un ordinateur à mémoire finie. Le standard IEEE 754 [IEEE, 2008] normalise les nombres à virgule flottante, leurs formats, et les propriétés arithmétiques qui doivent être respectées en machine. Dans ce chapitre nous présentons les notions de base liées aux nombres à virgule flottante nécessaires à la compréhension de nos travaux. Ces notions couvrent la représentation des nombres flottants, les particularités de leur arithmétique par rapport à celle des réels, ainsi que la notion d'erreur inhérente aux calculs sur les nombres à virgule flottante.

2.1 Représentation

Un nombre à virgule flottante est représenté par le triplet $\langle s, m, e\rangle$ où s est le signe, m la mantisse, et e l'exposant. Le signe s prend ses valeurs dans l'ensemble $\{0, 1\}$, la mantisse m est de la forme $d_1 \,.\, d_2 \ldots d_p$ avec $d_i \in \{0, 1\}$, où p est la précision du nombre à virgule flottante, et l'exposant e est un entier naturel borné en fonction du format de représentation utilisé. La mantisse m est toujours précédé d'un bit implicit d_0 qui prend sa valeur dans l'ensemble $\{0, 1\}$. L'utilisation d'un biais b permet d'obtenir un exposant négatif, et donc de représenter à la fois des grands et des petits nombres. La valeur d'un nombre à virgule flottante binaire est donnée, dans le cas général, par la définition 2.1.1.

Définition 2.1.1 (Nombre à virgule flottante)**.** $(-1)^s \times d_0.m \times 2^{e-b}$

2.1.1 Formats

Le standard IEEE 754 [IEEE, 2008] défini plusieurs formats en machines pour les nombres à virgule flottante. En pratique, deux de ces formats sont le plus souvent utilisés : un format simple précision (souvent noté **float**) sur 32 bits et un format double précision (souvent noté **double**) sur 64 bits. Le tableau 2.1 donne la taille des éléments du triplet $\langle s,m,e\rangle$ et la valeur du biais b pour chacun de ces formats.

Table 2.1 – Principaux formats des nombres à virgule flottante dans le standard IEEE 754

Format	s	m	e	b
simple (32 bits)	1 bit	23 bits	8 bits	127
double (64 bits)	1 bit	52 bits	11 bits	1023

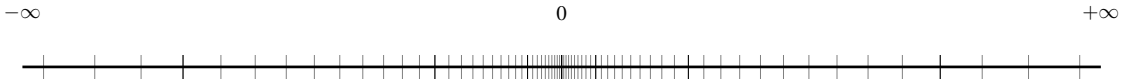
2.1.2 Nombres normalisés

Les nombres à virgule flottante *normalisés* couvrent la plus grande partie de la plage de flottants disponibles en machine. Un nombre normalisé (voir définition 2.1.2) s'écrit avec son bit implicite, d_0, à 1, et son exposant e prend une valeur entre 0 et sa valeur maximale [1] exclus.

Définition 2.1.2 (Nombre à virgule flottante normalisé)**.** $(-1)^s \times 1 \,.\, m \times 2^{e-b}$

1. La valeur maximale est de 255 pour un flottant 32 bits et de 2047 pour un flottant 64 bits.

Figure 2.1 – Distribution des nombres à virgule flottante



Les nombres normalisés ont une distribution non uniforme : la densité de flottants est plus grande vers 0 que vers les infinis, comme montré dans la figure 2.1. En effet, la moitiée des nombres à virgule flottante se trouve dans l'intervalle $[-1, 1]$. Ces nombres flottants sont regroupés par *binade* [2], où les nombres flottants ont tous le même exposant e. Dans une binade, les nombres flottants sont à égale distance et chaque binade contient la même quantité de nombres flottants. Cette distribution variable des nombres à virgule flottante permet de représenter de grande plage de valeurs.

L'exemple 2.1.1 donne quelques valeurs de nombres à virgule flottante normalisés.

Exemple 2.1.1 – Le plus grand nombre à virgule flottante normalisé positif sur 32 bits est $0111\,1111\,0111\,1111\,1111\,1111\,1111\,1111_2 \approx 3{,}402\,823\,466\,39 \times 10^{38}$ avec $s = 0$, $m = 2 - 2^{-23}$ et $e = 254$, tandis que le plus petit nombre flottant normalisé positif sur 32 bits est $0000\,0000\,1000\,0000\,0000\,0000\,0000\,0000_2 \approx 1{,}175\,494\,350\,8 \times 10^{-38}$ avec $s = 0$, $m = 2^0$ et $e = 1$.

2.1.3 Nombres dénormalisés

Les nombres à virgule flottante *dénormalisés* sont introduit dans le standard IEEE 754 pour éviter d'arrondir à 0 le résultat d'une opération qui serait plus petit que le plus petit nombre normalisé représentable dans le format utilisé. Un nombre dénormalisé (voir définition 2.1.3) s'écrit avec son bit implicite, d_0, et son exposant e à 0.

Définition 2.1.3 (Nombre à virgule flottante dénormalisé)**.** $(-1)^s \times 0\,.\,m \times 2^{0-b}$

L'utilisation de ces nombres offre une transition plus lente vers 0 et conserve une certaine précision sur le résultat d'une opération. Les nombres dénormalisés sont distribués de manière égale entre 0 et le plus petit nombre normalisé positif (de même pour les négatifs). L'exemple 2.1.2 donne quelques valeurs de nombres à virgule flottante dénormalisés.

Exemple 2.1.2 – Le plus grand nombre à virgule flottante dénormalisé positif sur 32 bits est $0000\,0000\,0111\,1111\,1111\,1111\,1111\,1111_2 = 0{,}999\,999\,988 \times 2^{-126} \approx 1{,}175\,494\,336\,7 \times 10^{-38}$. Il est proche du plus petit flottant normalisé positif $2^{-126} \approx 1{,}175\,494\,350\,8 \times 10^{-38}$. Le plus petit nombre à virgule flottante dénormalisé positif sur 32 bits est $0000\,0000\,0000\,0000\,0000\,0000\,0000\,0001_2 \approx 1{,}401\,298\,464\,3 \times 10^{-45}$. En dessous de ce nombre, le prochain nombre à virgule flottante est 0.

2.1.4 Nombres spéciaux

Le standard IEEE 754 défini plusieurs nombres spéciaux servant de représentations aux zéros, aux infinis, et aux *Not-a-Number* (ou NaN). Un nombre à virgule flottante est égal à 0 quand son exposant et sa mantisse sont tout les deux égaux à 0. Le zéro partage donc avec les nombres

2. Une binade est représentée par l'intervalle semi-ouvert $[2^n, 2^{n+1}[$.

dénormalisés la valeur de l'exposant. À noter que cette représentation du zéro peut avoir un signe positif, noté $+0$, ou signe négatif, noté -0. En général, la seule différence entre -0 et $+0$ est la propagation du signe à travers les autres opérations arithmétiques d'une expression.

L'infini, ou ∞, se caractérise par un exposant fixé à la valeur maximale et une mantisse à 0. La valeur maximale de l'exposant n'est jamais utilisé pour un nombre à virgule flottante numérique. Un infini peut être positif, noté $+\infty$, ou négatif, noté $-\infty$. Ces valeurs sont obtenues lorsque le résultat d'une opération a un dépassement de capacité (*overflow* en anglais) : la valeur du résultat est trop grande pour être représentée dans le format choisi.

Un NaN, est représenté avec un exposant à sa valeur maximale et une mantisse non nulle. Le signe n'est pas important pour cette valeur et peut être positif ou négatif. Un NaN est le plus souvent utilisé pour signaler le résultat d'une opération non définie sur les réels : $\frac{0}{0}, +\infty - +\infty, \sqrt{-1}, \ldots$ Il existe deux types de NaN : les qNaN et les sNaN. Un qNaN, pour *quiet* NaN, est silencieux et se propage à travers la plupart des opérations sans lever d'exception. La valeur de son premier bit de mantisse, d_1 est toujours de 1, alors que les autres bits prennent une valeur dans l'ensemble $\{0, 1\}$. Un sNaN, pour *signaling* NaN, lève une exception pour l'opération qui l'a produit et, si approprié, est transformé en qNaN afin d'être propagé dans le reste des opérations. Contrairement à un qNaN, le premier bit de la mantisse d'un sNaN est nul, tandis que le reste de sa mantisse contient au moins un bit non nul. Le support des sNaN est laissé à l'appréciation de l'implémentation au niveau du langage de programmation. En général, les sNaN sont utilisés pour détecter l'utilisation de données non initialisées [IEEE, 2008], *e.g.,* en initialisant ces données avec des sNaN avant l'exécution du programme. La table 2.2 donne la représentation des nombres spéciaux en fonction du format de nombres à virgule flottante choisi.

Table 2.2 – Représentation des nombres spéciaux

Valeur	e	m	s
$+0$	0	0	0
-0	0	0	1
$+\infty$	e_{max}	0	0
$-\infty$	e_{max}	0	1
NaN	e_{max}	> 0	$\{0, 1\}$

2.2 Arithmétique des nombres à virgule flottante

Les nombres flottants étant une approximation représentable en machine des nombres réels, leur arithmétique est caractérisée par des règles spécifiques. Elle utilise un opérateur d'arrondi pour approximer une valeur sur les réels ou le résultat d'une opération vers un nombre à virgule flottante proche. De ce fait, certaines propriétés classiques sur les réels, comme l'associativité ou la distributivité, ne sont pas conservées sur les flottants. L'absence de ces propriétés rend d'autant plus difficile la réécriture d'une expression sur les flottants sans impacter son résultat.

2.2.1 Opérateur d'arrondi

Le standard IEEE 754 propose un opérateur d'arrondi, noté rnd, pour arrondir un nombre réel vers un nombre flottant proche. Cet opérateur d'arrondi est appliqué sur chaque valeur et opération

élémentaire. En fonction du mode d'arrondi choisi, le nombre réel sera arrondi vers le plus petit (resp. plus grand) nombre flottant strictement plus grand (resp. plus petit) que le nombre réel à arrondir.

Le standard IEEE 754 [IEEE, 2008] introduit la notion d'*arrondi correct* pour certaines opérations. Une opération mathématique est dite *correctement arrondie* si son résultat sur les flottants est la valeur représentable la plus proche, en fonction du mode d'arrondi choisi, du résultat mathématique exact. Cela revient à évaluer une opération exactement, puis à arrondir son résultat vers un nombre à virgule flottante représentable proche. L'arrondi correct est défini pour les opérations arithmétiques suivantes : $\oplus, \ominus, \otimes, \oslash, \circledcirc$. L'exemple 2.2.1 illustre la notion d'arrondi correct sur une opération élémentaire.

Exemple 2.2.1 – Soit l'addition $x \oplus y$ sur les nombres à virgule flottante. Cette opération est arrondie correctement d'après le standard IEEE 754 [IEEE, 2008]. L'équation 2.1 donne la formule pour calculer cette opération correctement.

$$x \oplus y = \mathrm{rnd}(x + y) \tag{2.1}$$

L'opération est d'abord évaluée exactement, $x + y$, puis son résultat est arrondi correctement à l'aide de l'opérateur d'arrondi, rnd.

D'autres opérations mathématiques, comme les fonctions transcendantes ($\sin$, $\cos$, $\tan$, $\log$, ...), n'ont pas obligation d'être arrondies correctement d'après le standard IEEE 754. Cet arrondi est laissé à l'appréciation de l'implémentation de l'arithmétique des nombres à virgule flottante et peut donc varier grandement d'un système à l'autre.

2.2.1.1 Modes d'arrondi

Le standard IEEE 754 défini 5 modes d'arrondi répartis en deux catégories : l'arrondi dirigé et l'arrondi au plus proche.

L'arrondi *dirigé* va, à partir d'une direction donnée $+\infty$, $-\infty$, ou zéro, arrondir une valeur x vers un nombre à virgule flottante proche. L'arrondi vers $+\infty$, ou arrondi vers le haut, revient à arrondir vers le plus petit nombre à virgule flottante supérieur ou égal à x. L'arrondi vers $-\infty$, ou arrondi vers le bas, revient à arrondir vers le plus grand nombre à virgule flottante inférieur ou égal à x. L'arrondi vers zéro, ou troncation, est équivalent à un arrondi vers $-\infty$ (resp. $+\infty$), si le nombre à arrondir est positif (resp. négatif).

L'arrondi *au plus proche* revient à arrondir x vers le nombre flottant le plus proche. Contrairement aux arrondis dirigés, cet arrondi prend le nombre à virgule flottante à moindre distance de x, qu'il soit plus grand ou plus petit que x. Si la valeur à arrondir est à égale distance des nombres flottants les plus proches, deux choix sont possibles : arrondir vers le nombre *pair* ou *loin de zéro*. Le premier va arrondir vers le nombre à virgule flottante avec une mantisse paire, tandis que l'arrondi *loin de zéro* va effectuer un arrondi vers le haut (resp. vers le bas) dans le cas d'un nombre positif (resp. négatif). Par défaut, les ordinateurs effectuent un arrondi au plus proche, avec égalité vers le nombre pair. Il est souvent noté comme un *arrondi au plus proche pair*. Cet arrondi est celui choisi pour le reste du manuscrit, sauf si précisé autrement.

La table 2.3 reprend l'idée des modes d'arrondi et présente des exemples d'arrondi de nombres à virgule flottante vers des entiers pour les différents modes.

Table 2.3 – Arrondi de flottants vers des entiers pour les 5 modes d'arrondi du IEEE 754

Mode d'arrondi	$+1.5$	$+2.5$	-1.5	-2.5
au plus proche pair	$+2$	$+2$	-2	-2
au plus proche loin de zéro	$+2$	$+3$	-2	-3
vers zéro	$+1$	$+2$	-1	-2
vers $+\infty$	$+2$	$+3$	-1	-2
vers $-\infty$	$+1$	$+2$	-2	-3

2.2.2 ulp et ufp

L'ulp, ou *unit in the last place*, et l'ufp, ou *unit in the first place*, sont des unités de mesures de la précision des nombres à virgule flottante. Ils correspondent respectivement au dernier et au premier bit d'un nombre flottant.

2.2.2.1 ulp

L'ulp est souvent utilisé pour borner l'erreur des opérations sur les nombres à virgule flottante. Cette mesure correspond à la distance entre deux flottants consécutifs. La définition 2.2.1 est issue de [Muller, 2005] et offre un bon compromis entre les différentes définitions de l'ulp proposées par Kahan [Kahan, 2004], Harrison [Harrison, 1999], et Goldberg [Goldberg, 1991]. Cette définition est valide pour x positif. Elle s'étend par symétrie aux valeurs négatives.

Définition 2.2.1 (ulp – *unit in the last place*)**.** Si x est un nombre réel qui se trouve entre deux flottants consécutif a et b, sans être égal à l'un d'entre eux, alors $\mathrm{ulp}(x) = |b - a|$, sinon $\mathrm{ulp}(x)$ est la distance entre les deux flottants les plus proches de x. De plus, $\mathrm{ulp}(\mathrm{NaN})$ est égal à NaN et $\mathrm{ulp}(0)$ est égal à 0.

$$\mathrm{ulp}(x) = \begin{cases} 0 & \text{si } x = 0 \\ |b-a| & \text{si } a < x < b, b = a^+ \text{ et } |x| \leq L \\ |x - x^-| & \text{si } x = a \vee x = b \text{ et } |x| \leq L \\ L - L^- & \text{sinon} \end{cases} \tag{2.2}$$

Dans l'équation 2.2, a et b représentent les deux nombres à virgule flottante les plus proches du réel x. a^+ (resp. a^-) désigne le nombre à virgule flottante successeur (resp. prédecesseur) de a. L est le plus grand nombre à virgule flottante numérique représentable dans un format donné.

2.2.2.2 ufp

L'ufp a été introduite par S. M. Rump dans [Rump *et al.*, 2008] comme une alternative à l'ulp pour borner l'erreur issue des opérations sur les nombres à virgule flottante. Elle a été utilisée pour calculer des approximations des erreurs dans [Jeannerod, 2015, Jacquemin *et al.*, 2018]. La définition 2.2.2 donne la formule de l'ufp.

Définition 2.2.2 (ufp – *unit in the first place*)**.**

$$\mathrm{ufp}(x) = \begin{cases} 0 & \text{si } x = 0 \\ \beta^{\lceil \log_\beta |x| \rceil} & \text{sinon} \end{cases}$$

Pour $x \in \mathbb{R}$ et avec β la base du format des nombres à virgule flottante. $\lceil x \rceil$ est l'arrondi au plus petit entier strictement supérieur à x. En général, le standard IEEE 754 privilégie β égal à 2.

L'ulp et l'ufp sont liées par l'équation 2.3, donnée dans [Jeannerod, 2015], où u est l'*unité d'arrondi* (*unit roundoff* en anglais) et est égale à $\frac{1}{2}\beta^{1-p}$, p étant la précision d'un nombre à virgule flottante.

$$\mathrm{ulp}(x) = 2u\,\mathrm{ufp}(x) \tag{2.3}$$

2.2.3 Propriétés

L'arithmétique des nombres à virgule flottante est différente de celle des nombres réels et ne conserve pas les mêmes propriétés. En mathématique, l'addition et la multiplication de réels sont associatives, mais cette propriété n'est pas toujours verifiée sur les nombres à virgule flottante, comme affirmé par l'équation 2.4.

$$\exists a,b,c \in \mathbb{F}, (a \oplus b) \oplus c \neq a \oplus (b \oplus c) \tag{2.4}$$

Cette perte d'associativité est due à l'opérateur d'arrondi qui approxime le résultat de chaque opération et introduit une imprécision dans la suite des calculs. L'exemple 2.2.2 illustre la perte d'associativité à partir d'une expression calculant trois additions.

Exemple 2.2.2 – Prenons $a = 1 \times 10^{-1}$, $b = 2 \times 10^{-1}$, et $c = 3 \times 10^{-1}$ en double précision avec un arrondi au plus proche pair.

$$(a \oplus b) \oplus c = 6{,}000\,000\,000\,000\,000\,9 \times 10^{-1}$$
$$a \oplus (b \oplus c) = 5{,}999\,999\,999\,999\,999\,8 \times 10^{-1}$$

Ici, le choix de l'addition évaluée en premier impacte le résultat de l'expression sur les nombres à virgule flottante.

La distributivité est une autre propriété de l'addition et de la multiplication sur les réels qui n'est pas conservée sur les nombres à virgule flottante, comme illustré par l'équation 2.5.

$$\exists a,b,c \in \mathbb{F}, a \otimes (b \oplus c) \neq a \otimes b \oplus a \otimes c \tag{2.5}$$

Comme pour la perte d'associativité, la différence entre ces deux expressions vient de l'opérateur d'arrondi. L'exemple 2.2.3 montre cette perte de distributivité sur une expression calculant le produit d'un nombre flottant avec la somme de deux autres nombres flottants.

Exemple 2.2.3 – Prenons $a = 1{,}00 \times 10^{2}$, $b = 1 \times 10^{-1}$, et $c = 2 \times 10^{-1}$ en double précision avec un arrondi au plus proche pair.

$$a \otimes (b \oplus c) = 3{,}000\,000\,000\,000\,000\,355\,3 \times 10^{1}$$
$$a \otimes b \oplus a \otimes c = 3{,}0 \times 10^{1}$$

L'absence d'associativité et de distributivité dans l'arithmétique des nombres à virgule flottante empêche le plus souvent la réécriture d'une expression sans impacter son résultat, contrairement à ce qui pourrait se faire sur les réels. Cette différence entre l'arithmétique des flottants et l'arithmétique des réels vient de l'opérateur d'arrondi. Un tel opérateur, utilisé sur chaque opération élémentaire, est à l'origine des erreurs sur les nombres à virgule flottante.

2.3 Erreurs et nombres à virgule flottante

Les nombres à virgule flottante sont une approximation représentable en machine des nombres réels. L'impossibilité de représenter l'ensemble des réels dans un ordinateur introduit de l'imprécision dans le résultat de calculs numériques. Cette imprécision est appelée une erreur et provient de l'utilisation d'un opérateur d'arrondi sur chaque opération élémentaire d'un programme. En général, l'erreur sur une seule opération élémentaire est bornée. Le standard IEEE 754 requiert même que les opérations arithmétiques ($\oplus, \ominus, \otimes, \oslash, \circledcirc$) soient toujours calculées à $\frac{1}{2}\,\mathrm{ulp}$ du résultat exact sur les réels. Le problème est que les erreurs vont s'accumuler lorsque une expression avec plusieurs opérations est évaluée. Ces erreurs peuvent entrainer un comportement différent entre l'exécution du programme sur les nombres à virgule flottante et son exécution idéale sur les nombres réels. Il est donc important de pouvoir quantifier ces erreurs afin d'évaluer l'exécution du programme.

2.3.1 Les différentes erreurs

Il existe plusieurs mesures de l'erreur telles que l'erreur *absolue* ou l'erreur *relative*. Informellement, ces deux mesures quantifient la différence entre le résultat d'une expression évaluée sur les réels et le résultat de la même expression calculée sur les nombres à virgule flottante, respectivement en valeur absolue et en valeur relative. En plus de ces formules calculant l'erreur d'une expression, il est nécessaire d'avoir une formulation de l'erreur introduite par l'application de l'opérateur d'arrondi. Il existe deux modèles pour cette erreur : le premier est basé sur les erreurs absolue et relative maximales tandis que le second utilise l'ulp.

2.3.1.1 Erreur absolue

L'erreur *absolue* est la distance entre une valeur exacte sur les réels et sa valeur approximée sur les flottants. La définition 2.3.1 donne la formule pour calculer l'erreur absolue.

Définition 2.3.1 (Erreur absolue)**.** $\varepsilon_a = |f - \tilde{f}|$ où f et $\tilde{f}$ sont respectivement le résultat de l'expression sur $\mathbb{R}$ et sur $\mathbb{F}$.

Cette erreur représente la magnitude entre les deux valeurs considérées, mais doit être comparée avec la taille de f et de $\tilde{f}$, comme illustré dans l'exemple 2.3.1.

Exemple 2.3.1 – Pour une valeur en machine $\tilde{f} = 2^{127}$ et une valeur exacte $f = 2^{127} + 2^{104}$, l'erreur absolue est égale à 2^{104}. Cette erreur, certes grande, n'est pas très importante par rapport à la taille de $\tilde{f}$ et de f. D'ailleurs, sur les nombres à virgule flottante en simple précision avec un arrondi au plus proche pair, f est le successeur de $\tilde{f}$. Ici, l'erreur absolue est la distance entre un nombre à virgule flottante et son voisin.

2.3.1.2 Erreur relative

L'erreur *relative* est égale à l'erreur absolue divisée par la valeur exacte. La définition 2.3.2 donne la formule pour calculer l'erreur relative. Contrairement à l'erreur absolue, elle prend en compte la taille des valeurs mesurées et n'est pas dépendante de leur magnitude.

Définition 2.3.2 (Erreur relative)**.** $\varepsilon_r = \left| \frac{f - \tilde{f}}{f} \right|$ où f et $\tilde{f}$ sont respectivement le résultat de l'expression sur $\mathbb{R}$ et sur $\mathbb{F}$.

L'erreur relative peut toutefois porter à confusion autour de zéro. En effet, quand f est proche zéro, l'erreur relative calculée est grande. L'exemple 2.3.2 illustre ce cas.

Exemple 2.3.2 – Pour une valeur en machine $\tilde{f} = 2^{-10}$ et une valeur exacte $f = 2^{-20}$ sur les nombres à virgule flottante en simple précision avec un arrondi au plus proche, l'erreur relative est égale à 1 023, tandis que l'erreur absolue est égale à $\frac{1023}{1048576} \approx 9{,}756\,088\,256\,8 \times 10^{-4}$.

L'erreur relative n'est pas définie pour f égal à zéro : son calcul contient alors une division par zéro. Pour éviter ce cas particulier, il est possible de découper les domaines de valeurs des fonctions f et $\tilde{f}$ afin d'isoler le zéro. Cela permet de calculer l'erreur relative pour une partie des valeurs, mais ne résout pas le problème de division par zéro. Le dénominateur de la division f peut également être remplacé par $f + \delta$, avec un δ très petit. La division par zéro n'apparait plus dans cette formule, mais le résultat obtenu n'est pas correct par rapport aux valeurs de f.

2.3.1.3 Modèle pour l'erreur d'arrondi

L'erreur introduite par l'application d'un opérateur d'arrondi peut être bornée de deux façons différentes : en fonction des erreurs absolue et relative maximales par rapport au format considéré ou bien avec l'ulp (voir définition 2.2.1). La première borne [Goldberg, 1991] est donnée dans la définition 2.3.3 et dépend de ϵ et de δ qui sont respectivement l'erreur maximale relative et l'erreur maximale absolue produite par un opérateur d'arrondi.

Définition 2.3.3 (Modèle d'arrondi en format)**.** $\mathrm{rnd}(x) = x \times (1 + e) + d$ avec $e \leq \epsilon$, $d \leq \delta$, et $e \times d = 0$.

La table 2.4 donne les valeurs de ϵ et de δ en fonction du format à virgule flottante utilisé et pour un arrondi au plus proche. Les valeurs pour d'autres formats sont obtenues en multipliant les entrées de la table par 2.

Table 2.4 – Paramètres pour l'arrondi au plus proche

Format	ϵ	δ
32 bits	2^{-24}	2^{-150}
64 bits	2^{-53}	2^{-1075}

Le modèle à base d'ulp est dépendant de l'opération sur laquelle l'arrondi est appliqué. La définition 2.3.4 est valide pour les opérations arithmétiques de base d'après la norme IEEE 754 [IEEE, 2008].

Définition 2.3.4 (Modèle d'arrondi en ulp)**.** $|\mathrm{rnd}(x) - x| \leq \frac{1}{2}\,\mathrm{ulp}(x)$ pour un mode d'arrondi au plus proche, sinon $|\mathrm{rnd}(x) - x| \leq \mathrm{ulp}(x)$ pour les arrondis dirigés.

Le résultat d'une fonction transcendante ($\cos, \sin, \tan, \log, \ldots$) sur les nombres à virgule flottante simple précision, est le plus souvent arrondi à un ulp de la valeur exacte sur les réels. Toutefois, le standard IEEE 754 n'impose pas d'arrondi correct pour ces fonctions : l'arrondi varie donc d'une implémentation à l'autre.

2.3.2 Phénomènes liés à l'erreur

En plus des erreurs dues à l'arithmétique des nombres à virgule flottante, les deux phénomènes principaux liés aux erreurs d'arrondi sont l'*absorption* et la *cancellation*.

2.3.2.1 Absorption

Une *absorption* se produit lors de l'addition, ou de la soustraction, de deux nombres à virgule flottante de grandeur différente. Le résultat d'une telle opération est égal à la valeur du nombre ayant la plus grande magnitude. Une telle perte de précision est causée par l'opérateur d'arrondi qui va arrondir le résultat exact de l'opération vers l'opérande de plus grande magnitude. Cela se produit car la valeur de la plus petite opérande n'est pas assez grande pour que le résultat de l'opération, avant l'arrondi, soit à une distance supérieure à $\frac{1}{2}\,\mathrm{ulp}$ de la plus grande opérande. L'example 2.3.3 illustre le phénomène d'absorption par une addition de deux nombres à virgule flottante de magnitude différente.

Exemple 2.3.3 – L'addition suivante est calculée sur les nombres flottants simple avec un arrondi au plus proche pair.

$$1 \times 10^8 \oplus 1 = 1 \times 10^8$$

La même opération sur les réels, produit un résultat de $1{,}000\,000\,01 \times 10^8$, ce qui veut dire que l'erreur absolue pour cette addition est égale à la plus petite opérande de l'opération, *i.e.,* 1. Pour cette opération, la plus petite opérande 1 est absorbée par la plus grande car le successeur de 1×10^8 est $1 \times 10^8 \oplus 8$.

Une accumulation d'absorption peut avoir des conséquences catastrophiques, comme dans l'incident des missiles Patriot [General Accounting Office, 1992]. Par exemple, si l'expression causant une absorption est utilisée dans une boucle, à chaque itération l'erreur produite par l'absorption s'accumule et augmente la distance entre le résultat exact sur $\mathbb{R}$ et le résultat calculé en machine sur $\mathbb{F}$.

2.3.2.2 Cancellation

Une *cancellation*, ou perte de précision relative, se produit lors de la soustraction de deux nombres à virgule flottante proches. Cette perte de précision ne vient pas de la soustraction en elle même, l'opération étant calculée le plus souvent exactement [Sterbenz, 1974], mais d'une amplification d'erreurs issues des opérations précedentes. Il existe deux types de cancellation : la cancellation *bénigne* et la cancellation *catastrophique*. La cancellation bénigne se produit lorsque

les valeurs des opérandes sont exactes, *i.e.,* sans erreurs d'arrondi. La soustraction engendre seulement une perte de chiffres significatifs dans le résultat. Au contraire, une cancellation catastrophique se produit lorsque les opérandes, le plus souvent issues d'autres opérations, ont une erreur d'arrondi. Comme dans le cas bénin, le résultat de la soutraction a une perte de chiffres significatifs, mais les erreurs des opérandes sont également propagées au résultat. Utiliser ce résultat dans d'autres opérations peut également amplifier l'erreur générée et aggraver la déviation par rapport au résultat exact sur les réels. L'exemple 2.3.4 illustre le principe de la cancellation catastrophique sur le polynôme de Rump.

Exemple 2.3.4 – Le polynôme de Rump [Rump, 1988], rappelé dans l'équation 2.6, est un exemple classique proposé par S. M. Rump en 1988 pour illustrer le concept de cancellation. En prenant les valeurs $a = 77617$ et $b = 33096$, le résultat du polynôme sur les nombres à virgule flottante dévie largement par rapport au résultat sur les nombres réels. Pour ces valeurs d'entrée, le résultat entre $\mathbb{R}$ et $\mathbb{F}$ est de signe opposé et de magnitude très différente. Ici, la formule de l'erreur absolue (voir définition 2.3.1) est appliquée sans valeur absolue, afin d'illustrer la direction de la déviation à travers le signe de l'erreur.

$$p(a,b) = 333{,}75b^6 + a^2(11a^2b^2 - b^6 - 121b^4 - 2) + 5{,}5b^8 + \frac{a}{2b} \tag{2.6}$$

$$p(77617, 33096) = -\frac{54767}{66192} \approx -8{,}273\,960\,56 \times 10^{-1} \tag{2.7}$$

$$\tilde{p}(77617, 33096) \approx 6{,}338\,253\,001\,141\,1 \times 10^{29} \tag{2.8}$$

L'erreur absolue pour ces valeurs d'entrée est donnée dans l'équation 2.9.

$$e_p = -\frac{41954164265153480271934889285124096}{66192} \approx -6{,}338\,253\,001\,141\,1 \times 10^{29} \tag{2.9}$$

Pour ce polynôme, la cancellation catastrophique se produit sur la seconde addition. Soit $p_1 = 333{,}75b^6 + a^2(11a^2b^2 - b^6 - 121b^4 - 2)$ et $p_2 = 5{,}5b^8$.

En évaluant p_1 sur $\mathbb{F}$ et sur $\mathbb{R}$, puis en faisant la différence de ces deux valeurs, nous obtenons l'erreur absolue pour le terme p_1.

$$p_1 = 79171113406689613611011347015249428 48 \approx 7{,}917\,111\,340\,669\,0 \times 10^{36} \tag{2.10}$$

$$\tilde{p_1} \approx -7{,}917\,111\,189\,900\,1 \times 10^{36} \tag{2.11}$$

$$e_{p_1} = 158342225305690679740340938223491481 60 \approx 1{,}583\,422\,253\,056\,9 \times 10^{37} \tag{2.12}$$

Le même procédé est appliqué pour p_2.

$$p_2 = -79171113406689613611011347015249428 50 \approx -7{,}917\,111\,340\,669\,0 \times 10^{36} \tag{2.13}$$

$$\tilde{p_2} \approx 7{,}917\,111\,823\,725\,4 \times 10^{36} \tag{2.14}$$

$$e_{p_2} = -158342231643943680881487945707007508 50 \approx -1{,}583\,422\,316\,439\,4 \times 10^{37} \tag{2.15}$$

Les termes p_1 et p_2 sont de signes opposés et leurs valeurs sont proches [Sterbenz, 1974]. L'addition de p_1 et p_2 s'effectue donc exactement mais provoque une perte importante de chiffres significatifs dans le résultat p_3, tout en y propageant les erreurs e_{p_1} et e_{p_2}.

$$p_3 = -2 \tag{2.16}$$

$$\tilde{p_3} \approx 6{,}338\,253\,001\,141\,1 \times 10^{29} \tag{2.17}$$

$$e_{p_3} = -63382530011411470074835160269 0 \approx -6{,}338\,253\,001\,141\,1 \times 10^{29} \tag{2.18}$$

La valeur de $\tilde{p_3}$ est principalement issue des erreurs e_{p_1} et e_{p_2} : son erreur e_{p_3} a la même valeur, mais de signe opposé. L'utilisation de ce résultat dans le reste du polynôme produit une amplification de l'erreur, qui se retrouve dans l'erreur finale e_p.

2.4 Conclusion

Les nombres à virgule flottante sont une approximation représentable en machine des nombres réels. Certaines propriétés de l'arithmétique des réels ne sont pas conservées sur l'arithmétique des nombres flottants, comme l'associativité ou la distributivité. L'arithmétique des nombres à virgule flottante utilise un opérateur d'arrondi pour approximer une valeur sur les réels ou le résultat d'une opération vers un nombre flottant proche. Cet arrondi introduit une imprécision, appelée une erreur, dans les calculs sur les flottants. Il existe de nombreuses mesures de l'erreur : comme l'erreur absolue ou l'erreur relative, pour quantifier la déviation d'une expression sur les flottants par rapport aux réels. Ces erreurs sont à l'origine de déviation de calcul entre $\mathbb{R}$ et $\mathbb{F}$, pouvant avoir des conséquences graves, dans de nombreux programmes.

Une description complète et détaillée de l'arithmétique des nombres à virgule flottante et de ses propriétés est disponible dans [Goldberg, 1991, IEEE, 2008, Muller *et al.*, 2018].

CHAPITRE 3

Programmation par contraintes

Dans ce chapitre, nous présentons la programmation par contraintes, un paradigme permettant de résoudre des problèmes combinatoires difficiles. Elle offre une séparation claire entre modélisation et résolution du problème. La modélisation consiste à représenter un problème sous forme mathématique, exprimé sous forme de contraintes, tandis que la résolution utilise deux mécanismes, la propagation et la recherche, pour trouver une solution au problème. Nous présentons les techniques de résolution utilisées pour résoudre des problèmes sur le discret et sur le continu. En particulier, nous présentons le cadre des contraintes adaptées aux nombres à virgule flottante. La programmation par contraintes sur les nombres flottants est principalement utilisée pour la vérification de programme.

La programmation par contraintes [Montanari, 1974], abbrégée PPC, est un paradigme proposant des méthodes de résolution générique pour résoudre des problèmes combinatoires difficiles en ordonnancement [Baptiste *et al.*, 2012], en biologie [García-Martín *et al.*, 2013], ou en tournée de véhicules [Rabbouch *et al.*, 2019]. Dans ce chapitre, nous présentons les techniques utilisées en programmation par contraintes pour résoudre des problèmes sur le discret et sur le continu. Nous présentons en particulier la programmation par contraintes pour les nombres à virgule flottante appliqué à la vérification de programme.

L'originalité de la programmation par contraintes est de proposer une séparation claire entre la modélisation et la résolution d'un problème. La modélisation consiste à représenter un problème sous forme mathématique, de façon à exprimer des relations entre les variables du problème. Ces relations sont appelées des contraintes. Plus formellement, un problème est représenté par un CSP, ou problème de satisfaction de contraintes, qui est le triplet formé par les variables, les domaines, et les contraintes d'un problème (voir définition 3.0.1).

Définition 3.0.1 (Problème de satisfaction de contraintes)**.** Un problème de satisfaction de contraintes, ou CSP, est un triplet $\langle X, D, C\rangle$ où X est un ensemble de variables $\{x_1, \ldots, x_n\}$, D est un ensemble de domaines $\{D_1, \ldots, D_n\}$, et C est un ensemble de contraintes $\{C_1, \ldots, C_n\}$.

Une *variable* x_i représente une inconnue du problème. Pour une variable x_i, le *domaine* D_i représente l'ensemble des valeurs qu'elle peut prendre. Le plus souvent, D_i est un ensemble fini de valeur $\{v_1, \ldots, v_k\}$ ou bien un intervalle $[\underline{v}, \overline{v}] = \{v \in \mathbb{R} \mid \underline{v} \leq v \leq \overline{v}\}$. Une *contrainte* C_i exprime une relation sur un ensemble de variables du problème qui restreint les valeurs que peuvent prendre ces variables.

Une *solution* à un CSP est une affectation de chaque variable à une valeur, ou un intervalle de valeurs, de son domaine de façon à satisfaire toutes les contraintes du problème.

Une variante du problème de satisfaction de contraintes est le problème d'optimisation sous contraintes, ou COP (voir définition 3.0.2). Dans ce cas là, il ne s'agit pas uniquement de trouver une solution satisfaisant l'ensemble des contraintes du problème mais la solution optimale : la meilleure solution par rapport au critère d'optimisation considéré.

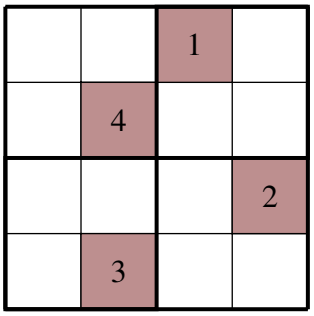
Définition 3.0.2 (Problème d'optimisation sous contraintes)**.** Un problème d'optimisation sous contraintes, ou COP, est un CSP auquel est ajouté une *fonction objectif* f. La résolution d'un tel problème consiste à maximiser, ou minimiser, la valeur de la fonction objectif f pour trouver une solution *optimale* au problème.

L'exemple 3.0.1 illustre la modélisation d'un problème de satisfaction de contraintes pour résoudre une grille de Sudoku.

Exemple 3.0.1 – Le Sudoku est un puzzle logique et combinatoire dont la version moderne a été inventée par l'américain Howard Garns en 1976. Un puzzle de Sudoku est représenté par une grille carrée de taille 9×9. Le principe du Sudoku est de remplir cette grille avec des chiffres allant de 1 à 9. Ces chiffres ne doivent jamais se répéter plus d'une fois sur une même ligne, colonne, ou bloc. La grille initiale est partiellement remplie avec des chiffres afin de limiter le nombre de solutions possibles.

Soit la grille de Sudoku 4×4 partiellement remplie de la figure 3.1. Dans ce cas, les valeurs possibles pour chaque case vont de 1 à 4.

Figure 3.1 – Grille de Sudoku 4×4



Une modélisation possible pour ce problème est d'avoir une variable représentant chaque case de la grille, comme illustré dans la figure 3.2. Ici, nous obtenons un problème avec 16 variables.

Figure 3.2 – Modélisation d'une grille de Sudoku 4×4 par un ensemble de variables

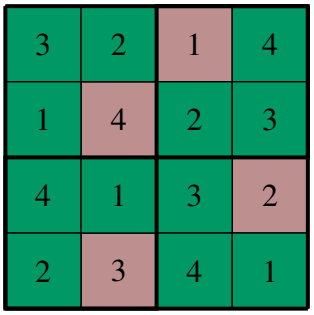
x_1	x_2	x_3	x_4
x_5	x_6	x_7	x_8
x_9	x_{10}	x_{11}	x_{12}
x_{13}	x_{14}	x_{15}	x_{16}

Pour chaque variable x_i son domaine D_i est l'ensemble $\{1, 2, 3, 4\}$. Les contraintes $x_3 = 1$, $x_6 = 4$, $x_{12} = 2$, et $x_{14} = 3$ définissent les valeurs des variables initialement connues. Ces contraintes permettent de réduire trivialement les domaines des variables de la façon suivante : $D_{x_3} = 1$, $D_{x_6} = 4$, $D_{x_{12}} = 2$, et $D_{x_{14}} = 3$.

D'après les règles du Sudoku, les chiffres sur chaque colonne, ligne, et bloc doivent être différents. Ces règles peuvent être modélisées par des contraintes d'inéquation entre les différentes variables. Ainsi, pour la variable x_1, les contraintes qui appliquent ces règles sont : $x_1 \neq x_2$, $x_1 \neq x_3$, et $x_1 \neq x_4$ pour la ligne, $x_1 \neq x_5$, $x_1 \neq x_9$, et $x_1 \neq x_{13}$ pour la colonne, et $x_1 \neq x_6$ pour le bloc. Les contraintes redondantes entre la ligne, ou la colonne, et le bloc n'ont pas besoin d'être écrites plus d'une fois. Les autres lignes, colonnes, et blocs sont traités de la même façon pour exprimer l'ensemble des contraintes exprimant les règles du Sudoku.

Une solution à ce problème est la grille de Sudoku remplie de la figure 3.3.

Figure 3.3 – Une solution de la grille de Sudoku 4×4



La résolution d'un problème repose sur des algorithmes génériques pour trouver une solution au problème. Elle alterne entre deux processus :

— la *propagation* va pour chaque contrainte appliquer un algorithme de filtrage supprimant les valeurs inconsistantes des domaines des variables puis propager aux autres contraintes ces changements

— la *recherche* va sélectionner un sous-ensemble des domaines des variables à explorer dans l'espace de recherche, via différentes heuristiques, lorsque la propagation n'est pas suffisante pour trouver une solution

3.1 Filtrage

Un algorithme de filtrage pour une contrainte donnée va retirer des domaines des variables impliquées les valeurs n'appartenant pas trivialement à une solution. Cette suppression revient à s'assurer que la contrainte est *consistante* par rapport aux domaines des variables qu'elle contient. Plusieurs niveaux de consistance existent, hiérarchisés par efficacité des suppressions de valeurs et complexité de calcul. Un filtrage est dédié à une contrainte et à la nature du domaine des variables impliquées. Dans la suite, une consistance est définie au niveau d'une contrainte. Pour une consistance quelconque A, un CSP est dit A-consistant si et seulement si chaque contrainte du CSP est A-consistante. Afin d'appliquer une consistance à l'ensemble d'un CSP, il suffit de propager les modifications des domaines d'une contrainte aux autres contraintes du problème jusqu'à obtenir la consistance au niveau du CSP. L'ordre de propagation des contraintes n'influe pas sur l'atteignabilité de la consistance au niveau du CSP [Benhamou, 1996, Apt, 1999], mais permet de réduire le temps nécessaire pour l'atteindre.

Dans le cadre des contraintes sur le discret, où les domaines sont finis, énumérables, et donc représentent un ensemble d'entiers, la consistance la plus utilisée est la *arc-consistance* (voir définition 3.1.1), introduite dans [Montanari, 1974].

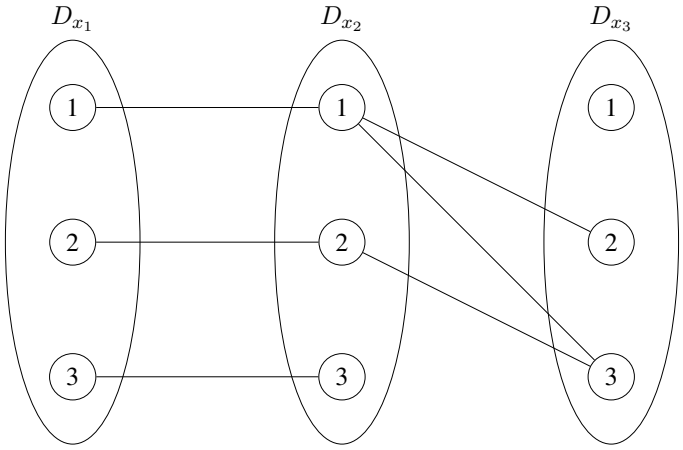
Définition 3.1.1 (Arc-consistance)**.** Une contrainte c est arc-consistante (AC) si, pour chaque variable x de c et pour chaque valeur v de son domaine D_x, il existe une affectation pour chaque autre variable de c a une valeur de son domaine satisfaisant la contrainte.

De nombreux algorithmes ont été proposés pour assurer une propagation efficace de l'arc-consistance à l'ensemble des contraintes binaires d'un CSP [Mackworth, 1977, Mohr et Hen-

derson, 1986, Bessière, 1994, Bessière et Régin, 2001]. L'exemple 3.1.1 illustre l'application de l'arc-consistance sur un réseau de contraintes avec trois variables et deux contraintes.

Exemple 3.1.1 – Soit le CSP de la figure 3.4 contenant trois variables x_1, x_2, et x_3 avec les domaines $D_{x_1} = D_{x_2} = D_{x_3} = \{1, 2, 3\}$ et deux contraintes $x_1 = x_2$ et $x_2 < x_3$.

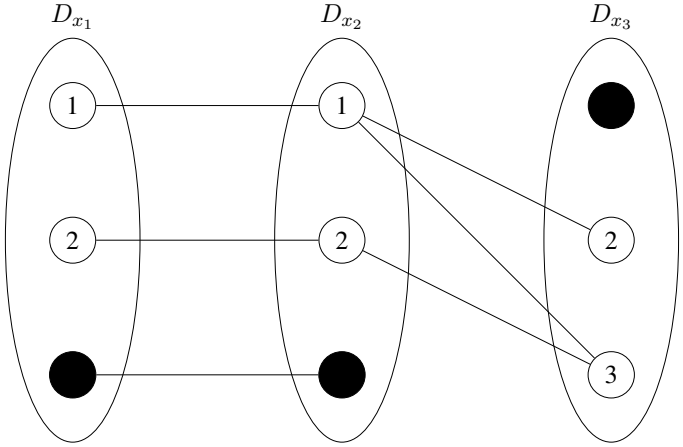
Figure 3.4 – Réseau de contraintes avant application de l'arc-consistance



Le réseau de la figure 3.4 n'est pas arc-consistant : il existe des valeurs dans les domaines qui ne sont pas consistantes avec les contraintes. Par exemple, pour la contrainte $x_2 < x_3$, 3 peut être supprimée du domaine D_{x_2} car il n'existe pas de valeur, ou support, dans le domaine de x_3 de façon à satisfaire la contrainte. De même pour 1 qui peut être supprimé de D_{x_3}. Après avoir fait ces suppressions, la propagation va supprimer 3 de D_{x_1} car la contrainte $x_1 = x_2$ n'est plus satisfaite avec cette valeur.

La figure 3.5 illustre le CSP après application de l'arc-consistance, les noeuds en noir correspondent aux valeurs supprimées des domaines des variables.

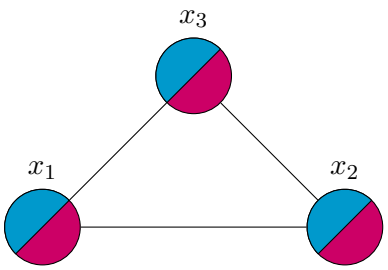
Figure 3.5 – Réseau de contraintes après application de l'arc-consistance



L'arc-consistance est une consistance *locale*. Ce n'est pas parce que toutes les contraintes d'un CSP sont arc-consistantes qu'il existe une solution au problème. L'exemple 3.1.2 illustre la limite de l'arc-consistance sur un problème de coloration de graphe.

Exemple 3.1.2 – La figure 3.6 montre un graphe avec 3 nœuds et 3 arêtes qui doit être coloré avec 2 couleurs : rose et bleu.

Figure 3.6 – Graphe triangle à colorer avec 2 couleurs



Le CSP correspondant à ce problème est défini comme suit : $X = \{x_1, x_2, x_3\}$, $D = \{D_{x_1}, D_{x_2}, D_{x_3}\}$ avec $D_{x_1} = D_{x_2} = D_{x_3} = \{\bullet, \bullet\}$, et $C = \{c_1 : x_1 \neq x_2, c_2 : x_1 \neq x_3, c_3 : x_2 \neq x_3\}$. Les trois contraintes sont arc-consistantes : pour chaque contrainte, les domaines des variables concernées satisfont la contrainte. Pourtant il n'existe aucune solution à ce problème et l'arc-consistance n'est pas suffisante pour le montrer.

L'utilisation d'une consistante plus forte permet souvent de détecter des inconsistances dans un réseau de contraintes que l'arc-consistance n'a pas pu détecter. La *chemin-consistance* (voir définition 3.1.2) introduite par [Montanari, 1974] est une consistance plus forte que l'arc-consistance.

Définition 3.1.2 (Chemin-consistance)**.** Une contrainte $c_{x,y}$ est chemin-consistante (PC) par rapport à une variable z, si pour tout couple de valeurs $\langle a,b\rangle$ satisfaisant $c_{x,y}$, avec $a \in D_x$ et $b \in D_y$, il existe au moins une valeur d du domaine D_z, telle que $\langle a,d\rangle$ satisfait la contrainte $c_{x,z}$ et $\langle b,d\rangle$ satisfait la contrainte $c_{y,z}$.

Une fois l'arc-consistance appliquée sur toutes les paires de contraintes d'un CSP, la chemin-consistance peut être utilisée pour vérifier si chaque paire de valeurs arc-consistante est solution par rapport à une troisième contrainte. Si elle n'est pas solution, la paire de valeurs est supprimée des domaines des variables. En reprenant l'exemple 3.1.2, où l'arc-consistance ne supprime aucune valeur des domaines, alors qu'il n'existe pas de solution au problème, la chemin-consistance permet de supprimer toutes les valeurs des domaines et d'affirmer qu'il n'existe aucune solution à ce problème.

Dans le cas où les domaines des variables sont très grands, et representés par des intervalles d'entiers, la *bound-consistance* (voir définition 3.1.3), introduite dans [Hentenryck *et al.*, 1994], peut être utilisée pour ne considérer que les bornes des domaines. Cette consistance est plus faible que l'arc-consistance, mais réduit le temps de calcul de la consistance car elle ne considère pas toutes les valeurs d'un domaine.

Définition 3.1.3 (Bound-consistance)**.** Une contrainte c est bound-consistante (BC) si, pour chaque variables x de la contrainte et pour chaque valeur v aux bornes de son domaine tel que $v \in \{\min(D_x), \max(D_x)\}$, il existe une affectation couplant chaque autre variable de c à une valeur aux bornes de son domaine satisfaisant la contrainte.

Ces consistances ont un raisonnement local, qui ne permet pas de toujours trouver une solution au problème. Afin de mieux détecter certaines inconsistances des contraintes globales sont définies. Ces contraintes globales capturent un sous-problème du problème initial et appliquent des algorithmes plus puissant que dans un filtrage local. Une contrainte globale exprime souvent, mais pas toujours, un ensemble de contraintes élémentaires. Elle apporte également une meilleure vue de la structure du problème à résoudre. Il existe de nombreuses contraintes globales : *all-different* [Régin, 1994] représente une conjonction de contraintes d'inégalités, *sum* [Yunes, 2002] force la somme des valeurs des variables à être égale à une valeur s, ou encore *cumulative* [Aggoun et Beldiceanu, 1992] modélise des problèmes d'ordonnancement.

Les contraintes sur le continu ne peuvent pas appliquer les consistances les plus courantes définies pour le discret, comme l'arc-consistance ou la chemin-consistance, pour réduire les domaines des variables. En effet, les domaines des variables continues sont sur $\mathbb{R}$ et contiennent une infinité de valeurs. Il est donc impossible d'utiliser des techniques basées sur l'énumération de valeurs. En pratique, le domaine d'une variable continue est représenté par un intervalle à bornes dans $\mathbb{F}$, les réels n'étant pas représentables en machine. Différents niveaux de consistances ont donc été proposés en utilisant les particularités de ces domaines. Les consistances sur le continu utilisent le plus souvent l'arithmétique des intervalles [Moore *et al.*, 2009].

La *2B-consistance* (voir définition 3.1.4) est une relaxation de l'arc-consistance aux bornes de l'intervalle introduite par [Lhomme, 1993]. Elle est aussi connue sous le nom de *hull-consistance* [Benhamou et Older, 1997]. Cette consistance s'intéresse aux bornes d'un domaine afin de calculer le plus petit intervalle possible qui soit consistant.

Définition 3.1.4 (2B-consistance)**.** Une contrainte c est 2B-consistante (2B) si, pour chaque variable x de la contrainte avec un domaine $D_x = [a,b]$, lorsque x est réduit à a et à b il existe des valeurs dans les domaines de toutes les autres variables de c satisfaisant la contrainte.

La 2B-consistance repose sur des fonctions de projections. Dans le cas général, calculer ces fonctions pour la contrainte initiale est impossible [Collavizza *et al.*, 1998]. Les contraintes d'un problème sont donc décomposées en contraintes binaires et ternaires afin de pouvoir calculer les fonctions de projections. L'exemple 3.1.3 illustre la 2B-consistance sur un problème avec deux variables et une contrainte.

Exemple 3.1.3 – Soit le CSP suivant : $X = \{x_0, x_1\}$, $D = \{D_0, D_1\}$ avec $D_0 = [1, 4]$ et $D_1 = [-2, 2]$, et $C = \{c_1 : x_0 = {x_1}^2\}$. La contrainte c_1 est 2B-consistante : en prenant les valeurs aux bornes, la contrainte est satisfaite. Par contre, c_1 n'est pas arc-consistante, car la valeur 0 dans D_1 n'a pas de valeur équivalente dans D_0 pour satisfaire la contrainte.

La réduction des bornes d'un domaine, même d'un seul nombre à virgule flottante, via une 2B-consistance, déclenche la propagation des modifications aux autres contraintes. Cette propagation peut être coûteuse en temps et même conduire à une convergence lente lors de la résolution. Afin d'éviter ce phénomène, la *2B(w)-consistance* est généralement utilisée. Le paramètre w fixe la taille minimale qu'une réduction de domaine doit avoir pour être prise en compte : il s'exprime le plus souvent en pourcentage de la taille du domaine. À noter qu'il n'y a pas de rapport direct entre la valeur de w et l'efficacité du filtrage.

La 2B-consistance étant une relaxation de l'arc-consistance, elle a également un raisonnement local qui ne permet pas, en général, de supprimer toutes les inconsistances au niveau d'un réseau de contraintes. Des consistantes plus fortes sont donc définies sur le continu pour supprimer plus de valeurs inconsistantes dans les domaines des variables. La *3B-consistance* (voir définition 3.1.5)

est une généralisation de la 2B-consistance [Lhomme, 1993]. Elle consiste à s'assurer qu'un système de contrainte ne devient pas inconsistant lorsque le domaine d'une variable est réduit à l'une de ces bornes. En d'autres mots, la 3B-consistance applique une 2B-consistance à chaque borne des domaines des variables du problème.

Définition 3.1.5 (3B-consistance)**.** Une contrainte est 3B-consistante (3B) si, pour chaque variable x de la contrainte avec un domaine $D_x = [a, b]$, lorsque son domaine est réduit à a et à b le CSP est 2B-consistant.

La *box-consistance* (voir définition 3.1.6), introduite dans [Benhamou *et al.*, 1994] est une autre approximation de l'arc-consistance. Contrairement à la 2B-consistance, elle ne nécessite pas une décomposition des contraintes en contraintes élémentaires. Une box-consistance revient à remplacer toutes les variables d'une contrainte, excepté une, par l'intervalle représentant son domaine. Cela génère un système de fonctions univariables pouvant être résolu par la méthode de Newton [Moore *et al.*, 2009].

Définition 3.1.6 (Box-consistance)**.** Une contrainte c, sur un ensemble de variables $\{x_1, \ldots, x_k\}$, est box-consistante (BoxC) si, pour chaque variable x_i dans $\{x_1, \ldots, x_k\}$ tel que $D_{x_i} = [a, b]$ les relations suivantes sont satisfaites.

$$\begin{aligned} &\mathbf{c}(D_{x_1}, \ldots, D_{x_{i-1}}, [a, a^+[, D_{x_{i+1}}, \ldots, D_{x_k}) \\ &\mathbf{c}(D_{x_1}, \ldots, D_{x_{i-1}},]b^-, b], D_{x_{i+1}}, \ldots, D_{x_k}) \end{aligned} \tag{3.1}$$

où $\mathbf{c}$ est l'extension aux intervalles de la contrainte c.

La bornes-consistance (BC) [Hentenryck *et al.*, 1994] (voir définition 3.1.3) s'utilise également sur le continu. La bornes-consistance applique le principe de la 3B-consistance à la box-consistance : elle s'assure que la contrainte est box-consistante lorsque le domaine d'une variable est réduit à l'une de ces bornes. Une analyse des relations entre les consistances sur le continu est proposée dans [Collavizza *et al.*, 1998]. En particulier, les limites du raisonnement local de la 2B-consistance et de la box-consistance sont mises en avant : la décomposition des contraintes en 2B supprime les relations entre les occurrences multiples d'une variable ; le système de fonctions univariables de la BC traite le problème d'occurrences multiples pour une seule variable mais peut ne pas le supporter pour plusieurs variables. Les consistances plus fortes, *e.g.,* 3B et BC, atténuent ces limites locales des consistances plus faibles. Une hiérarchie des consistances sur le continu est également proposée : $BC \preceq 3B \preceq BoxC \preceq 2B$, avec $A \preceq B$ une relation où un filtrage par *A-consistance* produit des domaines plus réduits qu'un filtrage par *B-consistance* sur le même CSP (avec décomposition des contraintes pour la $3B$ et la $2B$).

De la même façon que sur le discret, des contraintes globales sont définies pour le continu, *e.g., quad* [Lebbah *et al.*, 2002] est une contrainte globale pour représenter et résoudre un système de contraintes quadratiques.

3.2 Recherche

En général, un filtrage seul n'est pas suffisant pour obtenir une solution à un problème. C'est pourquoi, une fois que la propagation à toutes les contraintes ne produit plus de réduction, il est nécessaire d'explorer l'espace de recherche de manière efficace. Cette recherche relance ensuite une nouvelle propagation afin d'obtenir de nouvelles réductions de domaines. Il existe de nombreuses

stratégies et heuristiques pour guider cette recherche de manière efficace. En général, la recherche est vue de manière arborescente avec un parcours en profondeur. À chaque nœud de l'arbre une stratégie est employée pour sélectionner une variable et réduire son domaine. Cette réduction revient souvent à réduire le domaine à une seule valeur ou à un sous-ensemble du domaine. Lorsque la recherche mène à une inconsistance, un *retour en arrière*, ou *backtrack*, est effectué pour annuler le choix et explorer d'autres sous problèmes. D'autres concepts sont utilisés pour améliorer l'exploration de l'espace de recherche, comme le *retour en arrière non chronologique* [Stallman et Sussman, 1977], ou *backjumping*, pour remonter plus haut que le nœud parent direct, le *redémarrage* [Harvey, 1995], ou *restart*, pour redémarrer la recherche à l'état initial et éviter de refaire les mêmes mauvais choix, ou bien encore l'exploration parallèle [Perron, 1999, Régin *et al.*, 2013, Bergman *et al.*, 2014] de l'espace de recherche. Les stratégies appliquées à chaque nœud lors de la recherche tirent avantage de la nature des domaines des variables, de la structure du problème, et d'autres mesures sur les CSP.

De nombreux travaux ont proposé des stratégies sur le discret, qui suivent souvent le principe du *fail-first*, introduit dans [Haralick et Elliott, 1979] : l'idée est d'explorer en premier des domaines avec de plus grandes chances d'échouer pour effectuer, le plus tôt possible, de plus grandes coupes dans l'arbre de recherche. D'autres stratégies très utilisées sont :

— **lex**, qui calcule un ordre lexicographique sur les variables
— **dom** [Golomb et Baumert, 1965, Haralick et Elliott, 1979], pour sélectionner la variable avec le plus petit domaine
— **dom+deg** [Brélaz, 1979], qui sélectionne la variable ayant le plus petit domaine et impliquée dans le plus de contraintes

La stratégie **lex** est statique : l'ordre est décidé une seule fois et ne change plus lors de la résolution, tandis que **dom** et **dom+deg** sont des ordres dynamiques impactés par les réductions de domaines effectuées lors de la propagation de contraintes. D'autres généralisations de **dom** existent, comme **dom/deg** [Bessière et Régin, 1996] ou **dom/wdeg** [Boussemart *et al.*, 2004] avec w un poids associé à chaque contrainte. Après sélection d'une variable, son domaine est réduit afin de contraindre l'espace de recherche et d'en explorer une sous-partie. Contrairement aux stratégies de choix de variables, les stratégies de choix de valeurs suivent souvent l'idée du *succeed-first* : choisir une valeur qui a de grandes chances d'appartenir à une solution. Souvent une stratégie **min-value**, ou **max-value**, est employée pour sélectionner la plus petite, ou la plus grande, valeur du domaine de la variable.

D'autres stratégies ont été proposées pour les variables sur le continu. En général les stratégies de choix de variable utilisent **lex** comme sur le discret, ou bien **round-robin** pour changer de variable à chaque choix et s'assurer que tous les domaines sont réduits lors de la recherche. Une autre stratégie, **largest-firt** [Ratz, 1994], choisit la variable avec le plus grand domaine pour obtenir rapidement des intervalles plus petits. Contrairement aux variables discrètes, les domaines sur le continu sont des intervalles contenant une infinité de valeurs. La sélection de valeur revient donc à choisir un intervalle plus petit que l'intervalle initial. Le plus souvent, une **bissection** est appliquée pour couper le domaine en deux : l'intervalle initial $[a, b]$ est découpé en deux intervalles plus petits $[a, \frac{a+b}{2}]$ et $]\frac{a+b}{2}, b]$.

3.3 Contraintes sur les nombres à virgule flottante

Les contraintes sur les nombres à virgule flottante, introduite dans [Michel *et al.*, 2001], ont pour application la vérification de programmes, en particulier pour la génération de cas de test ou la correction de programme. Même si les nombres à virgule flottante sont un sous-ensemble [1] fini et discret des nombres réels (voir chapitre 2), les contraintes sur le continu ne sont pas adaptées pour capturer les particularités de l'arithmétique des nombres à virgule flottante. La grande taille d'un domaine sur les nombres à virgule flottante ne permet pas non plus de profiter des contraintes sur le discret, qui réalisent souvent une énumération de valeurs. L'exemple 3.3.1 illustre la différence entre une résolution d'équations sur $\mathbb{R}$ et sur $\mathbb{F}$.

Exemple 3.3.1 – Soit les systèmes d'équations 3.2 et 3.3 et le mode d'arrondi *au plus proche pair* pour les nombres à virgule flottante.

$$\begin{gathered} x + 1 \times 10^8 = 1 \times 10^8 \\ x \neq 0 \end{gathered} \tag{3.2}$$

$$x^2 = 2 \tag{3.3}$$

L'équation 3.2 n'a pas de solutions sur $\mathbb{R}$ alors qu'il existe de nombreuses solutions sur $\mathbb{F}$, à cause du phénomène d'absorption (voir l'exemple 2.3.3 du chapitre 2). Au contraire, l'équation 3.3 a $\sqrt{2}$ et $-\sqrt{2}$ comme solutions sur $\mathbb{R}$ alors qu'il n'en existe pas sur $\mathbb{F}$.

Les nombres à virgule flottante n'ayant pas les mêmes propriétés que les nombres réels, des algorithmes de filtrage dédiés sont nécessaires pour réduire correctement les domaines des variables flottantes. La programmation par contraintes pour les nombres à virgule flottante s'inspire des contraintes sur le discret et des contraintes sur le continu pour ses algorithmes de filtrage et ses stratégies de recherche. Un problème de satisfaction de contraintes sur les flottants est similaire à un CSP classique (voir définition 3.0.1), à l'exception de D où un domaine est un ensemble fini de nombres à virgule flottante représenté par un intervalle à bornes dans $\mathbb{F}$.

3.3.1 Filtrage des domaines sur les nombres à virgule flottante

De façon similaire aux domaines sur le continu, les domaines sur les nombres flottants sont trop grands pour appliquer une arc-consistance sur l'ensemble de leurs valeurs. C'est pourquoi les filtrages sur les domaines flottants [Michel *et al.*, 2001, Michel, 2002, Botella *et al.*, 2006, Marre et Michel, 2010] s'inspirent des filtrages sur le continu, et en particulier de la box-consistance et de la 2B-consistance.

La *box-consistance* (voir définition 3.1.6) a été adaptée aux domaines sur les nombres à virgule flottante dans [Michel *et al.*, 2001]. Cette consistance s'intéresse aux bornes des domaines afin de les réfuter pour réduire les domaines. L'algorithme de filtrage de [Michel *et al.*, 2001] adapte **Netwon**, un *branch-and-prune* proposé dans [Van Hentenryck *et al.*, 1997]. Ce filtrage utilise l'arithmétique des intervalles sur les nombres à virgule flottante pour réduire un intervalle $[a, b]$ à $[a_m, b]$ en évaluant l'intervalle $[a, a_m[$ pour s'assurer qu'il ne contient pas de solution. La

1. Ce sous-ensemble est augmenté des valeurs spéciales, *i.e.,* NaN, $-\infty$, $+\infty$, -0, $+0$, ...

box-consistance sur les nombres flottants est conservative des solutions : les intervalles calculés contiennent toutes les solutions du problème. Mais, l'arrondi extérieur nécessaire à l'arithmétique des intervalles produit en général des bornes qui ne sont pas solution. La réfutation des bornes par box-consistance est également coûteuse en temps de calcul.

La *2B-consistance* (voir définition 3.1.4) a également été adaptée aux problèmes sur les flottants dans [Michel, 2002, Botella *et al.*, 2006, Marre et Michel, 2010]. Cette consistance réalise une arc-consistance aux bornes des domaines des variables d'une contrainte. De la même façon que sur le continu, cette consistance utilise des *fonctions de projections* pour réduire les domaines. Elle nécessite également une décomposition des contraintes en contraintes élémentaires, afin de pouvoir calculer ces fonctions. Pour une contrainte $z = x \odot y$, il existe deux types de fonctions de projections : *directe* afin de réduire le domaine de la variable résultat, ici z ; *inverse* pour calculer les réductions de domaines pour les autres variables de la contrainte, ici x et y. L'exemple 3.3.2 illustre ces fonctions de projections sur une contrainte ternaire.

Exemple 3.3.2 – Soit la contrainte suivante, avec un mode d'arrondi *au plus proche pair*.

$$z = x \oplus y \tag{3.4}$$

où x, y, z sont des variables avec des domaines dans $\mathbb{F}$.
La fonction de projection directe pour z est f_1.

$$f_1 : D_z \leftarrow D_z \cap [\underline{x} \oplus \underline{y}, \overline{x} \oplus \overline{y}] \tag{3.5}$$

Les fonctions de projections inverses pour x et y sont respectivement les fonctions f_2 et f_3, où $mid(a,a^+)$ calcule le nombre au milieu de a et a^+ et $\oplus_{+\infty}$ (resp. $\oplus_{-\infty}$) est l'opération $\oplus$ avec un arrondi vers $+\infty$ (resp. $-\infty$). Afin de représenter correctement le milieu, $mid(a,a^+)$, il est nécessaire d'ajouter un bit supplémentaire dans la mantisse.

$$f_2 : D_x \leftarrow D_x \cap [mid(\underline{z}^-,\underline{z}) \ominus_{+\infty} \overline{y}, mid(\overline{z},\overline{z}^+) \ominus_{-\infty} \underline{y}] \tag{3.6}$$

$$f_3 : D_y \leftarrow D_y \cap [mid(\underline{z}^-,\underline{z}) \ominus_{+\infty} \underline{x}, mid(\overline{z},\overline{z}^+) \ominus_{-\infty} \underline{x}] \tag{3.7}$$

Une définition plus précise des fonctions de projections est disponible dans [Michel, 2002, Botella *et al.*, 2006, Marre et Michel, 2010].

D'autres travaux se sont intéressés aux fonctions de projections pour réduire les domaines des variables dans des contraintes arithmétiques, comme [Gallois-Wong *et al.*, 2020] qui propose une projection inverse optimale pour l'addition, ou [Andrlon *et al.*, 2019] qui généralise des résultats de [Marre et Michel, 2010] et propose des fonctions de projections plus précises.

En pratique, la *2B(w)-consistance* est préférée à la 2B-consistance afin d'éviter une convergence lente. La répartition des nombres à virgule flottante étant non uniforme [2], w est exprimé comme un pourcentage de la taille du domaine de la variable. De la même manière que sur le continu, les consistances plus fortes comme la *3B-consistance* peuvent être facilement adaptées aux contraintes sur les flottants.

2. Près de la moitié des nombres à virgule flottante se trouve dans l'intervalle $[-1, 1]$.

3.3.2 Stratégies de recherche dédiées aux nombres à virgule flottante

Les stratégies définies sur le continu, comme **lex** et **bissection**, peuvent être directement appliquées sur les nombres flottants. Les nombres à virgule flottante n'ayant pas les mêmes propriétés que les nombres réels (voir chapitre 2), les stratégies sur le continu ne prennent pas en compte ces particularités et perdent donc en efficacité. Un des premiers travaux à proposer une stratégie dédiée aux nombres flottants est [Collavizza *et al.*, 2016]. La stratégie **fpc** combine l'énumération et la bissection pour guider l'exploration dans l'espace de recherche : un domaine $[a, b]$ dans $\mathbb{F}$ est découpé comme suit, a, b et le milieu de l'intervalle m sont énumérés, tandis que les deux intervalles $]a, m[$ et $]m, b[$ sont générés.

De façon plus générale, la thèse d'Heytem Zitoun [Zitoun, 2018] introduit des stratégies de recherche pour les variables et pour les valeurs en prenant avantage des nombres à virgule flottante. Certaines de ces stratégies se basent sur la structure du domaine des variables, et maximise ou minimise un critère donné :

— **dom** est similaire à la stratégie du même nom dans le cas discret [Golomb et Baumert, 1965] et s'intéresse à la taille des domaines
— **card** calcule le nombre de valeurs dans un domaine
— **dens** part de l'intuition que si un domaine est dense, le nombre potentiel de solutions qu'il contient est plus grand
— **magn** sélectionne les variables ayant un domaine contenant de grandes valeurs
— **deg** est une adaptation directe de **dom/wdeg** [Boussemart *et al.*, 2004] venant du discret
— **occ** cherche les occurrences multiples des variables au sein d'une contrainte.

D'autres stratégies se basent sur les contraintes pour sélectionner une variable, toujours en maximisant ou minimisant un critère donné : **abs** est une stratégie qui s'intéresse au phénomène d'absorption sur les nombres à virgule flottante (voir l'exemple 2.3.3 du chapitre 2), tandis que **canc** calcule le taux de cancellation (voir l'exemple 2.3.4 du chapitre 2) de chaque variable. Des combinaisons de plusieurs stratégies sont également proposées, comme une stratégie associant **abs** et **dens**.

Comme sur le discret ou le continu, une fois qu'une variable est sélectionnée, son domaine est réduit afin de poursuivre la recherche. Les stratégies de sélection de valeurs sont inspirées de celle sur le continu : la grande cardinalité des domaines ne permettant pas une énumération de toutes les valeurs. Une généralisation de **fpc** est proposée dans [Zitoun, 2018] et déclinée en plusieurs stratégies :

— **3Split** énumère uniquement le milieu
— **6Split** énumère les valeurs aux bornes et deux valeurs au milieu du domaine

Ces deux stratégies génèrent aussi deux intervalles entre les bornes et le milieu du domaine. D'autres généralisations sont proposées et détaillées dans [Zitoun, 2018], comme **Enum-N** ou **Delta-N**. Une autre stratégie, **splitAbs**, reprend le concept de l'absorption sur les nombres à virgule flottante et sélectionne les valeurs les plus à même de provoquer une absorption. La stratégie **3BSplit** s'inspire d'une consistance plus forte : la *3B-consistance* (voir définition 3.1.5). Elle découpe dynamiquement le domaine de la variable en fonction du comportement de la recherche associée à une 3B-consistance. L'objectif est de chercher des solutions dans des petits intervalles aux bornes du domaine de la variable ou de réfuter ces petits intervalles.

3.4 Conclusion

La programmation par contraintes permet de résoudre efficacement des problèmes combinatoires difficiles. Elle offre une séparation claire entre la modélisation du problème et sa résolution. La modélisation propose une représentation mathématique du problème sous forme de contraintes, tandis que la résolution utilise deux mécanismes : la propagation et la recherche, pour trouver une solution au problème. Les techniques utilisées pour la résolution dépendent grandement de la nature des variables du problème. De nombreux travaux ont proposé des algorithmes de filtrage et des stratégies de recherche efficaces pour la résolution de problèmes discret ou continu. La programmation par contraintes a été étendue aux nombres à virgule flottante pour l'adapter à la vérification de programme, notamment pour la génération de cas de test ou la correction de programme. Néanmoins, les contraintes sur les nombres à virgule flottante ne sont pas encore capables de traiter l'analyse d'erreurs, *i.e.,* calculer ou approximer les erreurs de calcul inhérentes aux opérations sur les nombres à virgule flottante.

CHAPITRE 4

Analyse de programmes

Dans ce chapitre nous présentons les différentes méthodes et techniques en analyse de programmes dédiées à l'analyse d'erreurs issues des calculs sur les nombres à virgule flottante. Ces méthodes sont séparées en deux catégories : l'analyse statique et l'analyse dynamique de programmes.

L'analyse de programmes est un ensemble de techniques et de méthodes qui cherche à garantir le bon fonctionnement des programmes informatiques. En général, elle se sépare en deux catégories : l'analyse dynamique et l'analyse statique. L'analyse dynamique, plus connue sous le nom de *test* [Myers, 2004], ou *testing*, montre la présence ou l'absence de bugs sur une exécution du programme. Pour cela, le test repose sur la simulation ou l'instrumentation du programme associé à un ensemble de valeurs d'entrée à exécuter. De ce fait, un ensemble de tests ne considère qu'une partie des traces d'exécution d'un programme. Il assure qu'une exécution du programme se produit avec ou sans bug. Le test est souvent utilisé pour assurer la couverture [1] d'un programme, *i.e.,* l'ensemble des branches d'un programme sont exécutables, et ainsi identifier la présence de code *mort* : du code qui n'ai jamais exécuté. Mais n'étant pas une approche complète, il ne peut pas garantir la correction du programme. L'analyse statique assure la correction d'un programme sans nécessiter son exécution. Elle raisonne sur l'ensemble des comportements que le programme peut prendre lors de son exécution et en déduit des règles logiques qui permettent de prouver la correction du programme. Comme l'analyse de programmes est un problème indécidable [Rice, 1953], ces méthodes raisonnent le plus souvent sur des sur-approximations du comportement d'un programme. Les résultats peuvent être moins précis et parfois générer un *faux positif* : la détection d'une potentielle erreur dans la sur-approximation qui n'existe pas en pratique dans le comportement du programme. Toutefois, prouver la correction de la sur-approximation est suffisant pour assurer la correction du programme. Sachant qu'une sur-approximation capture à minima le comportement en pratique du programme, si la sur-approximation est correcte, alors le programme est toujours correct, mais comme dit précédemment l'inverse n'est pas toujours vrai.

Le calcul d'erreurs inhérentes aux opérations sur les nombres à virgule flottante a été largement étudié en analyse statique de programme [Goubault et Putot, 2006, Daumas et Melquiond, 2010, Darulova et Kuncak, 2014, Solovyev *et al.*, 2015, Moscato *et al.*, 2017, Magron *et al.*, 2017, Darulova *et al.*, 2018b, Magron, 2018]. Ces techniques produisent, le plus souvent, une sur-approximation des erreurs qui peuvent apparaître dans un programme. Ces méthodes se basent le plus souvent sur une abstraction des erreurs, à base d'intervalles ou de formes affines, ou une expression de l'erreur sous forme de problème d'optimisation globale. D'autres méthodes s'intéressent à la détection de *grandes* erreurs dans un programme [Chiang *et al.*, 2014, Zou *et al.*, 2020, Xia *et al.*, 2020]. Pour cela, la technique privilégiée est le *test aléatoire*. Il consiste à générer des valeurs d'entrées aléatoires pour le programme afin de calculer l'erreur produite lors de son exécution. L'objectif est de trouver des valeurs d'entrées pour lesquelles le programme produit une erreur importante : une erreur qui augmente la différence entre le résultat sur $\mathbb{R}$ et $\mathbb{F}$. La génération aléatoire est le plus souvent guidée par des stratégies d'exploration de l'espace de recherche, afin d'améliorer l'efficacité du processus.

Dans ce chapitre, nous présentons les principales approches, en analyse statique et dynamique, approximant les erreurs générées par des programmes ou détectant des cas d'exécution produisant de grandes erreurs.

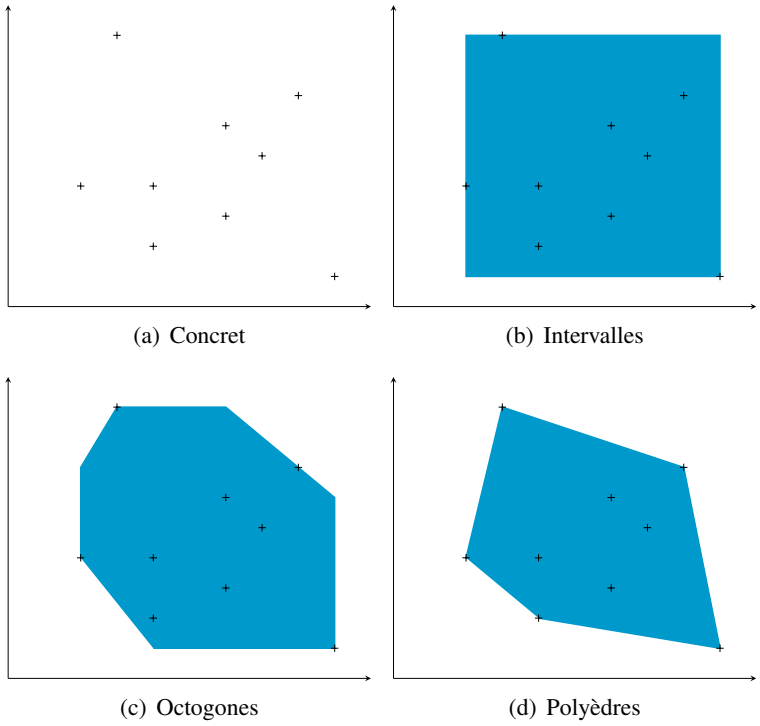
4.1 Interprétation abstraite

L'interprétation abstraite [Cousot et Cousot, 1977b, Cousot et Cousot, 1977a] est une théorie d'approximation et de comparaison des sémantiques. La *sémantique* d'un programme est l'ensemble des valeurs prises par les variables d'un programme, tandis que la *spécification* est l'en-

1. La couverture est une mesure utilisée afin de quantifier le taux de code source exécuté par l'application des tests.

semble des règles et comportements que le programme doit respecter. Si la sémantique d'un programme respecte l'ensemble des spécifications, le programme est dit correct. L'interprétation abstraite fait la différence entre plusieurs niveaux de sémantiques : la sémantique *concrète* représente l'ensemble des valeurs que le programme prend réellement, tandis qu'une sémantique *abstraite* est une sur-approximation de la sémantique concrète. En général, calculer la sémantique concrète, d'un programme est très coûteux voire impossible [Rice, 1953]. C'est pourquoi l'interprétation abstraite utilise des domaines abstraits, pour permettre l'analyse d'un programme. Il existe différents domaines abstraits, dont les principaux sont les intervalles [Cousot et Cousot, 1977b], les polyèdres [Cousot et Halbwachs, 1978], et les octogones [Miné, 2006]. Des domaines adaptés aux nombres à virgule flottante [Miné, 2004, Chen *et al.*, 2008] ont également été mis au point. Ils ont été intégrés dans un analyseur statique, Astrée [Cousot *et al.*, 2006], capable de détecter des exceptions à l'exécution d'un programme, comme des dépassements, ou *overflow*, de nombres à virgule flottante. L'exemple 4.1.1 illustre l'abstraction d'un domaine concret en différents domaines abstraits.

Exemple 4.1.1 – Dans la figure 4.1 le nuage de points représente un domaine concret 4.1(a) qui est abstrait, respectivement, vers un domaine des intervalles 4.1(b), vers un domaine des polyèdres 4.1(c), et vers un domaine des octogones 4.1(d). Les domaines abstraits sont donnés dans l'ordre de précision par rapport au domaine concret.



(a) Concret (b) Intervalles

(c) Octogones (d) Polyèdres

Figure 4.1 – Abstraction d'un domaine concret en différents domaines abstraits

Par construction, un domaine abstrait est correct : toute propriété satisfaite dans le domaine abstrait est satisfaite dans le domaine concret. Cela assure que la correction de l'abstraction d'un programme implique la correction réelle du programme. L'inverse n'est par contre pas toujours

vrai. Une propriété peut ne pas être satisfaite dans le domaine abstrait d'un programme : elle ne respecte pas la spécification, tout en étant vraie dans son domaine concret. Cette notion, appelée *faux positif*, signifie qu'une analyse par interprétation abstraite peut détecter une erreur dans un programme qui ne se produit jamais en pratique. L'exemple 4.1.2 illustre la notion de faux positif par rapport aux différentes sémantiques d'un programme.

Exemple 4.1.2 – La figure 4.2 montre un faux positif. La forme verte P est le domaine concret d'un programme : il représente l'ensemble des valeurs prises par les variables du programme. La sphère S est la spécification du programme, *i.e.,* l'ensemble des règles et comportements que le programme doit respecter pour fonctionner correctement. Ici, l'interprétation abstraite cherche à montrer que P est strictement inclus dans S : il n'existe aucune valeurs pouvant être prise par les variables du programme de façon à violer sa spécification.

Le domaine concret étant très coûteux à calculer, il est sur-approximé par le rectangle bleu I représentant le domaine abstrait des intervalles.

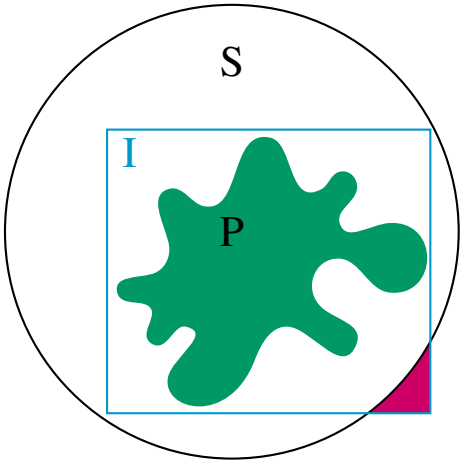


Figure 4.2 – Faux positif

La zone rose dans I qui se trouve en dehors de S est un faux positif : cette partie du domaine abstrait ne satisfait pas la spécification, alors que le domaine concret la satisfait en tout point.

Plusieurs travaux ont adapté l'interprétation abstraite à l'analyse d'erreurs de calcul sur les nombres à virgule flottante. En particulier, Fluctuat [Goubault et Putot, 2006, Goubault et Putot, 2011] et PRECiSA [Moscato *et al.*, 2017, Titolo *et al.*, 2018] sont deux analyseurs statiques qui calculent des sur-approximations des erreurs.

Fluctuat [Goubault et Putot, 2006, Goubault et Putot, 2011] est un analyseur statique qui utilise le domaine abstrait des zonotopes [Ghorbal *et al.*, 2009, Ghorbal *et al.*, 2010] et l'arithmétique affine [de Figueiredo et Stolfi, 2004] pour produire une sur-approximation des erreurs de calcul. Fluctuat utilise le modèle d'arrondi à base d'ulp (voir définition 2.3.4). L'idée de la sémantique concrète de Fluctuat est de décrire la différence entre l'évaluation exacte du programme sur les réels et son exécution machine sur les nombres à virgule flottante. Pour cela, une variable est décrite par le triplet (f^x, r^x, e^x), où $f^x \in \mathbb{F}$ est sa valeur sur les nombres à virgule flottante, $r^x \in \mathbb{R}$ est sa valeur sur les nombres réels, et $e^x = r^x - f^x$ est son erreur d'arrondi introduite en passant

de $\mathbb{R}$ à $\mathbb{F}$. Une abstraction de la sémantique concrète proposée par Fluctuat utilise le domaine des zonotopes : un domaine abstrait relationnel qui conserve certaines relations entre variables et erreurs à travers le programme. Comme les propriétés algébriques des nombres réels ne sont pas conservées sur les nombres à virgule flottante, un tel domaine commence par borner r^x et e^x puis infère f^x à partir de la différence $r^x - e^x$. Pour abstraire le triplet (f^x, r^x, e^x), les zonotopes utilisent l'arithmétique affine : une extension plus précise de l'arithmétique des intervalles prenant en compte des corrélations linéaires entre les variables. L'exemple 4.1.3 illustre l'arithmétique affine sur une variable x quelconque.

Exemple 4.1.3 – Soit x une variable et $\hat{x}$ sa forme affine. Une forme affine est une somme formelle comme montré dans l'équation 4.1.

$$\hat{x} = \alpha_0^x + \sum_{i=1}^{n} \alpha_i^x \varepsilon_i \tag{4.1}$$

Les symboles de bruits ε_i sont des variables symboliques indépendantes avec une valeur inconnue dans l'intervalle $[-1, 1]$. Les coefficients $\alpha_i^x \in \mathbb{R}$ sont des dérivées partielles du centre $\alpha_0^x \in \mathbb{R}$ de la forme affine. Elles peuvent exprimer des incertitudes sur les valeurs de la variable ainsi que des incertitudes issues des calculs. Le partage de symboles de bruits entre différentes variables exprime des dépendances implicites.

L'intervalle des valeurs pouvant être prises par une variable x, défini par une forme affine $\hat{x}$, est donné dans l'équation 4.2.

$$\gamma(\hat{x}) = \left[\alpha_0^x - \sum_{i=1}^{n} |\alpha_i^x|,\ \alpha_0^x + \sum_{i=1}^{n} |\alpha_i^x|\right] \tag{4.2}$$

γ est une fonction convertissant une forme affine en intervalle.

En utilisant l'arithmétique affine, r^x et e^x sont abstraits avec deux symboles de bruits : ε_i^r et ε_i^e représentant respectivement l'incertitude sur la valeur réelle et l'incertitude sur l'erreur. Ainsi la valeur abstraite d'une variable est composée d'un intervalle et de deux formes affines, tel que $x = (\mathbf{f}^x, \hat{r}^x, \hat{e}^x)$ où $\mathbf{f}^x$ est l'intervalle des valeurs dans $\mathbb{F}$ que peut prendre la variable x. L'équation 4.3 donne les valeurs de $\hat{r}^x$ et de $\hat{e}^x$.

$$\hat{r}^x = r_0^x + \sum_i r_i^x \varepsilon_i^r \tag{4.3}$$

$$\hat{e}^x = e_0^x + \sum_i e_i^x \varepsilon_i^r + \sum_l e_l^x \varepsilon_l \tag{4.4}$$

Dans l'expression de l'erreur $\hat{e}^x$, $e_i^x \varepsilon_i^r$ exprime la propagation de l'incertitude de la valeur au point i, tandis que $e_l^x \varepsilon_l$ exprime l'incertitude de l'erreur d'arrondi au point l. Ces deux incertitudes modélisent la dépendance entre erreurs et valeurs. Fluctuat applique des *fonctions de transfert* afin de calculer les valeurs et erreurs des variables pour chaque opération du programme. L'équation 4.5 donne les fonctions de transfert de $\hat{r}^x$, $\hat{e}^x$, et $\mathbf{f}^x$ pour une opération $z = x \diamond y$ où $\diamond$ est une addition ou une soustraction.

$$\hat{r}^z = \hat{r}^x \diamond \hat{r}^y \tag{4.5}$$
$$\hat{e}^z = \hat{e}^x \diamond \hat{e}^y + new_{\varepsilon^e}(\mathbf{e}(\gamma(\hat{r}^z - \hat{e}^x \diamond \hat{e}^y))) \tag{4.6}$$
$$\mathbf{f}^z = (\mathbf{f}^x \diamond \mathbf{f}^y) \cap (\hat{r}^z - \hat{e}^z) \tag{4.7}$$

La fonction new_{ε^e} crée un nouveau symbole de bruit pour l'erreur et la fonction $\mathbf{e}$ calcule un intervalle pour l'erreur d'arrondi. L'expression de ces deux fonctions est donnée dans [Goubault et Putot, 2011].

Un des mécanismes de Fluctuat permettant d'obtenir des bornes plus précises sur l'erreur est la division d'intervalles : l'utilisateur défini jusqu'à deux variables dont l'intervalle de valeurs est découpé en sous-intervalles de tailles égales. L'analyse est ensuite réalisée indépendamment sur chaque sous-intervalle, l'erreur globale étant le maximum de toutes les erreurs obtenues. Cette technique permet d'améliorer l'approximation calculée, mais est coûteuse en temps d'exécution et demande une bonne connaissance du problème afin de choisir des intervalles à découper pertinents.

PRECiSA [Moscato *et al.*, 2017, Titolo *et al.*, 2018] est un analyseur statique qui estime l'erreur d'arrondi de programme sur les nombres à virgule flottante. En plus de l'estimation de l'erreur produite, PRECiSA génère un certificat de preuve assurant la correction des bornes calculées. Ce certificat est vérifié de façon externe avec l'assistant de preuve PVS [Owre *et al.*, 1992]. PRECiSA utilise une représentation symbolique des erreurs et raisonne par rapport à leur équivalent sur les réels. Un nombre à virgule flottante est représenté par une paire d'entiers $(m, e) \in \mathbb{Z}^2$ où m est la mantisse et e est l'exposant du nombre à virgule flottante (voir définition 2.1.1). Un nombre à virgule flottante $\tilde{v}$ est l'approximation en machine d'un nombre réel r et son erreur d'arrondi est notée e. À partir du modèle d'arrondi en ulp (voir définition 2.3.4) et de la notation des nombres choisie, l'équation 4.8 donne la formule permettant d'exprimer l'erreur sur une opération.

$$\phi_\diamond(r_i)_{i=1}^n \wedge \phi_{\tilde{\diamond}}(\tilde{v}_i)_{i=1}^n \implies |R(\tilde{\diamond}(\tilde{v}_i)_{i=1}^n) - \diamond(r_i)_{i=1}^n| \leq \epsilon_{\tilde{\diamond}}(r_i, e_i)_{i=1}^n \tag{4.8}$$

Dans l'équation 4.8, les fonctions ϕ sont des fonctions booléennes exprimant les propriétés à satisfaire sur $\mathbb{R}$ et sur $\mathbb{F}$ pour une opération $\diamond$, R est une fonction de conversion pour faire référence au nombre réel représenté par un nombre à virgule flottante donné, et ϵ est une fonction donnant l'expression de l'erreur pour une opération sur les nombres à virgule flottante. Il peut exister plusieurs formules pour une opération sur les nombres à virgule flottante qui satisfont l'équation 4.8. L'équation 4.9 donne le résultat de l'application de l'équation 4.8 aux quatre opérations arithmétiques.

$$
\begin{aligned}
\epsilon_{\tilde{+}}(r_1,e_1,r_2,e_2) &= e_1 + e_2 + \frac{1}{2}\,\mathrm{ulp}(|r_1 + r_2| + e_1 + e_2) \\
\phi_+(r_1,r_2) &= \mathit{true},\ \phi_{\tilde{+}}(\tilde{v_1},\tilde{v_2}) = \mathit{true} \\
\epsilon_{\tilde{-}}(r_1,e_1,r_2,e_2) &= e_1 + e_2 + \frac{1}{2}\,\mathrm{ulp}(|r_1 - r_2| + e_1 + e_2) \\
\phi_+(r_1,r_2) &= \mathit{true},\ \phi_{\tilde{+}}(\tilde{v_1},\tilde{v_2}) = \tilde{v_2}\tilde{/}2 > \tilde{v_1} \vee \tilde{v_1} > 2\tilde{*}\tilde{v_2} \\
\epsilon_{\tilde{-}}(r_1,e_1,r_2,e_2) &= e_1 + e_2 \\
\phi_+(r_1,r_2) &= \mathit{true},\ \phi_{\tilde{+}}(\tilde{v_1},\tilde{v_2}) = \tilde{v_2}\tilde{/}2 \leq \tilde{v_1} \wedge \tilde{v_1} \leq 2\tilde{*}\tilde{v_2} \\
\epsilon_{\tilde{*}}(r_1,e_1,r_2,e_2) &= |r_1|e_2 + |r_2|e_1 + e_1e_2 + \frac{1}{2}\,\mathrm{ulp}((|r_1| + e_1)(|r_2| + e_2)) \\
\phi_+(r_1,r_2) &= \mathit{true},\ \phi_{\tilde{+}}(\tilde{v_1},\tilde{v_2}) = \mathit{true} \\
\epsilon_{\tilde{/}}(r_1,e_1,r_2,e_2) &= \frac{|r_1|e_2 + |r_2|e_1}{r_2r_2 - e_2|r_2|} + \frac{1}{2}\,\mathrm{ulp}\left(\frac{|r_1| + e_1}{|r_2| - e_2}\right) \\
\phi_+(r_1,r_2) &= r_2 \neq 0,\ \phi_{\tilde{+}}(\tilde{v_1},\tilde{v_2}) = \tilde{v_2} \neq 0
\end{aligned}
\tag{4.9}
$$

Les fonctions ϕ sont définies pour l'opération sur les réels et pour l'opération sur les flottants. Elles doivent être vrai pour que l'expression ϵ de l'opération correspondante soit valide.

PRECiSA utilise une sémantique concrète dénotationnelle, structurelle, et compositionnelle : elle capture les expressions des erreurs et se base sur la notation de l'équation 4.8. Cette sémantique traite aussi les appels de fonctions et propose une représentation correcte des différents chemins d'exécution d'un programme. La sémantique concrète utilisée par PRECiSA est calculable dans certains cas [2], mais nécessite l'utilisation d'une sémantique abstraite dans le cas général. La forme symbolique de l'erreur est obtenue sur une abstraction de la sémantique concrète. Cette abstraction est également applicable aux programmes récursifs. Dans ce cas, elle offre une convergence dans un nombre fini d'itérations et réduit l'explosion combinatoire due au traitement correct des différents chemins d'exécution (en particulier pour les expressions *if-then-else* imbriquées). En effet, la sémantique concrète produit une borne conditionnelle de l'erreur pour chaque combinaison de chemins d'exécution sur les réels et sur les nombres à virgule flottante. Cela garantie un traitement correct des tests instables mais produit en pratique quatre bornes conditionnelles de l'erreur pour chaque *if-then-else*. Contrairement à la sémantique concrète qui explicite chaque borne conditionnelle, la sémantique abstraite utilise une fonction permettant de sur-approximer ces bornes de l'erreur.

Une fois l'expression symbolique de l'erreur obtenue, il est important de calculer sa valeur numérique. Pour cela, PRECiSA utilise le solveur Kodiak [Smith *et al.*, 2015], afin de maximiser l'expression symbolique de l'erreur. Ce solveur applique un branch-and-bound formellement vérifié [Narkawicz et Muñoz, 2013] pour borner l'erreur à l'aide de l'arithmétique des intervalles ou des bases de Bernstein [Lorentz, 1986]. La condition d'arrêt du branch-and-bound est soit la précision choisie pour l'erreur ou bien la profondeur maximale d'exploration autorisée.

4.2 Optimisation globale

L'optimisation globale est une branche des mathématiques appliquées et de l'analyse numérique qui cherche à trouver le maximum, ou minimum, d'une fonction. Plusieurs méthodes

2. En particulier pour des programmes sans conditions et non récursifs.

sont issues de l'optimisation globale pour calculer des sur-approximations d'erreurs. Les principales méthodes sont celles utilisées dans FPTaylor [Solovyev *et al.*, 2015, Solovyev *et al.*, 2018], Real2Float [Magron *et al.*, 2017], et FPSDP [Magron, 2018]. Ces méthodes ont en commun l'expression du calcul d'erreurs sous forme de problème de maximisation. L'équation 4.10 reprend ce problème d'optimisation où $\mathbf{x} \in \mathbf{X}$ est un vecteur des variables d'entrée dans l'ensemble des variables du programme et r^* est l'erreur maximale que peut produire la fonction $\tilde{f}$ par rapport à son équivalent exact f.

$$r^* = \max_{\mathbf{x} \in \mathbf{X}} |\tilde{f}(\mathbf{x}) - f(\mathbf{x})| \tag{4.10}$$

Résoudre ce problème est très difficile dans le cas général : la fonction f est fortement irrégulière et discontinue. C'est pourquoi les approches à base d'optimisation globale travaillent sur une relaxation du problème afin de sur-approximer les erreurs calculées, comme donné dans l'équation 4.11. Cette relaxation décompose l'erreur en deux termes l et h, où l est un terme affine et h est non linéaire.

$$r^* \leq \max_{\mathbf{x} \in \mathbf{X}} |l(\mathbf{x})| + \max_{\mathbf{x} \in \mathbf{X}} |h(\mathbf{x})| \tag{4.11}$$

La valeur du premier terme l est en général obtenue à l'aide d'un algorithme d'optimisation globale tandis que la valeur du second terme h est le plus souvent approximée avec des intervalles. Le modèle d'arrondi utilisé est celui donné de la définition 2.3.3, où ϵ et δ bornent respectivement l'erreur relative et l'erreur absolue [3] introduites par l'opérateur d'arrondi. Ces approches calculent des sur-approximations de l'erreur et souffrent donc du même problème que les méthodes à base d'abstractions.

FPTaylor [Solovyev *et al.*, 2015, Solovyev *et al.*, 2018] utilise des séries de Taylor symbolique couplées à des méthodes d'optimisation globale pour borner les erreurs. L'équation 4.12 donne une approximation du problème d'optimisation initial où les erreurs du modèle d'arrondi sont explicites. La fonction $\tilde{f}$ prend maintenant trois paramètres : $\mathbf{x} \in \mathbf{X}$ un vecteur de variables d'entrée dans l'ensemble des variables du problème, et $\mathbf{e}$ et $\mathbf{d}$ les erreurs associées à chaque variable par rapport au modèle d'arrondi de la définition 2.3.3. Toutefois, cette approximation reste compliquée à calculer : elle introduit $2k$ nouvelles variables pour $\mathbf{e}$ et $\mathbf{d}$, où k est le nombre d'opérations sur $\mathbb{F}$ potentiellements inexactes.

$$r^* \leq \max_{\mathbf{x} \in X} |\tilde{f}(\mathbf{x},\mathbf{e},\mathbf{d}) - f(\mathbf{x})| \tag{4.12}$$

Ce problème d'optimisation peut être simplifié en exprimant $\tilde{f}(\mathbf{x},\mathbf{e},\mathbf{d})$ sous forme de série de Taylor, comme montré dans l'équation 4.13. L'expression du terme R_2 est détaillée dans [Solovyev *et al.*, 2018].

$$\tilde{f}(\mathbf{x},\mathbf{e},\mathbf{d}) = \tilde{f}(\mathbf{x},\mathbf{0},\mathbf{0}) + \sum_{i=1}^{k} \frac{\partial \tilde{f}}{\partial e_i}(\mathbf{x},\mathbf{0},\mathbf{0}) e_i + R_2(\mathbf{x},\mathbf{e},\mathbf{d}) \tag{4.13}$$

Sachant que la fonction $\tilde{f}(\mathbf{x},\mathbf{0},\mathbf{0})$ est égale à $f(\mathbf{x})$, d'après le modèle d'arrondi utilisé, il est donc possible d'exprimer l'erreur de l'équation 4.12 sous forme de problème d'optimisation

3. Pour les nombres proche de zéro.

plus simple à calculer. Ce nouveau problème est donné dans l'équation 4.14 où M_2 est une sur-approximation calculable de $R_2(\mathbf{x},\mathbf{e},\mathbf{d})$.

$$r^* \leq \max_{\mathbf{x}\in\mathbf{X},|e_i|<\epsilon} \left| \sum_{i=1}^{k} \frac{\partial \tilde{f}}{\partial e_i}(\mathbf{x},\mathbf{0},\mathbf{0}) e_i \right| + M_2 \tag{4.14}$$

Le terme à maximiser dans l'équation 4.14 est le premier terme : il est exprimé sous forme de séries de Taylor symbolique, puis est donné à un solveur d'optimisation globale. FPTaylor est capable d'utiliser plusieurs solveurs pour résoudre ce problème : un branch-and-bound simple à base d'intervalles interne à FPTaylor, NLopt [Johnson, 2017] une librairie pour l'optimisation non-linéaire, Z3 [de Moura et Bjørner, 2008] un solveur SMT, ou bien Gelpia [Baranowski et Briggs, 2017] un branch-and-bound *coopératif* [4] à base d'intervalles proposé par les auteurs de FPTaylor. NLopt n'est pas rigoureux et peut parfois produire des résultats incorrects, tandis que Z3 supporte les contraintes d'inégalités, mais n'est pas adaptés pour traiter les fonctions transcendantes ou discontinues. Le solveur le plus efficace pour résoudre ce problème est Gelpia. Il trouve une limite rigoureuse supérieure au maximum global d'une fonction multi-variée sur un intervalle donné. Cela signifie que sa solution est garantie d'être supérieure au maximum global sur l'intervalle lorsqu'elle est évaluée à l'aide de l'arithmétique réelle. Il utilise la librairie GAOL [Goualard, 2017] pour l'arithmétique des intervalles. Le deuxième terme de l'erreur M_2 est obtenu par arithmétique des intervalles ou un nombre fini d'itérations d'un solveur d'optimisation globale.

Real2Float [Magron *et al.*, 2017] repose sur la programmation semi-définie positive [Wolkowicz *et al.*, 2000] pour borner les erreurs. Contrairement à FPTaylor, l'erreur δ est négligée dans le modèle d'arrondi de la définition 2.3.3 : cette approche ne prend donc pas les nombres à virgule flottante dénormalisés (voir définition 2.1.3) en compte. Real2Float, note $\mathbf{K} = \mathbf{X} \times \mathbf{E}$ où $\mathbf{X}$ et $\mathbf{E}$ représentent respectivement l'ensemble des variables d'entrée et l'ensemble de leurs erreurs d'arrondi. De la même façon que pour FPTaylor, l'erreur est décomposée en deux termes, comme montré dans l'équation 4.15 où r^* est le maximum de $r(\mathbf{x},\mathbf{e})$ la fonction calculant l'erreur absolue (voir définition 2.3.1) sur $\mathbf{K}$.

$$r^* \leq \max_{(\mathbf{x},\mathbf{e})\in\mathbf{K}} |l(\mathbf{x},\mathbf{e})| + \max_{(\mathbf{x},\mathbf{e})\in\mathbf{K}} |h(\mathbf{x},\mathbf{e})| \tag{4.15}$$

Une façon d'obtenir l est de calculer le vecteur de dérivée partielle par rapport à $\mathbf{e}$ évalué en $(\mathbf{x},0)$ et de prendre le produit scalaire de ce vecteur et de $\mathbf{e}$. L'expression de l est donnée dans l'équation 4.16. Ce terme est affine par rapport à $\mathbf{e}$.

$$l(\mathbf{x},\mathbf{e}) = \epsilon \sum_{j=1}^{m} \frac{\partial r(\mathbf{x},\mathbf{e})}{\partial e_j}(\mathbf{x},\mathbf{0}) \frac{e_j}{\epsilon} \tag{4.16}$$

La maximisation de l repose sur l'utilisation de relaxation en programmation semi-définie positive [Lasserre, 2006], tandis que h est approximé à partir d'intervalles. Cette approche génère également des certificats *sum of squares (SOS)* [Parrilo et Thomas, 2020], afin de garantir la validité des approximations obtenues. Ces certificats sont facilement vérifiables dans Coq [The Coq Development Team, 2020].

4. Gelpia est coopératif dans le sens où les algorithmes d'approximation sont exécutés de manière concurrente.

FPSDP [Magron, 2018] est une approche, similaire à celle de Real2Float, calculant une sous-approximation de l'erreur maximale produite par un programme. Comme pour les autres méthodes à base d'optimisation globale, l'erreur est décomposée en deux termes. Mais contrairement à l'équation 4.15, la somme des deux termes est remplacée par leur différence afin d'obtenir une borne inférieure. L'équation 4.17 reprend cette formulation du problème.

$$r^* \geq \max_{(\mathbf{x},\mathbf{e})\in\mathbf{K}} |l(\mathbf{x},\mathbf{e})| - \max_{(\mathbf{x},\mathbf{e})\in\mathbf{K}} |h(\mathbf{x},\mathbf{e})| \tag{4.17}$$

Approximer h par intervalles est suffisant, car le terme est en général négligeable par rapport à l. En faisant la différence entre la borne supérieure de h et n'importe quelle borne inférieure calculée pour l, le résultat est une borne inférieure pour r^*. Le cœur de cette méthode est donc d'approximer l : là où Real2Float génère une hiérarchie de bornes supérieures convergentes pour borner l à l'aide de relaxations issues de l'optimisation semi-définie positive, FPSDP cherche à obtenir une hiérarchie de bornes inférieures, inspirée de [Lasserre, 2011]. La borne inférieure sur l est obtenue par l'équation 4.18 où $\underline{l} = \min_{(\mathbf{x},\mathbf{e})\in\mathbf{K}} l(\mathbf{x},\mathbf{e})$ et $\bar{l} = \max_{(\mathbf{x},\mathbf{e})\in\mathbf{K}} l(\mathbf{x},\mathbf{e})$.

$$l^* = \max_{(\mathbf{x},\mathbf{e})\in\mathbf{K}} |l(\mathbf{x},\mathbf{e})| = \max\{|\underline{l}|, |\bar{l}|\} \tag{4.18}$$

FPSDP propose trois méthodes pour résoudre l'équation 4.18 : une hiérarchie de problèmes de valeurs propres généralisées, une hiérarchie de bornes utilisant uniquement des opérations élémentaires, et une hiérarchie de relaxations de programmation semi-définie positive. Chaque méthode prend en entrée un entier k, le *niveau de relaxation*, afin de borner la résolution. La méthode obtenant les meilleures bornes est celle basée sur la programmation semi-définie positive.

Cette approche repose sur une sous-approximation pour fournir une borne inférieure de l'erreur maximale d'un programme. Mais la sous-approximation calculée n'est pas toujours correcte : elle peut souffrir d'une sur-approximation. Dans certains cas, la sous-approximation obtenue est plus grande qu'une sur-approximation correcte de l'erreur calculée par d'autres techniques. L'exemple 4.2.1 illustre ce problème sur un programme commun dans la communauté d'analyse d'erreurs sur les nombres à virgule flottante.

Exemple 4.2.1 – Soit `rigidBody1`, un programme issu de la suite FPBench [Damouche *et al.*, 2017b], reprit dans le programme 4.1 écrit en C où les variables d'entrée $x1, x2, x3$ prennent leurs valeurs dans l'intervalle $[-15, 15]$.

```
double rigidBody1(double x1, double x2, double x3)
{
  return -x1 * x2 - 2 * x2 * x3 - x1 - x3;
}
```

Programme 4.1: rigidBody1.

FPSDP calcule une borne inférieure pour l'erreur maximale de $3{,}55 \times 10^{-13}$ (résultat provenant de [Magron, 2018]). Gappa, Daisy, et FPTaylor donnent une borne supérieure de l'erreur maximale de $2{,}95 \times 10^{-13}$, Fluctuat et Rosa une borne supérieure de $3{,}22 \times 10^{-13}$, et PRECiSA une borne supérieure de $3{,}23 \times 10^{-13}$ (résultats provenant de [Garcia *et al.*, 2020b]).

La seule borne plus grande que celle calculée par FPSDP est celle obtenue par un outil du même auteur et utilisant la même méthode : Real2Float. La borne est de $5{,}33 \times 10^{-13}$ (résultat provenant de [Garcia *et al.*, 2020b]). La sous-approximation de l'erreur maximale calculée par

FPSDP n'est donc pas correcte et souffre d'une sur-approximation. Cette sur-approximation est probablement due à la procédure de relaxation. En effet, pour un niveau de relaxation k de 4 au lieu de 8, la borne inférieure produite pour `rigidBody1` est de $1{,}62 \times 10^{-13}$. Cette borne est donc valide par rapport aux bornes supérieures produites par les autres outils. Néanmoins, FPSDP n'offre aucune indication sur le niveau de relaxation à appliquer afin d'obtenir une borne correcte.

4.3 Approches basées sur la satisfiabilité modulo théories

Un problème de *satisfiabilité modulo théories* [Biere *et al.*, 2009], ou SMT, est un problème de décision pour des formules logiques du premier ordre avec égalités et sans quantificateur [5]. Dans ces formules certains symboles de prédicats et certaines fonctions sont définis dans des théories. Il existe de nombreuses théories comme celle des nombres réels, de l'arithmétique linéaire, ou de diverses structures de données : listes, tableaux, tableaux de bits, ... La théorie pour les nombres à virgule flottante [Brain *et al.*, 2015] utilise des vecteurs de bits pour représenter les nombres à virgule flottante. Un solveur SMT peut être divisé en deux parties : un solveur SAT [6] et un, ou plusieurs, solveur de théorie. La résolution se déroule comme suit : la formule logique de premier ordre est d'abord traduite en formule logique propositionnelle puis résolue par le solveur SAT. S'il existe une solution, le solveur de théorie applique une procédure pour vérifier qu'elle est valide dans la théorie choisie. Sinon, la formule n'est pas satisfiable. Un solveur SMT utilise également le mécanisme de *retour en arrière*, ou *backtrack*, afin d'annuler un choix et explorer d'autres parties de l'espace de recherche. L'exemple 4.3.1 illustre le type de formules accepté par un solveur SMT, ainsi que sa traduction pour le solveur SAT.

Exemple 4.3.1 – Soit la formule de logique du premier ordre de l'équation 4.19 accepté par un solveur SMT.

$$(2 < x \vee x < 4) \wedge (x = 8 - y) \wedge (y > 0) \tag{4.19}$$

L'équation 4.19 est traduite en logique propositionnelle, pour un solveur SAT, dans l'équation 4.20

$$(p \vee q) \wedge r \wedge s \tag{4.20}$$

Les solveurs SMT ont été largement appliqués à la vérification de programme, notamment en vérification de modèles [Tinelli, 2012], ou *model checking*. Des outils comme ESBMC [Gadelha *et al.*, 2018], 2LS [Schrammel *et al.*, 2017], ou CDFL [D'Silva *et al.*, 2012] utilisent des solveurs SMT pour vérifier des programmes sur les nombres à virgule flottante.

Certaines approches en analyse d'erreurs utilisent également des solveurs SMT pour approximer l'erreur issue de programmes sur les nombres à virgule flottante. Rosa [Darulova et Kuncak, 2014, Darulova et Kuncak, 2017] et Daisy [Izycheva et Darulova, 2017, Darulova *et al.*, 2018b, Darulova *et al.*, 2018a, Darulova et Volkova, 2019] sont deux outils qui utilisent un solveur SMT pour calculer les erreurs produites par un programme.

5. Le support de quantificateur peut être ajouté via des théories dédiées.
6. SAT : problème de satisfaisabilité booléenne, voir [Biere *et al.*, 2009].

Rosa [Darulova et Kuncak, 2014, Darulova et Kuncak, 2017] est un compilateur *source-à-source* capable de faire de la synthèse de programme. Il prend en entrée le code d'un programme écrit dans un langage de spécification non-exécutable avec un type de valeur sur les réels et produit un code source avec un type de valeur fini approprié (nombre à virgule flottante ou fixe). Rosa peut également être utilisé comme outil d'analyse. En définissant un type de sortie pour les valeurs, il calcule l'intervalle des valeurs sur les réels et borne l'erreur de calcul. Pour réaliser cette analyse, Rosa combine l'arithmétique des intervalles [Moore *et al.*, 2009] et le solveur SMT non-linéaire Z3 [de Moura et Bjørner, 2008]. L'utilisation d'un solveur SMT permet à Rosa de prendre en compte des contraintes additionnelles arbitraires sur les valeurs d'entrées, *e.g.,* une inégalité ou égalité sur les valeurs prisent par une variable. L'erreur globale est décomposée en erreurs d'arrondi et en propagation des erreurs initiales sur les entrées afin d'être calculée. Les erreurs d'arrondi sont calculées à l'aide de l'arithmétique affine [de Figueiredo et Stolfi, 2004] (voir exemple 4.1.3), tandis que la propagation d'erreurs initiales est estimée avec une approximation de Taylor de premier ordre (voir section 4.2). Contrairement à FPTaylor, l'approximation de Taylor est ici développée par rapport aux entrées du programme. L'approximation de Taylor est évaluée automatiquement à l'aide du solveur SMT non-linéaire Z3. Rosa utilise le modèle d'arrondi de la définition 2.3.3 pour borner l'erreur d'arrondi à chaque opération. L'appel à un solveur SMT est un processus coûteux. C'est pourquoi Rosa propose plusieurs optimisations empiriques minimisant son utilisation. Rosa fait le choix d'utiliser un solveur SMT afin d'évaluer numériquement les intervalles de valeurs et d'erreurs pour les variables d'un programme, mais les expressions produites par Rosa pourraient être passées à n'importe quel solveur capable de résoudre des expressions non-linéaires.

Daisy [Izycheva et Darulova, 2017, Darulova *et al.*, 2018b, Darulova *et al.*, 2018a, Darulova et Volkova, 2019] est un outil réalisé par les mêmes auteurs que Rosa. Il se veut modulaire et extensible et combine plusieurs techniques d'analyse statique et dynamique de programme sur les nombres à virgule flottante. Outre les techniques d'analyse introduite dans Rosa, Daisy applique l'analyse d'erreurs à base d'optimisation globale de FPTaylor ou la division d'intervalles de Fluctuat. En plus du solveur SMT Z3, Daisy supporte le solveur SMT dReal [Gao *et al.*, 2013]. Les fonctions transcendantes sont également traitées, par analyse du flux de données [Khedker *et al.*, 2009], ou *data-flow analysis*, en suivant l'approche de [Darulova et Kuncak, 2011]. Une des particularités de cet outil est qu'il calcule directement l'erreur relative [Izycheva et Darulova, 2017], contrairement aux autres approches qui peuvent l'inférer à partir de l'erreur absolue qu'elles produisent. À noter que les bornes de l'erreur absolue et de l'erreur relative ne sont pas corrélées, *i.e.,* le maximum de l'erreur absolue n'est pas le même que celui de l'erreur relative.

4.4 Assistant de preuves

Un assistant de preuves [Geuvers, 2009] a pour objectif d'écrire et de vérifier des preuves mathématiques. Ces preuves s'appliquent souvent sur des théorèmes, au sens mathématique, ou bien sur des assertions relatives à un programme afin de prouver la correction du programme. Il existe différentes formalisations de l'arithmétique des nombres à virgule flottante dans les assistants de preuve [Harrison, 2006, Melquiond, 2012]. Plusieurs outils dans ce domaine supportent séparément l'analyse séquentielle d'un programme et l'analyse des boucles et conditions de branchement [Boldo *et al.*, 2009, Boldo et Melquiond, 2011, Boldo *et al.*, 2013, Goodloe *et al.*,

2013, Boldo *et al.*, 2015, Damouche *et al.*, 2017a]. En général, ces approches décomposent le problème global en sous-problèmes, équivalents à des lignes de code du programme, qui sont plus simples à vérifier. Ces sous-problèmes sont souvent résolut par des outils dédiés. Cette section ne s'intéresse pas directement aux assistants de preuves permettant de vérifier formellement des programmes sur les nombres à virgule flottante, mais plutôt à l'outil Gappa [Daumas et Melquiond, 2010] intégrés dans ces assistants. Gappa est utilisé dans certaines de ces approches pour borner l'erreur d'arrondi des expressions d'un programme.

Gappa [Daumas et Melquiond, 2010] est un outil pour vérifier formellement des propriétés de programmes sur les nombres à virgule flottante et est utilisé dans le vérificateur Frama-C [Kirchner *et al.*, 2015]. Il repose sur l'arithmétique des intervalles [Moore *et al.*, 2009] et des règles de réécriture afin d'améliorer les résultats calculés. Dans Gappa un programme s'exprime sous forme de propositions logiques qui sont ensuite vérifiées afin de générer des preuves formelles. À partir d'une analyse du flux de données d'un programme Gappa est capable de fournir une borne sur-approximant les erreurs de manière automatique. Il reste toutefois plus efficace sur des propriétés précises de l'arithmétique des nombres à virgule flottante. Dans ce cas, il est possible de donner des *indices* à Gappa afin d'améliorer la résolution et d'obtenir une sur-approximation plus précise de l'erreur. Il génère aussi un certificat de preuve pouvant être vérifié dans Coq [The Coq Development Team, 2020]. Le modèle d'arrondi utilisé dans Gappa est celui de la définition 2.3.4. L'exemple 4.4.1, issue de la documentation de Gappa, illustre comment une formule logique est traduite et donnée en entrée à Gappa.

Exemple 4.4.1 – Soit la formule logique de l'équation 4.21 sur les nombres à virgule flottante.

$$c \in [-0{,}3; -0{,}1] \wedge (2a \in [3;4] \Rightarrow b + c \in [1;2]) \wedge a - c \in [1{,}9; 2{,}05] \Rightarrow b + 1 \in [2; 3{,}5] \quad (4.21)$$

Supposons que l'équation 4.21 est traduite dans le programme 4.2 afin d'être donné en entrée à Gappa.

```
{
  c in [-0.3,-0.1] /\
  (2 * a in [3,4] -> b + c in [1,2]) /\
  a - c in [1.9,2.05]

  -> b + 1 in [2,3.5]
}

a -> a - c + c;
b -> b + c - c;
```

Programme 4.2: Formule logique transformée en script pour Gappa.

À l'aide des propriétés de l'arithmétique des nombres à virgule flottante, *e.g.,* l'opérateur d'arrondi implicite à chaque opération, et l'évaluation sur les intervalles, Gappa est capable de sur-approximer l'erreur générée par cette formule et de produire un certificat de preuve validable directement dans Coq.

4.5 Test avec stratégie d'exploration

Le test de programme [Myers, 2004] est souvent utilisé pour assurer la couverture du programme. Le test repose sur une simulation ou instrumentation du programme associée à un ensemble de valeurs d'entrée. Contrairement aux méthodes d'approximation, correctes par construction et englobant toutes les traces d'exécution d'un programme, les tests se concentrent sur un ensemble fini de traces d'exécution. Le test ne peut donc que garantir la validité de l'ensemble de traces d'exécution considéré.

En analyse d'erreurs, le test est employé afin de calculer l'erreur produite par un programme sur certaines exécutions. Pour ce faire, le programme est exécuté en *haute précision* [7], afin de simuler l'exécution sur les réels, en plus de son exécution dans le format de nombres à virgule flottante à vérifier. À partir des valeurs obtenues en sortie, l'erreur absolue (voir définition 2.3.1) est calculée en faisant la différence des deux résultats. Le même processus est utilisé pour l'erreur relative (voir définition 2.3.2). Cette erreur est complémentaire à l'approximation de l'erreur calculée en analyse statique. En effet, là où une sur-approximation de l'erreur considère toutes les exécutions possibles d'un programme, le test exhibe une seule trace d'exécution produisant une grande erreur.

De nombreux outils comme S^{3}FP [Chiang *et al.*, 2014], Atomu [Zou *et al.*, 2020], ou FPED [Xia *et al.*, 2020] appliquent une analyse dynamique, à base de tests aléatoires guidée par des stratégies d'exploration, afin de détecter les valeurs d'entrées produisant de grandes erreurs dans un programme. L'objectif de ces outils est de détecter des cas critiques que peut atteindre un programme : les cas où l'erreur produite par le programme est importante et potentiellement proche de l'erreur maximale. L'erreur maximale représente la plus grande erreur, en valeur absolue, que peut produire un programme. La sélection aléatoire de valeurs offre de bons résultats en pratique grâce à la distribution non-uniforme de l'erreur. Néanmoins, la forte densité [8] des intervalles sur les nombres à virgule flottante, couplée à la combinatoire de toutes les combinaisons de valeurs possibles pour les variables, ne permet pas une exploration exhaustive de l'espace de recherche dans un temps raisonnable. C'est pourquoi ces outils appliquent des stratégies d'exploration, afin de guider efficacement la recherche de grandes erreurs.

S^{3}FP [Chiang *et al.*, 2014] est un outil proposant une heuristique de recherche guidée pour détecter les valeurs d'entrée d'un programme qui cause de grandes erreurs. Cet algorithme, appelé *BGRT* pour *Binary Guided Random Testing*, travaille sur un partitionnement fini de l'espace de recherche. L'algorithme 4.3 donne la procédure de génération de partitions à partir d'une partition initiale *conf* avec N_{part} itérations. Une partition est un ensemble d'intervalles de valeurs pour chaque variable d'entrée du programme. La partition initiale est donc l'ensemble des intervalles donnant les valeurs que peuvent prendre les variables d'entrée du programme. BGRT commence par générer, de façon aléatoire, deux sous partitions incomplètes c_x et c_y à partir de la partition initiale. Chaque sous partition incomplète contient un sous-ensemble des intervalles de la partition initiale, tel que $c_x \cap c_y = \varnothing$. Ensuite, les intervalles de c_x et de c_y sont découpés en deux parties de tailles égales et permutés pour former deux nouvelles partitions complètes. Le processus est répété N_{part} fois, jusqu'à ce que *nextg* contienne l'ensemble des partitions générées. Dans l'algorithme 4.3, $\cup$ est l'union de deux partitions avec des domaines disjoints, tandis que $\uplus$ est la

7. Le plus souvent dans un format 128 bits.
8. La densité des nombres à virgule flottante est plus importante proche de zéro que des infinis.

création d'un ensemble de partitions et $\overbrace{c}$ (resp. $\underbrace{c}$) est la moitié supérieure (resp. inférieure) de la partition c. Sachant que $\overbrace{c_x} \cup \overbrace{c_y}$ (resp. $\underbrace{c_x} \cup \underbrace{c_y}$) est, pour n'importe quel c_x et c_y, égale à $\overbrace{conf}$ (resp. $\underbrace{conf}$), elle n'est ajoutée qu'une seule fois au début de la procédure.

Entrée : $conf, N_{part}$ *(* partition initiale et nombre d'itérations pour la génération de partitions *)*
Sortie : $nextg$ *(* ensemble des partitions générées *)*

$nextg \leftarrow \overbrace{conf} \uplus \underbrace{conf}$
pour i **de** 1 **à** N_{part} **faire**
  $(c_x, c_y) = PartConf(conf)$
  $nextg \leftarrow nextg \uplus (\overbrace{c_x} \cup \underbrace{c_y}) \uplus (\underbrace{c_x} \cup \overbrace{c_y})$
fin pour
retourner $nextg$

Algorithme 4.3: Génération de partitions pour BGRT (S^{3}FP).

L'exemple 4.5.1 donne un exemple de partitionnement via *PartConf* d'une partition c_p.

Exemple 4.5.1 – Soit une partition c_p, donnée dans 4.22, avec I_0, I_1, et I_2 les intervalles des variables d'entrées du programme.

$$c_p : \left\{ \begin{array}{ll} I_0 & \mapsto [-1;1] \\ I_1 & \mapsto [0{,}1;0{,}2] \\ I_2 & \mapsto [-0{,}2;-0{,}1] \end{array} \right\} \tag{4.22}$$

$$c_q : \left\{ \begin{array}{ll} I_0 & \mapsto [-1;1] \\ I_2 & \mapsto [-0{,}2;-0{,}1] \end{array} \right\} \tag{4.23}$$

$$c_r : \left\{ I_1 \quad \mapsto [0{,}1;0{,}2] \right\} \tag{4.24}$$

La partition initiale est découpée en deux sous partitions incomplètes c_q et c_r strictement plus petites, comme montré dans 4.23 et 4.24.

BGRT va, à partir de l'ensemble des partitions, évaluer chaque partition k fois : le programme prend en entrée des valeurs aléatoires dans la partition et s'exécute en précision machine (32 bits) et en haute précision (128 bits) pour simuler une exécution sur les réels. La différence entre le résultat sur 128 bits et le résultat sur 32 bits donne la formule de l'erreur absolue pour les valeurs d'entrée sélectionnées. À noter que la différence entre les deux résultats étant effectuée sur $\mathbb{F}$, l'erreur obtenue souffre d'arrondi et peut ne pas être exacte.

Cette étape est appliquée pour chaque partition, et l'erreur la plus grande est conservée avec les valeurs qui la produisent. Pour éviter de rester bloquer dans un maximum local, BGRT applique un mécanisme de redémarrage, ou *restart*, afin de relancer la recherche sur la partition initiale. Ce redémarrage est provoqué avec une certaine probabilité. S^{3}FP se limite à l'évaluation de programme sur 32 bits, car la simulation d'une exécution sur les réels au format 128 bits n'est pas suffisante pour correctement mesurer l'erreur sur un programme 64 bits (ou plus) sans aggraver l'approximation de l'erreur causée par l'opérateur d'arrondi nécessaire à l'arithmétique des nombres à virgule flottante (voir chapitre 2). La stratégie d'exploration de S^{3}FP, malgré l'utilisation d'un algorithme

de découpe, reste guidée par de l'aléatoire. Même si le découpage en sous partitions permet à l'algorithme d'explorer différentes parties de l'espace de recherche, la sélection d'une sous partition ne dépend d'aucun critère augmentant les chances de trouver une plus grande erreur.

Atomu [Zou *et al.*, 2020] est un outil basé sur la notion de *conditionnement*, une mesure en analyse numérique [Higham, 2002] représentant la dépendance d'un problème numérique par rapport aux données du problème, indépendamment de son implémentation. Cette mesure détecte des valeurs d'entrée pouvant causer grandes erreurs relatives (voir définition 2.3.2) dues à l'utilisation des nombres à virgule flottante à la place des nombres réels. L'équation 4.25 donne la formule du conditionnement pour une fonction f prenant une variable x.

$$\Gamma_f(x) = \left| \frac{x f'(x)}{f(x)} \right| \tag{4.25}$$

Le conditionnement mesure de combien l'erreur relative introduite par les valeurs d'entrées est amplifiée par la fonction f et donc fait abstraction de toute erreur introduite par l'implémentation de f. En général, calculer le conditionnement $\Gamma_f(x)$ est plus difficile que calculer la fonction $f(x)$ car sa dérivée $f'(x)$ ne peut être facilement obtenue sans avoir précalculé certaines quantités [Fu *et al.*, 2015]. C'est pourquoi Atomu calcule le conditionnement atomique : le conditionnement pour chaque opération [9] atomique du programme. Le code source du programme est donc nécessaire pour analyser le conditionnement de chaque opération et annoter le programme. Contrairement à S^{3}FP, le programme n'a pas besoin d'être simulé sur les réels. Ici, l'erreur n'est pas calculée pour détecter les valeurs d'entrée pouvant causer une grande erreur. Une exécution similaire à S^{3}FP peut être appliquée sur les résultats d'Atomu pour vérifier si les valeurs d'entrée sélectionnées par Atomu produisent effectivement une grande erreur. La table 4.1 donne la valeur des conditionnements pour les quatre opérations arithmétiques de base ainsi que la *zone de danger*, *i.e.,* les valeurs pour lesquelles les opérandes vont produire un grand conditionnement atomique.

Table 4.1 – Conditionnement atomique pour les opérations arithmétiques

Opération	Conditionnement atomique	Zone de danger
$op(x,y) = x + y$	$\Gamma_{+,x}(x,y) = \left\lvert \frac{x}{x+y} \right\rvert, \Gamma_{+,y}(x,y) = \left\lvert \frac{y}{x+y} \right\rvert$	$x \approx -y$
$op(x,y) = x - y$	$\Gamma_{-,x}(x,y) = \left\lvert \frac{x}{x-y} \right\rvert, \Gamma_{-,y}(x,y) = \left\lvert -\frac{y}{x-y} \right\rvert$	$x \approx y$
$op(x,y) = x \times y$	$\Gamma_{\times,x}(x,y) = \Gamma_{\times,y}(x,y) = 1$	-
$op(x,y) = x \div y$	$\Gamma_{\div,x}(x,y) = \Gamma_{\div,y}(x,y) = 1$	-

Les zones de danger semblent être en contradictions avec les propriétés des nombres à virgule flottante connus [Sterbenz, 1974] : pour $x \approx y$ la soustraction $x \ominus y$ est le plus souvent calculée exactement. Mais, ces zones de danger restent compatibles avec l'idée de conditionnement, où une erreur existante *peut* être amplifiée dans le résultat si la condition est satisfaite. De plus, le conditionnement est une formule calculée sur les réels, et ne prend donc pas en compte l'implémentation de la fonction, par exemple sur les nombres à virgule flottante. Atomu utilise un algorithme évolutionnaire [Bäck *et al.*, 2000] pour chercher les valeurs d'entrée causant des erreurs importantes. Cet algorithme évolutionnaire simule le processus de sélection naturelle pour résoudre des problèmes

9. Les opérations arithmétiques et les fonctions de base : $\sin, \cos, \tan, \ldots$

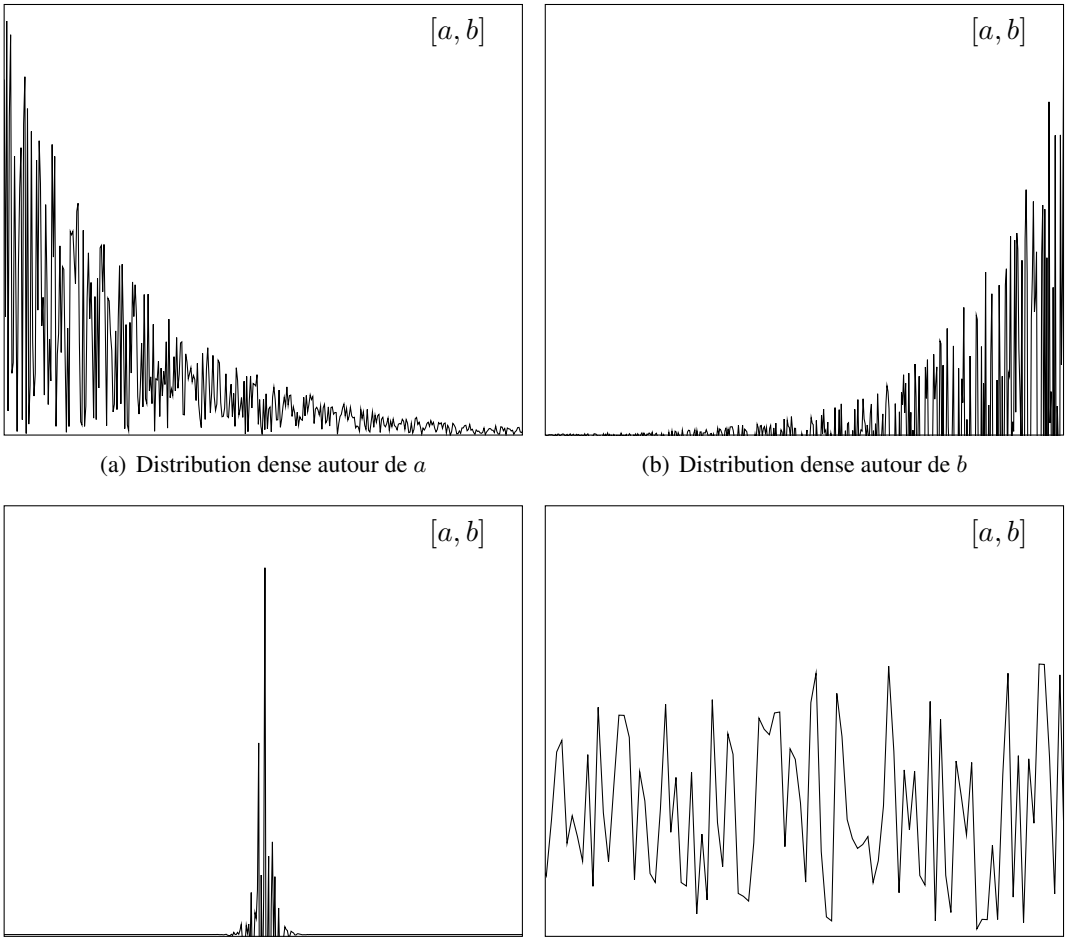
d'optimisation. Chaque solution candidate est appelée un individu et il existe une fonction d'évaluation déterminant la qualité des solutions. Pour Atomu, un individu est défini en temps qu'un ensemble de valeurs d'entrée sur $\mathbb{F}$, et la fonction d'évaluation est le conditionnement atomique sur une opération atomique. L'algorithme comprend trois étapes : l'*initialisation* génère aléatoirement des valeurs d'entrées et calcule les conditionnements atomiques pour chaque opération ; la *sélection* classe les individus en utilisant la fonction d'évaluation ; la *mutation* favorise les individus bien classés et en génère de nouveaux basés sur ces derniers. Les valeurs d'entrées obtenues par Atomu peuvent ensuite être vérifiées dans un programme externe qui applique la formule de l'erreur relative, de la même façon que S^{3}FP. Ce programme externe calcule l'erreur relative produite par ses valeurs d'entrée et permet de s'assurer que l'erreur est effectivement grande. À noter que Atomu ne s'intéresse pas directement aux erreurs produites par l'arithmétique des nombres à virgule flottante. Cela veut dire que Atomu peut ne pas détecter de *grandes erreurs* là où S^{3}FP en trouve une, et inversement. Ces deux approches ne sont donc pas directement comparables.

FPED [Xia *et al.*, 2020] est un inspecteur d'erreurs sur des expressions arithmétiques. Contrairement aux autres approches à base de test aléatoire, FPED analyse la distribution de l'erreur sur l'expression arithmétique avant de générer un ensemble de valeurs à tester. Afin d'obtenir une idée de la distribution de l'erreur, FPED applique une génération de valeurs, pour un faible nombre de valeurs, dans l'intervalle de l'expression. Générer un faible nombre de valeurs donne une idée suffisante de la distribution de l'erreur et permet de spécialiser une exploration plus approfondie. La génération de valeurs utilisée pour le test à petite échelle respecte la méthode nommée *RN-avg*, pour *Real Number average distribution generation method*, décrite ci-dessous. FPED propose trois méthodes de génération de valeurs. Elles s'appliquent sur un intervalle de test $[a, b]$ strictement plus petit que l'espace de recherche global. En fonction d'un paramètre N, avec $N > 0$, ces méthodes calculent un pas de génération *step* afin d'obtenir un ensemble de valeurs à tester. *RN-avg* est une méthode classique de génération de valeurs. Elle applique la formule $step = \frac{Length}{N}$, avec *Length* la taille de l'intervalle $[a, b]$. *FP-avg*, pour *Floating-Point average distribution generation method*, génère des valeurs en fonction des caractéristiques des nombres à virgule flottante. Pour *fp_num* la quantité de nombres à virgule flottante dans $[a, b]$, le pas de génération est calculé par la formule $step = \frac{fp\_num}{N}$. Cette méthode génère des valeurs plus proches de a et s'applique lorsque la densité des nombres à virgule flottante est plus importante autour de la borne inférieure de l'intervalle de test. Une telle méthode est efficace lorsque la borne inférieure de l'intervalle est proche de zéro, la densité des nombres à virgule flottante étant dans ce cas plus importante autour de a. *IFP-avg*, pour *Inverse Floating-Point average distribution generation method*, est similaire à *FP-avg*, mais les valeurs générées sont loin de zéro. Les valeurs générées sont donc plus proches de b, la borne supérieure de l'intervalle de test. Ces trois méthodes sont combinées dans un algorithme de détection d'erreurs générant des plages de valeurs à tester en fonction de la distribution de l'erreur.

L'algorithme commence par calculer la distribution de l'erreur dans l'intervalle de test en appliquant *RN-avg* pour la génération de valeurs. Une fois la distribution de l'erreur obtenue, elle est comparée, avec une tolérance de ε, aux quatre distributions de l'erreur présentées dans la figure 4.3.

L'algorithme produit des plages de valeurs sur l'intervalle de test en fonction de la distribution de l'erreur :

— Pour la sous-figure 4.3(a), la distribution des erreurs est plus dense autour de a, donc la méthode *FP-avg* est appliquée. Cette distribution apparait souvent quand l'expression



(a) Distribution dense autour de a

(b) Distribution dense autour de b

(c) Distribution avec une singularité

(d) Distribution irrégulière

Figure 4.3 – Quatre distributions de l'erreur sur $[a, b]$

contient des fonctions mathématiques et que l'intervalle de test a des valeurs autour de zéro.

— Dans la sous-figure 4.3(b), la distribution des erreurs est plus dense autour de b. Cette distribution est l'inverse de la distribution de la sous figure 4.3(a), donc la méthode *IFP-avg* est utilisée.

— La sous-figure 4.1(c) est une distribution présentant une singularité de l'erreur au milieu de l'intervalle. Pour cette distribution, la méthode de génération est une combinaison de *FP-avg* et *IFP-avg*. Soit $[m, n]$ l'intervalle de la singularité, avec m le point le plus important de la singularité. L'intervalle $[a, b]$ est découpé en deux intervalles : $[a, m+\xi]$ et $[m-\xi, b]$, avec m le point de découpe. À noter que ces deux intervalles ne sont pas disjoints, et que des valeurs peuvent être générées plusieurs fois. La plage de valeurs vérifiée expérimentalement pour ξ est très petite, avec $\xi = \frac{|a-b|}{100}$. *IFP-avg* est appliquée sur $[a, m + \xi]$ et *FP-avg* est appliquée sur $[m - \xi, b]$.

— La dernière distribution de la sous-figure 4.3(d) est une distribution irrégulière. L'algorithme ne détectant pas de caractéristique particulière, il applique les trois méthodes de

génération : *RN-avg*, *FP-avg*, et *IFP-avg* pour obtenir des plages de valeurs. Cette distribution apparait souvent lorsque l'intervalle de test est très petit.

L'algorithme utilise ces plages de valeurs sur l'espace de recherche global pour détecter de grandes erreurs. FPED réalise aussi une analyse de *points chauds*, ou *hotspots*, de l'erreur dans l'expression arithmétique. Ces points chauds correspondant aux parties de l'expression arithmétique qui produisent de grandes erreurs. À partir de cette analyse, FPED détermine les opérations dans l'expression qui mènent à cette erreur. Pour calculer les erreurs, FPED utilise la libraire MPFR [Fousse *et al.*, 2007], afin de simuler l'exécution sur les réels, avec une précision de 128 bits. La formule de l'erreur calculée est légèrement différente de celle de l'erreur relative classique (voir définition 2.3.2), le dénominateur de la fraction étant remplacé par l'ulp sur 64 bits de l'expression. Ce changement permet de refléter visuellement le degré d'approximation entre la valeur calculée et la valeur exacte.

4.6 Autres outils

De nombreux autres outils existent pour l'analyse de programme ou la vérification au sens large sur les nombres à virgule flottante. CADNA [Jézéquel et Chesneaux, 2008] estime l'erreur d'arrondi en remplaçant l'arithmétique des nombres à virgule flottante par l'arithmétique stochastique discrète [Vignes, 1993]. Un autre outil, PROMISE [Graillat *et al.*, 2019] utilise la même technique afin de réaliser un *delta-debugging* [Zeller, 2009] et détecter les précisions mixtes dans des programmes. Herbie [Panchekha *et al.*, 2015] améliore automatiquement l'exactitude des expressions sur les nombres à virgule flottante. Il a également été associé à Daisy afin de combiner les deux approches [Becker *et al.*, 2018]. Satire [Das *et al.*, 2020] est un analyseur statique utilisant différentes techniques comme l'analyse incrémentale, l'abstraction, ou l'utilisation judicieuse d'évaluation numérique et symbolique afin d'obtenir une analyse statique extensible. Cette analyse statique est capable de traiter des centaines de milliers d'opérations sur les nombres à virgule flottante.

Une liste plus complète des outils existants pour l'analyse de programme sur les nombres à virgule flottante est maintenue par FPBench [Damouche *et al.*, 2017b] sur leur site web [10].

4.7 Conclusion

L'analyse de programme est appliquée aux nombres à virgule flottante, en particulier pour l'analyse d'erreurs. Les approches existantes se classent en deux catégories : la sur-approximation correcte d'erreurs par analyse statique et la détection de grandes erreurs, le plus souvent à l'aide de tests en analyse dynamique. L'analyse statique calcule par construction des approximations correctes de l'erreur, c'est-à-dire que l'approximation est une borne supérieure valide pour toutes les erreurs d'un programme. Mais cette analyse souffre de faux positif : la détection d'une erreur, ou d'un bug, dans l'approximation ne se produit pas toujours en pratique lors de l'exécution du programme. Au contraire, les tests en analyse dynamique ne produisent pas de faux positifs, mais même la plus grande erreur calculée par ces méthodes ne peut pas garantir la correction d'un programme. Une analyse à base de tests ne couvre pas non plus l'ensemble des traces d'un programme, et doit donc choisir astucieusement comment explorer l'espace de recherche. Ces deux

10. http://fpbench.org/community.html

approches sont complémentaires, l'approximation obtenue en analyse statique est une borne supérieure pour l'erreur maximale produite par un programme, tandis que la plus grande erreur générée par des tests fournit une borne inférieure. Une combinaison de ces approches ouvre de nouvelles perspectives pour encadrer l'erreur maximale produite par un programme sur les nombres à virgule flottante, et détecter certains faux positifs.

Contributions

CHAPITRE 5

Un système de contraintes pour les erreurs d'arrondi

Dans ce chapitre nous présentons notre première contribution : un système de contraintes pour les erreurs d'arrondi liées aux opérations sur les nombres à virgule flottante. Ce système prend en entrée un CSP étendu aux erreurs. Dans un premier temps, nous présentons la modélisation d'un problème pour calculer des erreurs, puis nous détaillons la résolution d'un tel problème.

Nous avons vu dans les chapitres précédents que les nombres à virgule flottante sont présents dans de nombreuses applications critiques [Lions *et al.*, 1996, Slabodkin, 1998, Bureau d'Enquêtes et d'Analyse, 2012]. La précision finie des opérations sur les nombres à virgule flottante, due à la mémoire finie d'un ordinateur, est la cause de nombreuses erreurs de calcul. Ces erreurs représentent la différence entre l'exécution du programme en machine et son équivalent mathématique exact sur les réels. Une telle différence peut causer une divergence dans le flot d'exécution d'un programme et avoir des conséquences désastreuses à la fois humaines [General Accounting Office, 1992] et financières [Quinn, 1983].

Borner ces erreurs est un sujet majeur en analyse de programmes. Obtenir une borne supérieure de l'erreur permet de mieux connaître le comportement d'un programme et d'assurer que l'erreur ne dépasse jamais une certaine valeur. De nombreuses approches réalisent une analyse statique du programme pour obtenir une borne supérieure de l'erreur. Dans le cas général [Rice, 1953], l'analyse d'un programme est indécidable. De plus, la combinatoire des valeurs sur $\mathbb{F}$ pour les variables d'un programme est en général trop grande pour les analyser de manière exhaustive. C'est pourquoi ces approches raisonnent sur une sur-approximation du programme. Diverses méthodes sont appliquées à cette analyse : l'interprétation abstraite [Goubault et Putot, 2006, Goubault et Putot, 2011, Moscato *et al.*, 2017, Titolo *et al.*, 2018], l'optimisation globale [Solovyev *et al.*, 2015, Solovyev *et al.*, 2018, Magron *et al.*, 2017], ou bien encore l'analyse du flot de données d'un programme [Daumas et Melquiond, 2010, Darulova et Kuncak, 2014, Darulova et Kuncak, 2017, Izycheva et Darulova, 2017, Darulova *et al.*, 2018b, Darulova *et al.*, 2018a, Darulova et Volkova, 2019].

Notre approche s'insère dans cette problématique et utilise la programmation par contraintes (voir chapitre 3) afin d'obtenir une borne supérieure de l'erreur produite par un programme sur les nombres à virgule flottante. La programmation par contraintes permet de raisonner, à l'aide de contraintes exprimant des relations, sur les valeurs prises par les variables d'un problème. La séparation claire offerte entre la modélisation et la résolution d'un problème permet de représenter aisément un problème et d'utiliser des algorithmes génériques puissants pour trouver une solution. La programmation par contraintes a été étendue aux nombres à virgule flottante [Michel *et al.*, 2001, Michel, 2002, Botella *et al.*, 2006, Marre et Michel, 2010]. Les contraintes sur les flottants ont déjà été appliquées à la vérification de programme sur les nombres à virgule flottante [Ponsini *et al.*, 2016, Zitoun, 2018]. Toutefois, Ces contraintes ne raisonnent que sur les valeurs que peuvent prendre des variables sur $\mathbb{F}$ et ne permettent pas de capturer l'erreur inhérente aux opérations sur les flottants. Ici, Nous proposons donc des contraintes pour les erreurs d'arrondi dues aux opérations sur les nombres à virgule flottante. Nous étendons en premier la notion de CSP sur les nombres à virgule flottante pour prendre en compte l'erreur, afin de pouvoir modéliser des problèmes sur les erreurs. À cette fin, nous introduisons un nouveau domaine associé à chaque variable flottante qui représente son erreur. Nous définissons ensuite un filtrage dédié aux erreurs qui, couplé au filtrage existant pour les valeurs, permet de supprimer les erreurs inconsistantes des domaines des variables et donc de borner l'erreur.

5.1 Problématique

Notre objectif est de calculer une sur-approximation correcte de l'erreur : une borne supérieure que l'erreur ne dépasse jamais. Nous nous concentrons sur les opérations arithmétiques $(\oplus, \ominus, \otimes, \oslash)$ afin de pouvoir calculer exactement l'erreur sur $\mathbb{Q}$ lorsque les variables flottantes

sont réduites à une seule valeur. Le standard IEEE 754 [IEEE, 2008] définit plusieurs règles et cas particuliers pour ces opérations, tels que l'arrondi correct obligatoire ou l'évaluation de l'opération sans erreur.

Le reste des travaux dans ce chapitre est défini pour les quatre opérations arithmétiques $(\oplus, \ominus, \otimes, \oslash)$ sur les nombres à virgule flottante en simple et double précision. L'arrondi considéré ici est celui privilégié dans le standard IEEE 754 : l'arrondi au plus proche pair.

5.2 Modélisation

En programmation par contraintes, un problème s'exprime sous forme de CSP (voir définition 3.0.1). La définition 5.2.1 étend la notion de CSP aux erreurs.

Définition 5.2.1 (Problème de satisfaction de contraintes étendu aux erreurs)**.** Un problème de satisfaction de contraintes étendu aux erreurs, ou CSP sur les erreurs, est un triplet $\langle X, D, C\rangle$ où X est un ensemble de variables $\{x_1, \ldots, x_n\}$, D est un ensemble de domaines de valeurs $\{D_{x_1}, \ldots, D_{x_n}\}$, et de domaines d'erreurs $\{D_{e_{x_1}}, \ldots, D_{e_{x_n}}\}$, et C est un ensemble de contraintes $\{c_1, \ldots, c_n\}$. Ces contraintes portent sur les variables du problème et expriment des relations entre domaines de valeurs ou domaines d'erreurs.

La traduction d'un programme sur les nombres à virgule flottante sous forme de CSP est effectuée à l'aide d'une *forme statique à affectation unique* [Rosen *et al.*, 1988], ou *static single assignment form (SSA)*. Un programme se traduit facilement [Wotawa et Nica, 2007] sous forme de contraintes sur des domaines adaptés aux variables du problème. L'exemple 5.2.1 illustre la transformation d'un programme en C sur les nombres à virgule flottante vers un CSP étendu aux erreurs.

Exemple 5.2.1 – Soit `carbonGas`, un programme issu de la suite FPBench [Damouche *et al.*, 2017b], reprit dans le programme 5.1 écrit en C où la variable d'entrée v prend ses valeurs dans l'intervalle $[\frac{1}{10}, \frac{1}{2}]$.

```
double carbonGas(double v)
{
  double p =  35000000;
  double a = 0.401;
  double b = 0.0000427;
  double t = 300;
  double n = 1000;
  double k = 1.3806503e-23;

  return (((p + ((a * (n/v)) * (n/v))) * (v - (n * b))) - ((k * n) * t));
}
```

Programme 5.1: carbonGas.

Ce programme se traduit directement sous forme de CSP. Pour chaque variable du programme 5.1 il existe une variable dans X du même nom. Si cette variable est issue d'une constante, comme p, alors il est possible d'affecter son domaine sous forme de contrainte, en posant $p \leftarrow 35000000$, ou bien en réduisant son domaine à une valeur unique en écrivant $D_p = 35000000$. Lorsqu'une variable est équivalente à une variable d'entrée du programme, alors son domaine de valeurs est défini comme un intervalle, *e.g.,* $v \in \left[\frac{1}{10}, \frac{1}{2}\right]$.

Le CSP est donné dans 5.1.

$$\begin{aligned} v &\in \left[\frac{1}{10}, \frac{1}{2}\right] \\ p &\leftarrow 35000000 \\ a &\leftarrow 0{,}401 \\ b &\leftarrow 0{,}000\,042\,7 \\ t &\leftarrow 300 \\ n &\leftarrow 1000 \\ k &\leftarrow 1{,}380\,650\,3 \times 10^{-23} \\ r &= (((p + ((a \times \frac{n}{v}) \times \frac{n}{v})) \times (v - (n \times b))) - ((k \times n) \times t)) \end{aligned} \tag{5.1}$$

Les domaines d'erreurs des variables ne sont pas explicités au niveau de l'écriture d'un CSP, sauf si nécessaire. Ici, pour toute variable du CSP décrite dans 5.1, il existe à la fois un domaine de valeurs et un domaine d'erreurs, que nous allons définir dans la section suivante.

5.2.1 Variables et domaines d'un CSP

Pour un CSP traduit à partir d'un programme P, une *variable* x_i représente une variable sur les nombres à virgule flottante, déclarée dans P. Plusieurs types de variables se distinguent dans un CSP en fonction de leur rôle dans P : une *variable d'entrée* est une variable dont la valeur est donnée en entrée à P ; une *variable intermédiaire* est une variable dans P dont la valeur est inférée à partir d'opérations dans le programme ; une *constante* est une constante dans P dont la valeur est connue, unique, et indépendante de l'exécution de P. Pour une variable x_i, le *domaine de valeurs* D_{x_i} représente l'ensemble des valeurs que la variable peut prendre. Afin de prendre en compte les erreurs de calcul sur la variable x_i, un domaine supplémentaire lui est associé. Ce domaine, appelé le *domaine d'erreurs* et noté $D_{e_{x_i}}$ représente l'ensemble des valeurs que l'erreur sur le calcul des valeurs de la variable x_i peut prendre. D_{x_i} est un intervalle sur les nombres à virgule flottante, comme défini dans la définition 5.2.2.

Définition 5.2.2 (Domaine de valeurs d'une variable x_i)**.**

$$D_{x_i} = [\underline{v}, \overline{v}] = \{v \in \mathbb{F} \mid \underline{v} \leq v \leq \overline{v}\} \text{ avec } \underline{v}, \overline{v} \in \mathbb{F}$$

$D_{e_{x_i}}$ est un intervalle sur les nombres rationnels donné dans la définition 5.2.3.

Définition 5.2.3 (Domaine d'erreurs d'une variable x_i)**.**

$$D_{e_{x_i}} = [\underline{e}, \overline{e}] = \{e \in \mathbb{Q} \mid \underline{e} \leq e \leq \overline{e}\} \text{ avec } \underline{e}, \overline{e} \in \mathbb{Q}$$

Le domaine d'erreurs initial d'une variable est obtenu de façon différente en fonction de son type. Pour une *variable d'entrée*, son domaine d'erreurs initial est fixé par défaut à $\pm\frac{1}{2}\,\mathrm{ulp}$ de son domaine de valeurs. Ces bornes sont possibles grâce au standard IEEE 754 [IEEE, 2008]. Le domaine d'erreurs initial d'une *variable intermédiaire* est à $\pm\infty$. Il est par la suite réduit grâce au filtrage. Finalement, le domaine d'erreurs initial d'une *constante* est calculé exactement et n'est jamais modifié par le filtrage. Cette erreur s'obtient en évaluant la différence, sur $\mathbb{Q}$, de la valeur exacte de la constante sur $\mathbb{Q}$ et de sa valeur approximée en machine sur $\mathbb{F}$. En plus du domaine

d'erreurs associé à chaque variable, il existe un *domaine de l'erreur sur l'opération* pour chaque contrainte arithmétique. Contrairement aux domaines liés à une variable, ce domaine est directement attaché à une contrainte. Il représente l'erreur (voir chapitre 2) produite à chaque opération dans P par l'application de l'opérateur d'arrondi afin d'obtenir un résultat dans $\mathbb{F}$. Comme pour le domaine d'erreurs sur une variable, il est défini par un intervalle sur $\mathbb{Q}$, comme donné dans la définition 5.2.4.

Définition 5.2.4 (Domaine de l'erreur sur l'opération)**.**

$$D_\odot = [\underline{e}_\odot, \overline{e}_\odot] = \{e_\odot \in \mathbb{Q} \mid \underline{e}_\odot \leq e_\odot \leq \overline{e}_\odot\} \text{ avec } \underline{e}_\odot, \overline{e}_\odot \in \mathbb{Q}$$

L'exemple 5.2.2 illustre les domaines de valeurs et les domaines d'erreurs initiaux pour l'ensemble des valeurs d'un CSP étendu aux erreurs.

Exemple 5.2.2 – Reprenons le programme `carbonGas` donné dans le programme 5.1. Les domaines de valeurs des variables du CSP de 5.1 sont donnés dans 5.2 pour chaque variable du problème.

$$\begin{aligned} D_v &= \left[\frac{1}{10}, \frac{1}{2}\right] & D_p &= 35000000 \\ D_a &= 0{,}401 & D_t &= 300 \\ D_b &= 0{,}000\,042\,7 & D_n &= 1000 \\ D_k &= 1{,}380\,650\,3 \times 10^{-23} \end{aligned} \tag{5.2}$$

Les domaines d'erreurs des variables sont quant à eux donnés dans 5.3 pour chaque variable du problème.

$$\begin{aligned} D_{e_v} &= \left[-\frac{1}{2}\,\mathrm{ulp}(D_v), +\frac{1}{2}\,\mathrm{ulp}(D_v)\right] & D_{e_p} &= 0 \\ D_{e_a} &= \frac{401}{1000} - 0{,}401 & D_{e_t} &= 0 \\ D_{e_b} &= \frac{427}{10000000} - 0{,}000\,042\,7 & D_{e_n} &= 0 \\ D_{e_k} &= \frac{13806503}{10000000000000000000000000000000} - 1{,}380\,650\,3 \times 10^{-23} \end{aligned} \tag{5.3}$$

Pour v, la variable d'entrée du domaine, son domaine d'erreurs correspondant est borné par $\pm\frac{1}{2}$ ulp de son domaine de valeurs. Les variables p, t, et n sont des constantes exactement représentables dans $\mathbb{F}$. Leurs erreurs sont donc nulles. Au contraire, les variables a, b, et k sont des constantes dont la valeur n'est pas dans $\mathbb{F}$. Le domaine d'erreurs pour ces variables s'obtient en faisant la différence entre la valeur sur $\mathbb{Q}$ de la constante et sa valeur approchée en machine sur $\mathbb{F}$. Une telle opération est uniquement possible car ces constantes sont des rationnels.

5.2.2 Contraintes d'un CSP

Une *contrainte* c_i exprime une relation sur un ensemble de variables du problème qui restreint les valeurs que peuvent prendre ces variables ou leurs erreurs. Chaque contrainte est équivalente à une opération du programme P.

La sémantique classique des contraintes, *e.g.*, $z = x + y$, ne permet pas d'exprimer de relation sur le domaine d'erreurs des variables. Nous proposons donc une nouvelle notation, de la forme e_x, pour exprimer le domaine d'erreurs de la variable dans une contrainte tout en conservant la sémantique classique pour les variables. De telles contraintes permettent de raisonner sur les erreurs produites par un programme. Une contrainte peut à la fois exprimer une relation entre le domaine de valeurs et le domaine d'erreurs des variables. Lorsque c'est le cas, sachant que le domaine d'erreurs est un intervalle sur $\mathbb{Q}$, le domaine de valeurs est promu de $\mathbb{F}$ vers $\mathbb{Q}$, pour permettre d'évaluer la contrainte sur $\mathbb{Q}$. L'exemple 5.2.3 présente quelques contraintes sur les erreurs permettant de raisonner sur le comportement d'un programme.

Exemple 5.2.3 – Soit le CSP obtenu à partir de `carbonGas` donné dans le programme 5.1. Ajouter la contrainte $e_v \leftarrow 0$ revient à déclarer que la variable d'entrée v n'a pas d'erreur d'arrondi due à sa représentation machine. Cette contrainte impacte directement le calcul de l'erreur dans les autres contraintes.

Au contraire, une contrainte $w = v + e_v$, permet d'ajouter l'incertitude de représentation, *i.e.,* l'erreur, de la variable v à son domaine de valeurs et ainsi d'obtenir sa représentation exacte sur $\mathbb{Q}$, dans la variable w. Cette contrainte mixte implique le domaine de valeurs D_v et le domaine d'erreurs D_{e_v} d'une variable du problème et est donc évaluée comme une contrainte sur $\mathbb{Q}$.

Les contraintes de comparaison peuvent également être appliquées à un domaine d'erreurs d'une variable. La contrainte $e_r > 0$ force l'erreur sur la variable finale r de `carbonGas` à être strictement supérieure à zéro.

5.2.3 Solution d'un CSP

Une *solution* d'un CSP est une affectation de chaque variable, et de son erreur, à une valeur de son domaine de façon à satisfaire toutes les contraintes du problème. En pratique, le filtrage seul ne permet pas de réduire les domaines des variables à une seule valeur. Il va par contre produire des intervalles réduits pour chaque domaine, où les valeurs inconsistantes sont supprimées. De tels intervalles permettent d'obtenir une borne supérieure correcte pour l'erreur produite par le programme P. Cette borne est en général inatteignable et loin de l'erreur actuelle. L'exemple 5.2.4 illustre la solution obtenue sur un CSP après application du filtrage.

Exemple 5.2.4 – Soit le CSP obtenu à partir de `carbonGas` donné dans le programme 5.1. Le filtrage de ce CSP permet d'obtenir les domaines de valeurs réduits donné dans l'équation 5.4.

$$\begin{aligned} D_v &= [1{,}000\,000\,000\,000\,000\,055\,51e{-}01; 5{,}000\,000\,000\,000\,000\,000\,00e{-}01] \\ D_r &= [2{,}097\,409\,200\,000\,000\,186\,26e{+}06; 3{,}434\,323\,000\,000\,000\,000\,00e{+}07] \end{aligned} \tag{5.4}$$

Les domaines d'erreurs inférés par le filtrage sont donnés dans l'équation 5.5.

$$\begin{aligned} D_{e_v} &= [-2{,}775\,557\,561\,562\,891\,351\,06e{-}17; 5{,}551\,115\,123\,125\,782\,702\,12e{-}17] \\ D_{e_r} &= [-4{,}238\,217\,666\,738\,820\,285\,93e{-}08; 3{,}329\,671\,783\,767\,196\,537\,56e{-}08] \end{aligned} \tag{5.5}$$

Les autres variables du CSP étant des constantes, leurs domaines de valeurs et leurs domaines d'erreurs initiaux ne sont pas modifiés par le filtrage. À partir de ce filtrage, il est possible d'affirmer que l'erreur en sortie sur la variable r de `carbonGas` est toujours plus petite où égale à $4{,}238\,217\,666\,738\,820\,285\,93 \times 10^{-8}$ en valeur absolue.

5.3 Quantifier la déviation du calcul entre $\mathbb{R}$ et $\mathbb{F}$

Les opérations sur les nombres à virgule flottante sont différentes des opérations correspondantes sur les nombres réels. Cette différence provient de l'arrondi appliqué sur chaque opération élémentaire. L'ensemble des nombres à virgule flottante est un sous-ensemble fini des nombres réels le résultat d'une opération sur les flottants n'est pas un nombre à virgule flottante, en général. L'opérateur d'arrondi s'assure que le résultat de chaque opération élémentaire sur $\mathbb{F}$ est arrondi au nombre à virgule flottante le plus proche, en fonction du mode d'arrondi choisi.

Le standard IEEE 754 [IEEE, 2008] définit le comportement de l'arithmétique des nombres flottants. Pour les quatre opérations arithmétiques $(\oplus, \ominus, \otimes, \oslash)$, un arrondi correct (voir définition 2.2.1) est requis : le résultat d'une opération sur $\mathbb{F}$ doit être égal à l'arrondi du résultat de l'opération équivalente sur $\mathbb{R}$. Cette propriété nous donne une borne de $\pm\frac{1}{2}\,\mathrm{ulp}$ sur l'erreur introduite par une opération, avec un mode d'arrondi au plus proche pair. Lorsque le résultat d'une opération sur $\mathbb{F}$ est arrondi, il est différent du résultat attendu sur $\mathbb{R}$. Ceci vient du fait que chaque opération qui appartient à une expression complexe à de fortes chances d'introduire une différence entre le résultat sur $\mathbb{F}$ et celui sur $\mathbb{R}$. Même si pour une opération, le résultat est correctement arrondi, l'accumulation d'arrondis dans une expression peut mener à une déviation forte entre $\mathbb{F}$ et $\mathbb{R}$.

La déviation du calcul sur les nombres à virgule flottante est issue de chaque opération élémentaire. Il est donc possible de reconstruire cette déviation à partir de la composition de ces opérations élémentaires. Une fois cette déviation calculée pour chaque opération élémentaire, l'accumulation de ces erreurs en suivant le flot d'exécution de P donne l'erreur produite en sortie par le programme. Contrairement aux formules classiques de l'erreur absolue (voir définition 2.3.1) et de l'erreur relative (voir définition 2.3.2), l'erreur calculée ici est signée. Conserver le signe de l'erreur permet de prendre en compte la compensation d'erreur qui peut se produire entre plusieurs opérations arithmétiques. La définition 5.3.1 donne la déviation du calcul pour chaque opération arithmétique.

Définition 5.3.1 (Déviation du calcul pour les opérations arithmétiques)**.** Pour une opération arithmétique $\tilde{z} = \tilde{x} \odot \tilde{y}$ la déviation du calcul dans le résultat, noté e_z, est donné par l'équation 5.6.

$$e_z = (x \cdot y) - (\tilde{x} \odot \tilde{y}) \tag{5.6}$$

Les variables x et y, sur $\mathbb{R}$, sont équivalentes à $\tilde{x}$ et $\tilde{y}$, sur $\mathbb{F}$, sans erreurs de représentation. Les opérateurs $\cdot$ et $\odot$ sont respectivement l'opération arithmétique sur $\mathbb{R}$ et son opération correspondante sur $\mathbb{F}$.

En plus de la déviation du calcul pour une opération arithmétique, nous définissons l'erreur sur l'opération. Cette erreur est égale à la différence entre l'évaluation mathématique exacte de l'opération et son exécution machine approximée. Pour un terme d'erreur générique, $e_\odot$, dans une opération $\tilde{z} = \tilde{x} \odot \tilde{y}$, la définition 5.7 donne la formule pour calculer cette erreur sur l'opération.

Définition 5.3.2 (Erreur sur une opération arithmétique)**.** Pour une opération arithmétique $\tilde{z} = \tilde{x} \odot \tilde{y}$, avec $\odot \in \{\oplus, \ominus, \otimes, \oslash\}$, l'erreur sur une opération sur les nombres à virgule flottante, notée $e_\odot$, est définie comme suit :

$$e_\odot = ((\tilde{x} \cdot \tilde{y}) - (\tilde{x} \odot \tilde{y})) \tag{5.7}$$

où $\cdot$ et $\odot$ sont respectivement l'opération arithmétique sur $\mathbb{R}$ et sur $\mathbb{F}$.

Cette définition, proche de la définition 5.3.1 pour la déviation du calcul, ne capture pourtant pas la même chose. Elle s'intéresse uniquement à l'erreur produite par l'opération, *i.e.,* la différence entre l'opération sur $\mathbb{R}$ et sur $\mathbb{F}$ pour les valeurs en machine de $\tilde{x}$ et $\tilde{y}$. Tandis que la définition pour la déviation du calcul capture également les erreurs attachées aux opérandes de l'opération.

À partir de la définition 5.3.1 pour la déviation du calcul, il est possible d'obtenir l'expression de l'erreur sur $\tilde{z}$, le résultat d'une opération arithmétique. Dans la suite, nous montrons comment obtenir cette expression de l'erreur pour une soustraction. Le résultat sur la soustraction s'étend naturellement aux autres opérations arithmétiques.

Soit la soustraction définie par $\tilde{z} = \tilde{x} \ominus \tilde{y}$ avec $\tilde{z}, \tilde{x}, \tilde{y} \in \mathbb{F}$ et un mode d'arrondi au plus proche pair. Les erreurs des opérandes $\tilde{x}$ et $\tilde{y}$, respectivement e_x et e_y sont données dans 5.8 et 5.9. Ces erreurs représentent la différence entre la valeur exacte de l'opérande sur $\mathbb{R}$ et sa valeur approchée en machine sur $\mathbb{F}$.

$$e_x = x - \tilde{x} \tag{5.8}$$
$$e_y = y - \tilde{y} \tag{5.9}$$

L'erreur pour z, dans 5.10, est dépendante des opérandes de la soustraction. Elle s'exprime comme la différence entre l'évaluation de l'opération sur $\mathbb{R}$ et sur $\mathbb{F}$.

$$e_z = (x - y) - (\tilde{x} \ominus \tilde{y}) \tag{5.10}$$

À partir de 5.8 et 5.9 pour les erreurs des opérandes, il est possible de réécrire e_z comme montré dans 5.11.

$$e_z = ((\tilde{x} + e_x) - (\tilde{y} + e_y)) - (\tilde{x} \ominus \tilde{y}) \tag{5.11}$$

5.12 isole chaque terme de l'erreur présent dans 5.11.

$$e_z = e_x - e_y + ((\tilde{x} - \tilde{y}) - (\tilde{x} \ominus \tilde{y})) \tag{5.12}$$

Le terme $((\tilde{x} - \tilde{y}) - (\tilde{x} \ominus \tilde{y}))$ dans 5.12 représente l'erreur produite par l'opération de soustraction de la définition 5.3.2. Elle est par la suite notée $e_\ominus$. 5.13 donne la formule du calcul de la déviation pour l'opération de soustraction $\tilde{z} = \tilde{x} \ominus \tilde{y}$.

$$e_z = e_x - e_y + e_\ominus \tag{5.13}$$

L'évaluation de 5.13 est effectuée entièrement sur $\mathbb{R}$ excepté pour le terme sur $\mathbb{F}$ dans $e_\ominus$.

Les équations 5.14, 5.15, 5.16, et 5.17 expriment respectivement la déviation du calcul pour l'addition, la soustraction, la multiplication, et la division.

$$\tilde{z} = \tilde{x} \oplus \tilde{y} \rightarrow e_z = e_x + e_y + e_\oplus \tag{5.14}$$
$$\tilde{z} = \tilde{x} \ominus \tilde{y} \rightarrow e_z = e_x - e_y + e_\ominus \tag{5.15}$$
$$\tilde{z} = \tilde{x} \otimes \tilde{y} \rightarrow e_z = \tilde{x} \times e_y + \tilde{y} \times e_x + e_x \times e_y + e_\otimes \tag{5.16}$$
$$\tilde{z} = \tilde{x} \oslash \tilde{y} \rightarrow e_z = \frac{\tilde{y} \times e_x - \tilde{x} \times e_y}{\tilde{y} \times (\tilde{y} + e_y)} + e_\oslash \tag{5.17}$$

Ces équations sont évaluées sur $\mathbb{Q}$, car l'erreur liée à un nombre à virgule flottante n'est pas toujours définie dans $\mathbb{F}$. Cela permet de calculer exactement la déviation du calcul pour ces opérations.

Les termes $e_{\oplus}$, $e_{\ominus}$, $e_{\otimes}$, et $e_{\oslash}$ représentent l'erreur d'arrondi introduite par l'opération correspondante, comme donnée dans la définition 5.3.2.

5.3.1 Modèle de l'erreur d'arrondi

Deux modèles existent pour représenter l'erreur d'arrondi sur une opération (voir les définitions 2.3.3 et 2.3.4 du chapitre 2). Ici, nous choisissons le modèle à base d'ulp de la définition 2.3.4. Il a l'avantage d'exprimer directement la distance entre deux nombres à virgule flottante et de ne pas dépendre des bornes ϵ et δ, contrairement au modèle de la définition 2.3.3

Pour ce modèle, nous adaptons la définition de l'ulp (voir définition 2.2.1) aux intervalles. Un intervalle est conservateur des solutions, *i.e.,* par construction les bornes de l'intervalle sont toujours plus grandes ou égales aux valeurs effectivement prisent par l'ulp. Contrairement à la définition classique, pour $\mathrm{ulp}(x)$, x est restreint à $\mathbb{F}$. De plus, l'ulp d'un infini est égal à l'infini. Si x est un NaN, alors son ulp est à $+\infty$ (ou $-\infty$) afin d'obtenir une erreur finale infinie. Ici, $\mathrm{ulp}(0)$ n'est pas égal à 0 mais applique le cas général pour obtenir chaque borne, restant ainsi conservateur des solutions. La définition 5.3.3 décrit l'ulp plus formellement pour les contraintes, où $\mathbf{x}$ est l'intervalle de valeurs que peut prendre une variable x sur $\mathbb{F}$ et L est le plus grand nombre à virgule flottante numérique représentable dans un format donné.

Définition 5.3.3 (ulp pour les contraintes)**.** Soit $\mathbf{x}$ un intervalle à bornes dans $\mathbb{F}$ tel que $\underline{\mathbf{x}} \leq \overline{\mathbf{x}}$. $\mathrm{ulp}(\mathbf{x})$ est, pour la borne supérieure, le maximum entre les différences du successeur de chaque borne de $\mathbf{x}$ et de la valeur de la borne. La borne inférieure s'obtient par symétrie. De plus, si une des bornes de $\mathbf{x}$ est un NaN, alors $\mathrm{ulp}(\mathbf{x})$ est égal à $[-\infty, +\infty]$. L est le plus grand nombre à virgule flottante numérique représentable dans un format donné.

$$\mathrm{ulp}(\mathbf{x}) = \begin{cases} [-\infty, +\infty] & \text{si } \underline{\mathbf{x}} = -\infty \vee \overline{\mathbf{x}} = +\infty \\ [L^- \ominus L, L \ominus L^-] & \text{si } |\underline{\mathbf{x}}| = L \vee |\overline{\mathbf{x}}| = L \\ [\min(\underline{\mathbf{x}}^- \ominus \underline{\mathbf{x}}, \overline{\mathbf{x}}^- \ominus \overline{\mathbf{x}}), \max(\underline{\mathbf{x}}^+ \ominus \underline{\mathbf{x}}, \overline{\mathbf{x}}^+ \ominus \overline{\mathbf{x}})] & \text{sinon} \end{cases} \tag{5.18}$$

Dans certains cas, la définition 5.3.3 de l'ulp sur les intervalles capture une dissymétrie au niveau des bornes. Cette dissymétrie est due à la distribution des nombres à virgule flottante. En effet, La densité des nombres à virgule flottante est plus forte proche de zéro que vers l'infini. De plus, chaque nombre à virgule flottante dans l'intervalle ouvert $]2^u, 2^{u+1}[$, avec $u \in \mathbb{Z}$ est à la même distance de ses voisins, *i.e.,* l'ulp pour tout nombre à virgule flottante dans cet intervalle est le même. Donc, pour une opération $\mathbf{z} = \mathbf{x} \odot \mathbf{y}$ avec $\odot \in \{\oplus, \ominus, \otimes, \oslash\}$ et $\underline{\mathbf{z}} > 0$. Si $\max|\mathbf{z}|$ est un nombre à virgule flottante de la forme 2^u avec $u \in \mathbb{Z}$, alors la borne inférieure de $\mathrm{ulp}(\mathbf{z})$ est égale à la moitié de sa borne supérieure. Cette réduction s'obtient par simple symétrie sur les intervalles négatif. L'exemple 5.3.1 illustre cette dissymétrie pour différents intervalles dans $\mathbb{F}$.

Exemple 5.3.1 – Soit les intervalles $\mathbf{a} = [2; 4]$, $\mathbf{b} = [1{,}5; 2]$, $\mathbf{c} = [-2; 2]$, $\mathbf{d} = [1; 3]$, et $\mathbf{e} = [2; 2]$. L'ulp pour chaque intervalle est donné dans 5.19.

$$\begin{aligned}
\mathrm{ulp}(\mathbf{a}) &= \left[-2{,}220\,446\,049\,250\,313 \times 10^{-16}; 4{,}440\,892\,098\,500\,626 \times 10^{-16}\right] \\
\mathrm{ulp}(\mathbf{b}) &= [-1{,}110\,223\,024\,625\,157 \times 10^{-16}; 2{,}220\,446\,049\,250\,313 \times 10^{-16}] \\
\mathrm{ulp}(\mathbf{c}) &= [-2{,}220\,446\,049\,250\,313 \times 10^{-16}; 2{,}220\,446\,049\,250\,313 \times 10^{-16}] \\
\mathrm{ulp}(\mathbf{d}) &= [-2{,}220\,446\,049\,250\,313 \times 10^{-16}; 2{,}220\,446\,049\,250\,313 \times 10^{-16}] \\
\mathrm{ulp}(\mathbf{e}) &= [-1{,}110\,223\,024\,625\,157 \times 10^{-16}; 2{,}220\,446\,049\,250\,313 \times 10^{-16}]
\end{aligned} \tag{5.19}$$

L'ulp de $\mathbf{c}$ reste symétrique, car l'intervalle contient 0 et ses bornes sont identiques, en valeurs absolues. Pour $\mathbf{d}$, l'ulp est symétrique car aucune des bornes de l'intervalle n'est un nombre à virgule flottante qui s'écrit sous forme de puissance de deux. Il n'est donc pas possible de réduire plus les bornes de l'ulp sans perdre de solution. Au contraire, l'ulp de $\mathbf{a}$ est asymétrique étant donné que l'intervalle contient tous les nombres entre 2^u et 2^{u+1} inclus, pour $u = 1$. De même pour $\mathbf{b}$, où $\overline{\mathbf{b}}$ est égal à 2^1, donc la borne supérieure est sur-approximé avec l'ulp des nombres à virgule flottante de la puissance 2^1, tandis que la borne inférieure prend la moitiée de la borne supérieure. Pour l'intervalle $\mathbf{e}$, qui est un intervalle dégénéré égal à 2, l'ulp se trouve de part et d'autre de 2^1, donc il est possible de réduire la borne inférieure à la moitiée de la borne supérieure.

À partir de cette définition de l'ulp, il est possible de construire un modèle de l'erreur d'arrondi pour une opération arithmétique. Considérons l'opération binaire $\tilde{z} = \tilde{x} \odot \tilde{y}$ prenant deux nombres à virgule flottante $\tilde{x}$ et $\tilde{y}$ en entrée et donnant le nombre à virgule flottante $\tilde{z}$ en sortie, avec $\odot \in \{\oplus, \ominus, \otimes, \oslash\}$. L'erreur sur l'opération est bornée dans le cas général par la définition 5.3.4.

Définition 5.3.4 (Modélisation de l'erreur d'arrondi)**.** Soit une opération binaire $\tilde{z} = \tilde{x} \odot \tilde{y}$ prenant deux nombres à virgule flottante $\tilde{x}$ et $\tilde{y}$ en entrée et donnant le nombre à virgule flottante $\tilde{z}$ en sortie, avec $\odot \in \{\oplus, \ominus, \otimes, \oslash\}$. Soit $\mathbf{x}, \mathbf{y}, \mathbf{z}$ les intervalles représentant les nombres à virgule flottante $\tilde{x}, \tilde{y}, \tilde{z}$ respectivement. Sur les intervalles, l'erreur pour une opération est, dans le cas général, toujours dans l'intervalle $\frac{1}{2}\,\mathrm{ulp}(\mathbf{z})$. Si les intervalles des opérandes, $\mathbf{x}$ et $\mathbf{y}$, sont dégénérés, alors l'erreur sur l'opération est égale à la différence entre l'opération effectuée sur $\mathbb{R}$ et l'opération effectuée sur $\mathbb{F}$. L'équation 5.20 formalise cette borne sur les intervalles et traite le cas des intervalles dégénérés pour $\mathbf{x}$ et $\mathbf{y}$.

$$\mathbf{e}_{\odot} = \begin{cases} [(\underline{\mathbf{x}} \cdot \underline{\mathbf{y}}) - (\underline{\mathbf{x}} \odot \underline{\mathbf{y}}), (\underline{\mathbf{x}} \cdot \underline{\mathbf{y}}) - (\underline{\mathbf{x}} \odot \underline{\mathbf{y}})] & \text{si } \underline{\mathbf{x}} = \overline{\mathbf{x}} \wedge \underline{\mathbf{y}} = \overline{\mathbf{y}} \\ \frac{1}{2}\,\mathrm{ulp}(\mathbf{z}) & \text{sinon} \end{cases} \tag{5.20}$$

La définition 5.3.4 est générale pour les quatre opérations arithmétiques ($\oplus, \ominus, \otimes, \oslash$), mais il existe certains cas particuliers où, en fonction de l'opération, il est possible d'obtenir une meilleure borne de l'erreur.

Pour l'addition et la soustraction, sachant que $\tilde{x} \ominus \tilde{y} \equiv \tilde{x} \oplus (-\tilde{y})$, l'équation 5.21 donne les trois cas particuliers où l'opération est exacte, *i.e.,* l'opération ne produit pas d'erreur d'arrondi. Le premier cas est dû au théorème de Sterbenz [Sterbenz, 1974], tandis que les deux autres cas sont possibles grâce aux théorèmes de Hauser [Hauser, 1996].

$$\mathbf{e}_{\oplus} = \begin{cases} [0, 0] & \text{si } \mathbf{y} \oslash 2 \leq \mathbf{x} \leq 2 \otimes \mathbf{y} \\ [0, 0] & \text{si } \mathrm{rnd}(\mathbf{x} \oplus \mathbf{y}) \text{ est un nombre dénormalisé} \\ [0, 0] & \text{si } |\mathbf{x} \oplus \mathbf{y}| < 2 \otimes u, \text{ avec } u \text{ le plus petit nombre normalisé} \end{cases} \tag{5.21}$$

Une multiplication, $\tilde{z} = \tilde{x} \otimes \tilde{y}$, est exacte lorsque une des deux opérandes est un nombre à virgule flottante représentable exactement sous forme de puissance de deux. Cela est uniquement vrai lorsque le résultat de l'opération est un nombre normalisé. Un nombre dénormalisé n'est pas forcément suffisant pour représenter exactement le résultat de cette opération et peut produire un dépassement de capacité. L'équation 5.20 formalise ce cas.

$$\mathbf{e}_{\otimes} = [0,0] \text{ si } (\underline{\mathbf{x}} = 2^u \wedge \overline{\mathbf{x}} = 2^u) \vee (\underline{\mathbf{y}} = 2^u \wedge \overline{\mathbf{y}} = 2^u) \text{ avec } u \in \mathbb{Z} \text{ et } \mathbf{z} \text{ est un nombre normalisé} \tag{5.22}$$

Comme pour la multiplication, une division, $\tilde{z} = \tilde{x} \oslash \tilde{y}$, est exacte lorsque son résultat est un nombre normalisé et que son dénominateur est un nombre à virgule flottante représentable exactement par une puissance de deux. L'équation 5.23 donne ce cas exact pour la division.

$$\mathbf{e}_{\oslash} = [0,0] \text{ si } \underline{\mathbf{y}} = 2^u \wedge \overline{\mathbf{y}} = 2^u \text{ avec } u \in \mathbb{Z} \text{ et } \mathbf{z} \text{ est un nombre normalisé} \tag{5.23}$$

Les définitions 5.3.3 et 5.3.4 sont écrites sur les intervalles. Les calculs sont effectués sur $\mathbb{Q}$ et appliquent les règles classiques de l'arithmétique des intervalles [Moore *et al.*, 2009]. En plus de ces règles, l'ensemble $\mathbb{Q}$ utilisé pour les intervalles est étendu comme suit, afin de représenter correctement notre modèle de l'erreur, $\mathbb{Q} \cup \{-\infty, +\infty, \mathrm{NaN}\}$.

5.4 Filtrage dédié aux erreurs d'arrondi

Un problème de satisfaction de contraintes, avec des contraintes étendues aux erreurs couplées aux domaines d'erreurs et aux domaines d'erreurs sur l'opération permet de modéliser l'analyse d'erreurs d'un programme sur les nombres à virgule flottante. Une fois cette modélisation du problème obtenue, il est nécessaire d'appliquer un processus de filtrage pour supprimer les valeurs trivialement inconsistantes des domaines des variables.

Notre filtrage applique une 2B-consistance (voir définition 3.1.4 du chapitre 3). Pour les domaines de valeurs, sur les nombres à virgule flottante, cette consistance est issue de [Michel *et al.*, 2001, Michel, 2002, Botella *et al.*, 2006, Marre et Michel, 2010]. L'application de cette consistance nécessite des fonctions de projection, qui raisonnent sur une décomposition des contraintes en contraintes élémentaires, afin de réduire les domaines de valeurs. Ces fonctions de projections ne sont pas adaptées au calcul de l'erreur, et ne permettent donc pas de réduire les domaines d'erreurs des variables.

Nous proposons de nouvelles fonctions de projection, dédiées aux erreurs, afin de mettre en place une 2B-consistance sur les domaines d'erreurs des variables d'un problème. Ces fonctions sont définies à partir des équations 5.14, 5.15, 5.16, et 5.17. Elles expriment la déviation du calcul, pour chaque opération arithmétique, entre son exécution en machine sur $\mathbb{F}$ et son évaluation mathématique exacte sur $\mathbb{R}$. Comme ces formules de la déviation sont écrites sur $\mathbb{R}$, elles s'étendent naturellement aux intervalles. De plus, ces formules permettent d'isoler chaque terme d'erreur et donc de l'inférer à partir des autres erreurs. Par soucis de clarté, $\mathbf{x}$ et $\mathbf{e_x}$ sont respectivement l'intervalle du domaine de valeurs D_x et du domaine d'erreurs D_{e_x} de la variable x. Ces fonctions de projections sont toutes de la forme $\mathbf{e_z} \leftarrow \mathbf{e_z} \cap g(\mathbf{e_x},\mathbf{e_y},\mathbf{e}_{\odot})$ où, pour une erreur $\mathbf{e_z}$, la fonction de projection applique une fonction g prenant en entrée les autres erreurs de la contrainte ($\mathbf{e_x}$, $\mathbf{e_y}$, et $\mathbf{e}_{\odot}$) et donnant en sortie un intervalle d'erreurs sur $\mathbb{Q}$. Une fois cet intervalle obtenu, son intersection avec $\mathbf{e_z}$ produit un nouveau domaine d'erreurs correct pour l'erreur $\mathbf{e_z}$.

Pour une contrainte d'addition de la forme $z = x + y$, les équations 5.24, 5.25, 5.26, et 5.27 donnent les fonctions de projections calculant l'erreur associée à chaque variable de l'opération ainsi que l'erreur sur l'opération.

$$\mathbf{e_z} \leftarrow \mathbf{e_z} \cap (\mathbf{e_x} + \mathbf{e_y} + \mathbf{e}_\oplus) \tag{5.24}$$
$$\mathbf{e_x} \leftarrow \mathbf{e_x} \cap (\mathbf{e_z} - \mathbf{e_y} - \mathbf{e}_\oplus) \tag{5.25}$$
$$\mathbf{e_y} \leftarrow \mathbf{e_y} \cap (\mathbf{e_z} - \mathbf{e_x} - \mathbf{e}_\oplus) \tag{5.26}$$
$$\mathbf{e}_\oplus \leftarrow \mathbf{e}_\oplus \cap (\mathbf{e_z} - \mathbf{e_x} - \mathbf{e_y}) \tag{5.27}$$

Pour une contrainte de soustraction de la forme $z = x - y$, les équations 5.28, 5.29, 5.30, et 5.31 donnent les fonctions de projections calculant l'erreur associée à chaque variable de l'opération ainsi que l'erreur sur l'opération.

$$\mathbf{e_z} \leftarrow \mathbf{e_z} \cap (\mathbf{e_x} - \mathbf{e_y} + \mathbf{e}_\ominus) \tag{5.28}$$
$$\mathbf{e_x} \leftarrow \mathbf{e_x} \cap (\mathbf{e_z} + \mathbf{e_y} - \mathbf{e}_\ominus) \tag{5.29}$$
$$\mathbf{e_y} \leftarrow \mathbf{e_y} \cap (-\mathbf{e_z} + \mathbf{e_x} + \mathbf{e}_\ominus) \tag{5.30}$$
$$\mathbf{e}_\ominus \leftarrow \mathbf{e}_\ominus \cap (\mathbf{e_z} - \mathbf{e_x} + \mathbf{e_y}) \tag{5.31}$$

Pour une contrainte de multiplication de la forme $z = x \times y$, les équations 5.32, 5.33, 5.34, et 5.35 donnent les fonctions de projections calculant l'erreur associée à chaque variable de l'opération ainsi que l'erreur sur l'opération. Contrairement à l'addition et à la soustraction, où les valeurs des opérandes ne sont pas présentes dans les fonctions de projection, les domaines de valeurs de x et y sont utilisés pour calculer les erreurs dans une multiplication. Cela donne deux fonctions de projections supplémentaire pour $\mathbf{x}$ et $\mathbf{y}$, comme montré dans les équations 5.36 et 5.37. Même si ces fonctions calculent de nouveaux domaines de valeurs, leur évaluation est faite sur $\mathbb{Q}$, car elles impliquent également les domaines d'erreurs des variables de la contrainte. Après application de ces deux fonctions, les intervalles, sur $\mathbb{Q}$, obtenus pour x et y sont interséqués avec les domaines de valeurs correspondant, sur $\mathbb{F}$, calculés à partir des fonctions de projections de [Michel *et al.*, 2001, Michel, 2002, Botella *et al.*, 2006, Marre et Michel, 2010].

$$\mathbf{e_z} \leftarrow \mathbf{e_z} \cap (\mathbf{x e_y} + \mathbf{y e_x} + \mathbf{e_x e_y} + \mathbf{e}_\otimes) \tag{5.32}$$
$$\mathbf{e_x} \leftarrow \mathbf{e_x} \cap \left(\frac{\mathbf{e_z} - \mathbf{x e_y} - \mathbf{e}_\otimes}{\mathbf{y} + \mathbf{e_y}}\right) \tag{5.33}$$
$$\mathbf{e_y} \leftarrow \mathbf{y} \cap \left(\frac{\mathbf{e_z} - \mathbf{y e_x} - \mathbf{e}_\otimes}{\mathbf{x} + \mathbf{e_x}}\right) \tag{5.34}$$
$$\mathbf{e}_\otimes \leftarrow \mathbf{e}_\otimes \cap (\mathbf{e_z} - \mathbf{x e_y} - \mathbf{y e_x} - \mathbf{e_x e_y}) \tag{5.35}$$
$$\mathbf{x} \leftarrow \mathbf{x} \cap \left(\frac{\mathbf{e_z} - \mathbf{y e_x} - \mathbf{e_x e_y} - \mathbf{e}_\otimes}{\mathbf{e_y}}\right) \tag{5.36}$$
$$\mathbf{y} \leftarrow \mathbf{y} \cap \left(\frac{\mathbf{e_z} - \mathbf{x e_y} - \mathbf{e_x e_y} - \mathbf{e}_\otimes}{\mathbf{e_x}}\right) \tag{5.37}$$

Pour une contrainte de division de la forme $z = x \div y$, les équations 5.38, 5.39, 5.40, et 5.41 donnent les fonctions de projections calculant l'erreur associée à chaque variable de l'opération ainsi que l'erreur sur l'opération. De la même façon que pour la multiplication, les domaines de valeurs des opérandes x et y sont présents dans les fonctions de projections calculant les erreurs. Cela permet d'écrire les équations 5.42 et 5.43 donnant les fonctions de projections pour les domaines de valeurs des opérandes. Pour plus de lisibilité, la fonction de projection pour $\mathbf{y}$ est décomposée dans les trois équations 5.44, 5.45, et 5.46. Ces équations nécessitent de résoudre une équation quadratique et de calculer une racine carrée. Obtenir une racine carrée exacte sur $\mathbb{Q}$ n'est pas possible dans le cas général. Mais, grâce à un arrondi extérieur des bornes, un intervalle conservatif du résultat de la racine carrée sur $\mathbb{Q}$ est calculé en plongeant l'opération dans $\mathbb{F}$. Cette relaxation de la racine carrée demande toutefois une sur-approximation de la valeur résultante de l'opération obtenue due aux règles de l'arithmétique des intervalles.

$$\mathbf{e_z} \leftarrow \mathbf{e_z} \cap \left(\frac{\mathbf{y e_x} - \mathbf{x e_y}}{\mathbf{y}(\mathbf{y} + \mathbf{e_y})} + \mathbf{e}_\oslash \right) \tag{5.38}$$

$$\mathbf{e_x} \leftarrow \mathbf{e_x} \cap \left((\mathbf{e_z} - \mathbf{e}_\oslash)(\mathbf{y} + \mathbf{e_y}) + \frac{\mathbf{x e_y}}{\mathbf{y}} \right) \tag{5.39}$$

$$\mathbf{e_y} \leftarrow \mathbf{e_y} \cap \left(\frac{\mathbf{e_x} - \mathbf{e_z y} + \mathbf{e}_\oslash \mathbf{y}}{\mathbf{e_z} - \mathbf{e}_\oslash + \frac{\mathbf{x}}{\mathbf{y}}} \right) \tag{5.40}$$

$$\mathbf{e}_\oslash \leftarrow \mathbf{e}_\oslash \cap \left(\mathbf{e_z} - \frac{\mathbf{y e_x} - \mathbf{x e_y}}{\mathbf{y}(\mathbf{y} + \mathbf{e_y})} \right) \tag{5.41}$$

$$\mathbf{x} \leftarrow \mathbf{x} \cap \left(\frac{(\mathbf{e}_\oslash - \mathbf{e_z})\mathbf{y}(\mathbf{y} + \mathbf{e_y}) + \mathbf{y e_x}}{\mathbf{e_y}} \right) \tag{5.42}$$

$$\mathbf{y} \leftarrow \mathbf{y} \cap [\min(\underline{\delta}_\mathbf{1}, \underline{\delta}_\mathbf{2}), \max(\overline{\delta}_\mathbf{1}, \overline{\delta}_\mathbf{2})] \tag{5.43}$$

$$\delta_\mathbf{1} \leftarrow \frac{\mathbf{e_x} - (\mathbf{e_z} - \mathbf{e}_\oslash)\mathbf{e_y} - \sqrt{\mathbf{\Delta}}}{2(\mathbf{e_z} - \mathbf{e}_\oslash)} \tag{5.44}$$

$$\delta_\mathbf{2} \leftarrow \frac{\mathbf{e_x} - (\mathbf{e_z} - \mathbf{e}_\oslash)\mathbf{e_y} + \sqrt{\mathbf{\Delta}}}{2(\mathbf{e_z} - \mathbf{e}_\oslash)} \tag{5.45}$$

$$\mathbf{\Delta} \leftarrow [0, +\infty[\cap ((\mathbf{e_z} - \mathbf{e}_\oslash)\mathbf{e_y} - \mathbf{e_x})^2 + 4(\mathbf{e_z} - \mathbf{e}_\oslash)\mathbf{e_y x} \tag{5.46}$$

Pour une contrainte d'affectation de la forme $z \leftarrow x$, les équations 5.47 et 5.48 donnent les fonctions de projections calculant l'erreur associée à chaque variable de l'opération, uniquement lorsque le programme est exprimé sous *forme statique à affectation unique* [Rosen *et al.*, 1988], ou *static single assignment form (SSA)*. Contrairement aux contraintes arithmétiques, une contrainte d'affectation ne contient pas d'erreur sur l'opération, l'erreur d'une variable est directement transmise à l'autre variable de l'opération.

$$\mathbf{e_z} \leftarrow \mathbf{e_z} \cap \mathbf{e_x} \tag{5.47}$$

$$\mathbf{e_x} \leftarrow \mathbf{e_x} \cap \mathbf{e_z} \tag{5.48}$$

Les fonctions de projection, pour les domaines d'erreurs, sont uniquement définies sur les quatre contraintes arithmétiques ($+$, $-$, $\times$, et $\div$) et la contrainte d'affectation, où l'erreur calculée est simplement transmise à la variable assignée. En effet, l'erreur n'étant pas prise en compte dans les opérateurs de comparaisons, seules les fonctions de projections sur les domaines de valeurs sont nécessaires pour ces contraintes.

Ce filtrage, noté Φ, respecte les propriétés données dans les équations 5.49 à 5.53 où, pour un CSP étendu aux erreurs $\langle X, C, D\rangle$, S est l'ensemble des solutions existantes dans D et $D' \subseteq D$ pour tout sous-ensemble de domaines D'.

$$\Phi(D') \subseteq D' \tag{5.49}$$
$$\Phi(D') \cap S = D' \cap S \tag{5.50}$$
$$\Phi(D') = \varnothing \implies D' \cap S = \varnothing \tag{5.51}$$
$$(\underline{D'} = \overline{D'}) \wedge (D' \cap S = \varnothing) \implies \Phi(D') = \varnothing \tag{5.52}$$
$$(\underline{D'} = \overline{D'}) \wedge (D' \cap S \neq \varnothing) \implies \Phi(D') = D' \tag{5.53}$$

L'équation 5.49 garantie que le filtrage produit uniquement des domaines plus petits ou égaux aux domaines donnés en entrée et l'équation 5.50 assure que les domaines obtenus après filtrage reste conservateurs des solutions. L'équation 5.51 affirme que si le filtrage retourne l'ensemble vide, alors il n'existe pas de solution dans D' satisfaisant les contraintes du problème. Les équations 5.52 et 5.53 sont spécifiques à une instanciation de l'ensemble des domaines de D'. Si cette instanciation est solution du problème, alors l'application du filtrage retourne une instanciation des domaines comme tel, sinon le filtrage retourne l'ensemble vide.

De la même façon que sur le continu ou sur les nombres à virgule flottante, la 2B(w)-consistance est préférée à la 2B-consistance. En pratique, w est fixé à un nombre de valeurs du domaine. La propagation s'arrête donc lorsque les réductions de domaines ne sont pas plus grandes que w. Cette restriction accélère la résolution d'un problème et évite de possibles convergences lentes causées par le filtrage.

5.4.1 Liens entre domaines de valeurs et domaines d'erreurs

En plus du filtrage appliqué lors de la résolution pour réduire les domaines des variables, il est possible de tirer avantage de certaines propriétés entre les domaines de valeurs et les domaines d'erreurs afin de déclencher d'autres réductions. Contrairement aux domaines de valeurs, où les contraintes explicites sur les valeurs des variables impliquent naturellement des réductions de domaines, les relations entre valeurs et erreurs n'existent que pour la multiplication et la division au niveau des contraintes. Nous formalisons deux relations entre domaines de valeurs et domaines d'erreurs qui combinées au filtrage aide à réduire ces domaines.

Une première relation entre ces deux domaines se base sur le standard IEEE 754 [IEEE, 2008] et la notion d'arrondi correct (voir définition 2.2.1 du chapitre 2). Sachant que les quatre opérations arithmétiques sont correctement arrondies au nombre à virgule flottante le plus proche, alors l'équation 5.54 est valide pour $\tilde{x}, \tilde{y} \in \mathbb{F}$.

$$(\tilde{x} \odot \tilde{y}) - \frac{(\tilde{x} \odot \tilde{y}) - (\tilde{x} \odot \tilde{y})^-}{2} \leq (\tilde{x} \cdot \tilde{y}) \leq (\tilde{x} \odot \tilde{y}) + \frac{(\tilde{x} \odot \tilde{y})^+ - (\tilde{x} \odot \tilde{y})}{2} \tag{5.54}$$

Cette équation affirme que le résultat de l'opération sur $\mathbb{F}$ se trouve à un demi ulp près du résultat exact sur $\mathbb{R}$. À partir de cette équation, il est possible de borner l'erreur sur l'opération, comme montré dans l'équation 5.55 où $\tilde{x}, \tilde{y} \in \mathbb{F}$.

$$-\frac{(\tilde{x} \odot \tilde{y}) - (\tilde{x} \odot \tilde{y})^-}{2} \leq e_\odot \leq +\frac{(\tilde{x} \odot \tilde{y})^+ - (\tilde{x} \odot \tilde{y})}{2} \tag{5.55}$$

Les bornes sur cette erreur expriment une relation entre le domaine de valeurs et le domaine d'erreurs sur l'opération : l'erreur sur une opération ne peut jamais être plus grande que le plus grand demi ulp du domaine de valeurs de la variable résultat de l'opération. À partir de cette relation, il est possible d'étendre l'équation 5.55 aux intervalles afin d'améliorer le filtrage des domaines d'erreurs. Pour chaque contrainte arithmétique de la forme $z = x \cdot y$, avec $\cdot \in \{+, -, \times, \div\}$, l'équation 5.56 donne une fonction de projection supplémentaire, étendue aux intervalles, réduisant le domaine de l'erreur sur l'opération pour cette contrainte.

$$\mathbf{e}_\odot \leftarrow \mathbf{e}_\odot \cap \left[-\frac{\min((\underline{\mathbf{z}} - \underline{\mathbf{z}}^-), (\overline{\mathbf{z}} - \overline{\mathbf{z}}^-))}{2}, + \frac{\max((\underline{\mathbf{z}}^+ - \underline{\mathbf{z}}), (\overline{\mathbf{z}}^+ - \overline{\mathbf{z}}))}{2}\right] \tag{5.56}$$

La contraposée de cette propriété offre une autre relation entre domaine de valeurs et domaine d'erreurs. En effet, l'erreur sur une opération est, en valeur absolue, plus petite ou égale au plus grand demi ulp du domaine de valeurs de la variable résultat de l'opération. Pour une contrainte $z = x \cdot y$, avec $\cdot \in \{+, -, \times, \div\}$, si $|\underline{\mathbf{e}}_\odot| > 0$, alors la plus petite valeur de $\mathbf{z}$ peut ne pas être support du résultat. Pour ces petites valeurs, en valeur absolue proche de zéro, si leur demi ulp est plus petit que $|\underline{\mathbf{e}}_\odot|$, alors ces valeurs ne peuvent pas être associées à une erreur sur l'opération assez grande pour être dans le domaine de $\mathbf{e}_\odot$. La proposition 5.4.1 donne une borne inférieure sur le domaine de valeurs de z à partir du domaine d'erreurs de l'erreur sur l'opération.

Proposition 5.4.1. *Soit la contrainte $z = x \cdot y$ avec $\cdot \in \{+, -, \times, \div\}$. Si la borne inférieure du domaine de $|\mathbf{e}_\odot|$ est $1.m \times 2^\epsilon$, alors $\delta = 1.0 \times 2^{\epsilon+p+n}$ est une borne inférieure pour $|\mathbf{z}|$, où p est la longueur de la mantisse m. Si m est à zéro, alors $n = 1$, sinon $n = 2$.*

Cette proposition est valide lorsque la borne inférieure de $|\mathbf{e}_\odot|$ est un nombre normalisé, *c.f.,* la preuve 5.1. Son extension aux nombres dénormalisés est directe.

Preuve 5.1. *Si $m = 0$, l'erreur sur l'opération $\mathbf{e}_\odot$ est bornée comme montré dans l'équation 5.57.*

$$1.0\ldots0 \times 2^\epsilon \leq |\mathbf{e}_\odot| \tag{5.57}$$

Sachant que cette borne est un demi ulp*, il est possible d'expliciter la fraction afin d'obtenir l'*ulp *dans le numérateur, comme donné dans l'équation 5.58.*

$$\frac{1.0\ldots0 \times 2^{\epsilon+1}}{2} \leq |\mathbf{e}_\odot| \tag{5.58}$$

*Dans cette équation, le numérateur de la borne est un nombre normalisé, or l'*ulp *est l'unité en dernière position de la mantisse. L'équation 5.59 déplace le bit à 1 en dernière position de la mantisse, ce qui augmente l'exposant de p, la taille de la mantisse.*

$$\frac{0.0\ldots01 \times 2^{\epsilon+p+1}}{2} \leq |\mathbf{e}_\odot| \tag{5.59}$$

*Le numérateur de la fraction est remplacé par la différence entre les nombres à virgule flottante calculant l'*ulp*, comme montré dans l'équation 5.60. L'*ulp *est égal à $a^+ - a$, où a est un nombre à virgule flottante et a^+ est son successeur.*

$$\frac{1.0\ldots01 \times 2^{\epsilon+p+1} - 1.0\ldots0 \times 2^{\epsilon+p+1}}{2} \leq |\mathbf{e}_\odot| \tag{5.60}$$

Cette équation borne bien $|\mathbf{e}_\odot|$ par le demi ulp de $1.0\ldots0 \times 2^{\epsilon+p+1}$. En remplaçant cette valeur par δ, nous obtenons l'équation 5.61.

$$\frac{\delta^+ - \delta}{2} \leq |\mathbf{e}_\odot| \tag{5.61}$$

Cela borne $|\mathbf{e}_\odot|$ par le demi ulp de $\delta = 1.0 \times 2^{\epsilon+p+n}$ avec $n = 1$, pour $m = 0$. Si $m \neq 0$, alors l'exposant de la borne est décalé de 1, donnant $n = 2$.

*La valeur de l'*ulp *change entre chaque puissance de deux. Sachant que δ est une puissance de deux, alors toute valeur inférieure à δ a un* ulp *plus petit. Ainsi aucune valeur dans $|\mathbf{z}|$, strictement plus petite que δ, n'a de support dans le domaine de $|\mathbf{e}_\odot|$ pour l'erreur sur l'opération. La borne $\delta \leq |\mathbf{z}|$ est donc valide.*

La propriété 5.4.1 réduit le domaine de valeurs d'une variable résultant d'une opération, si et seulement si, le domaine de l'erreur sur l'opération correspondant ne contient pas zéro, *i.e.,* le résultat de l'opération n'est jamais calculé exactement, quelles que soient les valeurs d'entrée. En pratique, ce cas n'apparaît pas souvent. Néanmoins, si dans l'ensemble des contraintes, il existe une contrainte sur l'erreur qui force l'erreur à ne pas être égale à zéro, alors cette borne inférieure produit des réductions de domaines intéressantes. Sinon, le filtrage classique s'applique.

5.5 Contraintes sur les erreurs

Les contraintes sur les erreurs, introduites dans la sous-section 5.2.2, expriment des relations sur les domaines d'erreurs des variables d'un CSP. De telles contraintes permettent de raisonner sur les erreurs et de modéliser, puis résoudre de nouveaux problèmes. En effet, grâce aux contraintes sur les erreurs produites par un programme, il est possible de trouver des valeurs d'entrées exhibant un comportement particulier du programme de manière automatique. L'exemple 5.5.1 illustre ce raisonnement sur un programme qui calcule les racines réelles d'une équation cubique.

Exemple 5.5.1 – Soit la fonction `gsl_poly_solve_cubic` qui calcule les racines réelles de l'expression $x^3 + ax^2 + c = 0$. Cette fonction est directement issue de GSL [Galassi *et al.*, 2009], pour *GNU Scientific Library*, une librairie proposant de nombreux outils et fonctions dédiés au calcul scientifique. La fonction `gsl_poly_solve_cubic` est reprise dans le programme 5.2 écrit en C jusqu'à sa première condition.

Ici, nous nous intéressons à la première condition, *i.e.,* `R == 0 && Q == 0`. En général, tester l'égalité d'une variable à 0 est interdit dans les programmes numériques, sauf si cela permet d'éviter des exceptions comme une division par zéro. Nous cherchons à savoir s'il existe des valeurs d'entrée tel que R et Q sont toutes les deux égales à 0 et calculées sans erreurs. Afin de répondre à cette question, nous prenons $a \in [14, 16]$, $b \in [-200, 200]$, et $c \in [-200, 200]$.

```
int gsl_poly_solve_cubic (double a, double b, double c, double *x0,
double *x1, double *x2)
{
  double q = (a * a - 3 * b);
  double r = (2 * a * a * a - 9 * a * b + 27 * c);

  double Q = q / 9;
  double R = r / 54;

  double Q3 = Q * Q * Q;
  double R2 = R * R;

  double CR2 = 729 * r * r;
  double CQ3 = 2916 * q * q * q;

  if (R == 0 && Q == 0)
  {
    *x0 = - a / 3 ;
    *x1 = - a / 3 ;
    *x2 = - a / 3 ;

    return 3 ;
  }
   ...
}
```

Programme 5.2: gsl_poly_solve_cubic.

Les variables d'entrée sont également considérées sans erreur. Le calcul de R et Q dépend respectivement des variables r et q, qui sont obtenues à partir des variables d'entrées du programme. Le CSP modélisant ce problème est donné dans 5.62.

$$\begin{aligned} & a \in [14, 16], b \in [-200, 200], c \in [-200, 200] \\ & e_a \leftarrow 0, e_b \leftarrow 0, e_c \leftarrow 0 \\ & q = (2 \times a \times a \times a - 9 \times a \times b + 27 \times c) \\ & r = (a \times a - 3 \times b) \\ & Q = \frac{q}{9}, R = \frac{r}{54} \\ & e_Q = 0, e_R = 0 \end{aligned} \tag{5.62}$$

La résolution de ce problème est effectuée à partir du filtrage sur les erreurs introduit dans ce chapitre couplé à la stratégie de recherche **bissection** sur les nombres à virgule flottante (voir chapitre 3). Une solution de ce problème est donnée dans 5.63

$$\begin{aligned} & a = 15, b = 75, c = 125 \\ & e_a = 0, e_b = 0, e_c = 0 \\ & q = 0, r = 0 \\ & e_q = 0, e_r = 0 \\ & Q = 0, R = 0 \\ & e_Q = 0, e_R = 0 \end{aligned} \tag{5.63}$$

Pour cette solution, il est intéressant de noter que les variables q et r du programme sont aussi égales à 0 et que leur erreur est nulle. La condition est donc à la fois satisfaite sur les nombres à virgule flottante et sur les nombres réels.

Une autre question intéressante à se poser et de savoir s'il existe des valeurs d'entrée tel que Q et R sont égales à 0 avec une erreur de calcul. Afin de modéliser ce problème, les contraintes sur les erreurs de Q et R sont remplacées par des inégalités strictement supérieures à zéro. 5.64 reprend le CSP modélisant ce problème.

$$\begin{aligned}
&a \in [14, 16], b \in [-200, 200], c \in [-200, 200] \\
&e_a \leftarrow 0, e_b \leftarrow 0, e_c \leftarrow 0 \\
&q = (2 \times a \times a \times a - 9 \times a \times b + 27 \times c) \\
&r = (a \times a - 3 \times b) \\
&Q = \frac{q}{9}, R = \frac{r}{54} \\
&e_Q > 0, e_R > 0
\end{aligned} \tag{5.64}$$

La résolution de ce problème est effectuée à partir du filtrage sur les erreurs introduit dans ce chapitre et avec la stratégie de recherche **bissection** sur les nombres à virgule flottante. Une solution de ce problème est donnée dans 5.65.

$$\begin{aligned}
&a = 1{,}499\,999\,999\,999\,999\,644\,73 \times 10^{1} \\
&b = 7{,}499\,999\,999\,999\,995\,736\,74 \times 10^{1} \\
&c = 1{,}249\,999\,999\,999\,999\,289\,46 \times 10^{2} \\
&e_a = 0, e_b = 0, e_c = 0 \\
&q = 0, r = 0 \\
&e_q = 2{,}131\,628\,207\,3 \times 10^{-14}, e_r = 1{,}438\,849\,039\,9 \times 10^{-12} \\
&Q = 0, R = 0 \\
&e_Q = 2{,}368\,475\,785\,9 \times 10^{-15}, e_R = 2{,}664\,535\,259\,1 \times 10^{-14}
\end{aligned} \tag{5.65}$$

Cette solution illustre le fait qu'une faible perturbation des valeurs d'entrée d'un programme peut changer un calcul correct en un calcul avec erreurs. Ici, à cause des erreurs d'arrondi, la condition est satisfaite sur les nombres à virgule flottante, alors qu'elle ne l'est pas sur les nombres réels.

5.6 Conclusion

Dans ce chapitre, nous avons introduit notre système de contraintes pour les erreurs d'arrondi. Afin de modéliser un tel problème, nous avons étendu la notion de problème de satisfaction de contraintes aux erreurs. En plus du domaine de valeurs associé à chaque variable d'un problème, nous proposons un nouveau domaine : le domaine d'erreurs. Ce domaine représente l'erreur d'arrondi associée à chaque variable. En plus de ce domaine, un domaine de l'erreur sur l'opération est utilisé afin de modéliser l'erreur d'arrondi produite à chaque opération arithmétique. Pour résoudre un tel problème, nous introduisons un nouveau filtrage, sur les domaines d'erreurs, appliquant une 2B-consistance. Notre calcul de l'erreur tire pleinement avantage des spécificités de l'arithmétique

des nombres à virgule flottante. Ainsi, il est possible d'obtenir une meilleure borne de l'erreur. Nous proposons également de nouvelles contraintes sur les erreurs. Ces contraintes expriment des relations entre domaines d'erreurs, ou domaines d'erreurs et domaines de valeurs, pour différentes variables. Elles permettent de raisonner sur les erreurs produites par un programme. La résolution de ces contraintes se fait dans $\mathbb{Q}$ à l'aide de l'arithmétique des intervalles.

Malgré tout, notre système de contraintes calcule une sur-approximation de l'erreur, *i.e.,* la borne de l'erreur produite n'est pas forcément atteignable et ne se trouve pas toujours proche des valeurs actuelles de l'erreur. Avoir une borne plus proche de l'erreur réellement produite par un programme est intéressant pour mieux capturer le comportement d'un programme. Cela revient à résoudre un problème d'optimisation où les contraintes sur les erreurs sont un élément essentiel afin de modéliser le problème. Ce problème d'optimisation est traité dans le chapitre 6.

CHAPITRE 6

Un algorithme pour encadrer rigoureusement les erreurs d'arrondi

Ce chapitre aborde le problème de maximisation de l'erreur produite par un programme sur les nombres à virgule flottante. Cette erreur maximale, en valeur absolue, représente la plus grande déviation du calcul entre l'exécution en machine sur $\mathbb{F}$ et son équivalent mathématique exact sur $\mathbb{R}$, pour un ensemble de valeurs d'entrée donnée. Obtenir cette erreur maximale est difficile en général. Nous proposons un algorithme pour calculer un encadrement rigoureux de l'erreur maximale. Nous présentons d'abord formellement le problème de maximisation d'erreurs puis nous définisons l'algorithme permettant d'encadrer l'erreur maximale. Enfin, nous donnons quelques cas particuliers afin d'accélérer le temps de résolution de l'algorithme.

Il existe de nombreux outils calculant une borne supérieure de l'erreur que ce soit par interprétation abstraite [Goubault et Putot, 2006, Goubault et Putot, 2011, Moscato *et al.*, 2017, Titolo *et al.*, 2018], optimisation globale [Solovyev *et al.*, 2015, Magron *et al.*, 2017, Solovyev *et al.*, 2018], ou analyse du flot de données d'un programme [Daumas et Melquiond, 2010, Darulova et Kuncak, 2014, Darulova et Kuncak, 2017, Izycheva et Darulova, 2017, Darulova *et al.*, 2018b, Darulova *et al.*, 2018a, Darulova et Volkova, 2019]. Ces outils produisent tous une sur-approximation de cette borne supérieure. Il n'est pas possible de savoir à quelle distance une telle borne se trouve de l'erreur produite en pratique par un programme sur les nombres à virgule flottante. De plus, ces approximations ne capturent pas exactement le comportement réel d'un programme : ces outils peuvent générer des *faux positifs*, *i.e.,* signaler qu'une règle pourrait être violée même si, en pratique, aucune valeur d'entrée ne peut produire ce cas. Connaître l'erreur maximale produite par un programme permettrait d'éviter de produire des faux positifs et donc de mieux capturer le comportement effectif d'un programme. D'autres outils cherchent à calculer une sous-approximation de l'erreur maximale produite par un programme. Ces approches reposent le plus souvent sur du test de programme couplé à une stratégie d'exploration de l'espace de recherche [Chiang *et al.*, 2014, Zou *et al.*, 2020, Xia *et al.*, 2020] ou bien sur une relaxation par optimisation globale [Magron, 2018]. Ces sur-approximations et ces sous-approximations sont des approches complémentaires pour obtenir un encadrement de l'erreur maximale produite par un programme sur les nombres à virgule flottante. Pourtant, aucun outil existant ne permet de calculer à la fois une sur-approximation et une sous-approximation de l'erreur maximale, *i.e.,* un encadrement de l'erreur maximale.

Nous proposons dans ce chapitre un algorithme pour calculer un encadrement de l'erreur maximale d'un programme. Notre algorithme, basé sur un *branch-and-bound*, cherche à maximiser l'erreur produite par un programme et s'intègre directement dans notre système de contraintes pour les erreurs (voir le chapitre 5). À notre connaissance, notre outil est le premier à combiner une sur-approximation et une sous-approximation de l'erreur maximale. Un point clé de l'algorithme est que les deux bornes de l'erreur profitent l'une de l'autre pour s'améliorer. L'algorithme calcule une sur-approximation correcte de l'erreur et produit également des valeurs d'entrée exerçant la sous-approximation, la rendant ainsi atteignable. En général, maximiser une erreur est un processus très coûteux dû à la distribution non uniforme des erreurs. Même sur une unique opération, cette distribution est contraignante et la recherche de valeurs d'entrée exerçant cette erreur revient souvent à une énumération de valeurs [1]. Une combinaison d'opérations aggrave souvent ce comportement, mais, dans certains cas, peut l'améliorer grâce une compensation d'erreurs [2]. L'un des avantages de notre approche est que l'algorithme est *anytime* : il peut être arrêté à la fin de n'importe quelle itération et produit toujours un encadrement correct de l'erreur maximale ainsi que des valeurs d'entrée exerçant sa borne inférieure.

6.1 Définition du problème

La maximisation de l'erreur maximale produite par un programme sur les nombres à virgule flottante peut être vue comme un problème d'optimisation sous contraintes (voir définition 3.0.2 du chapitre 3). L'expression de la fonction objectif d'un tel problème est similaire à celle utilisée par

1. Une énumération de valeurs sur $\mathbb{F}$ n'est en général pas réalisable dans un temps raisonnable à cause de la forte densité de nombres dans les intervalles des domaines à énumérer.

2. Le raisonnement sur des erreurs signées au niveau du filtrage permet de capturer correctement cette compensation d'erreurs.

les approches d'optimisation globale sur-approximant l'erreur (voir la section 4.2 du chapitre 4) et est rappelée dans la définition 6.1.1

Définition 6.1.1 (Maximisation de l'erreur)**.** Soit un programme P sur les nombres à virgule flottante et $\mathbf{X}$ l'ensemble des variables du programme. La fonction $\tilde{f}$ sur $\mathbb{F}$ et son équivalent f sur $\mathbb{R}$ capturent l'ensemble des opérations de P. L'erreur maximale e, en valeur absolue, produite par P, est définie par le problème de maximisation de l'erreur suivant :

$$e = \max_{\mathbf{x}\in\mathbf{X}} |f(\mathbf{x}) - \tilde{f}(\mathbf{x})| \tag{6.1}$$

avec $\mathbf{x}$ un vecteur de variables d'entrée dans l'ensemble des variables $\mathbf{X}$ du programme P.

Trouver l'erreur maximale produite par un programme sur les nombres à virgule flottante est très difficile dans le cas général. La distribution non-uniforme des erreurs rend ce processus très long et nécessite souvent une énumération de valeurs sur les domaines des variables du problème. Nous travaillons donc sur une relaxation de ce problème qui consiste à calculer un encadrement correct de e. Une borne supérieure, notée $\overline{e}$, de l'erreur maximale est équivalente à une sur-approximation classique de l'erreur, tandis qu'une borne inférieure, notée e^*, est une sous-approximation de l'erreur maximale. Ces deux bornes sont exprimées en valeurs absolues, de la même façon que l'erreur maximale. L'équation 6.2 donne l'encadrement de l'erreur maximale.

$$e^* \leq e \leq \overline{e} \tag{6.2}$$

L'exemple 6.1.1 illustre la traduction d'un programme sur les nombres à virgule flottante en problème d'optimisation sous contraintes pour les erreurs et compare l'encadrement obtenu par rapport à d'autres outils de sur-approximation de l'erreur de l'état de l'art.

Exemple 6.1.1 – Soit `predatorPrey`, un programme issu de la suite FPBench [Damouche *et al.*, 2017b], reprit dans le programme 6.1 écrit en C où la variable d'entrée x prend ses valeurs dans l'intervalle $[\frac{1}{10}, \frac{3}{10}]$.

```
double predatorPrey(double x)
{
  double r =  4;
  double K = 111/100;
  return (r*x*x) / (1 + (x/K)*(x/K));
 }
```

Programme 6.1: predatorPrey.

La traduction de ce programme en CSP étendu aux erreurs est directe :

$$\begin{aligned} x &\in \left[\frac{1}{10}, \frac{3}{10}\right] \\ r &\leftarrow 4 \\ K &\leftarrow 1{,}11 \\ z &= \frac{r \times x \times x}{1 + \frac{x}{K} \times \frac{x}{K}} \end{aligned} \tag{6.3}$$

Ce problème de satisfaction de contraintes se transforme facilement en problème d'optimisation sous contraintes grâce à l'ajout d'une fonction objectif. La fonction objectif pour ce problème est donnée par l'équation 6.4.

$$\max|e_z| \tag{6.4}$$

La résolution de ce problème via notre algorithme donne un encadrement de l'erreur e_z, tel que $e^* \leq |e_z| \leq \overline{e}$. Les valeurs des deux bornes sont :

$$\begin{aligned} e^* &= 1{,}433\,909\,753\,866\,319\,175\,47 \times 10^{-16} \\ \overline{e} &= 1{,}670\,620\,154\,466\,705\,020\,70 \times 10^{-16} \end{aligned} \tag{6.5}$$

Ces deux bornes sont en valeurs absolues et les valeurs des variables exerçant e^* sont données dans 6.6.

$$\begin{aligned} x &= 2{,}903\,654\,689\,036\,342\,939\,62 \times 10^{-1} & e_x &= -2{,}775\,557\,561\,562\,891\,351\,06 \times 10^{-17} \\ z &= 3{,}156\,487\,084\,523\,316\,166\,70 \times 10^{-1} & e_z &= -1{,}433\,909\,753\,866\,319\,175\,47 \times 10^{-16} \end{aligned} \tag{6.6}$$

L'atteignabilié de la borne inférieure peut être vérifiée dans un oracle externe en s'assurant que les valeurs données dans 6.6 produisent la sortie attendue.

La table 6.1 donne les sur-approximations de l'erreur calculées par les autres outils de l'état de l'art.

Table 6.1 – Sur-approximations de l'erreur pour `predatorPrey`

Fluctuat	Gappa	PRECiSA	Real2Float	Daisy	Rosa	FPTaylor	FErA
2,36e-16	**1,68e-16**	2,09e-16	**2,52e-16**	1,75e-16	1,98e-16	**1,59e-16**	**1,68e-16**

La meilleure sur-approximation est en **gras et vert**, la seconde meilleure est en **gras**, et la pire est en **gras et rose**. Notre algorithme se classe second et a une meilleure sur-approximation que les autres outils, excepté pour FPTaylor.

De la même façon que pour notre système de contraintes sur les erreurs (voir chapitre 5), notre algorithme pour encadrer rigoureusement l'erreur maximale s'intéresse aux opérations arithmétiques ($\oplus, \ominus, \otimes, \oslash$). Cette restriction permet de calculer exactement l'erreur sur $\mathbb{Q}$. L'exactitude de l'erreur est d'autant plus importante pour notre sous-approximation de l'erreur maximale qui doit être une erreur atteignable. De plus, le standard IEEE 754 [IEEE, 2008] définit plusieurs règles et cas particuliers pour ces opérations, tel que l'arrondi correct obligatoire ou l'opération exacte.

6.2 Algorithme pour encadrer rigoureusement l'erreur maximale d'un programme

L'algorithme 6.2 se base sur un branch-and-bound maximisant l'erreur absolue d'une variable d'un problème. Une telle erreur représente l'erreur maximale, *i.e.,* la plus grande déviation possible

entre le résultat mathématique exact sur $\mathbb{R}$ et le résultat obtenu en machine sur $\mathbb{F}$. Notre algorithme peut facilement être modifié afin de maximiser, ou minimiser, une erreur signée.

Il prend en entrée un CSP étendu aux erreurs, $\langle X, C, D\rangle$ et e l'erreur à maximiser. Cette erreur résulte d'opérations sur les nombres à virgule flottante lors de l'exécution du programme. L'algorithme calcule deux bornes : une borne inférieure e^*, la plus grande erreur atteignable calculée à un moment donné, et une borne supérieure $\overline{e}$, une sur-approximation correcte de e. Comme montré dans l'équation 6.2 ces deux bornes produisent un encadrement de l'erreur maximale e et s'expriment en valeurs absolues. L'originalité de la borne inférieure est qu'il s'agit d'une erreur atteignable, *i.e.,* les valeurs d'entrée exerçant cette erreur sont connues. L'ensemble ordonné S (voir équation 6.7) contient les couples (e,s) tels que e est une erreur et s les valeurs d'entrée qui permettent de produire e.

$$S = \{(e_i, s_i) \mid \forall (e_i, s_i),\, e_i \in \mathbb{Q} \wedge s_i \subset D\} \tag{6.7}$$

En sortie l'algorithme retourne le triplet $(e^*, \overline{e}, S)$ de façon à obtenir un encadrement de l'erreur maximale, mais aussi connaître les valeurs d'entrée exerçant sa borne inférieure. S permet également de suivre l'évolution de la borne inférieure lors de l'exécution de l'algorithme. Dans un branch-and-bound, l'espace de recherche est vu sous forme d'arbre, où la racine correspond à l'état initial des domaines des variables. Cet algorithme de branch-and-bound alterne entre deux étapes : séparation (*branching*) et évaluation (*bounding*). La séparation découpe l'espace de recherche en sous-problèmes de façon arborescente et guide l'exploration dans l'arbre. L'évaluation consiste, pour chaque sous-problème à essayer d'améliorer les bornes inférieures et supérieures du problème. Par la suite, nous appelons un sous-problème une boîte (voir définition 6.2.1), notée $\mathbb{B}$. Une boîte est le produit cartésien des domaines des variables du problème. Une boîte est dite *dégénérée* lorsque l'ensemble des domaines de valeurs des variables du problème sont réduits à une seule valeur.

Définition 6.2.1 (Boîte)**.** Soit un CSP étendu aux erreurs $\langle X, C, D\rangle$. Une *boîte*, notée $\mathbb{B}$, est le produit cartésien, pour chaque variable x dans X, du couple d'ensembles représentant un sous-ensemble de son domaine de valeurs D_x et un sous-ensemble de son domaine d'erreurs D_{e_x}. L'équation 6.8 formalise cette définition.

$$\mathbb{B} = \left\{ \prod_{x \in X} I_x \;\middle|\; I_x = (d_x, e_x),\, d_x \subseteq D_x,\, e_x \subseteq D_{e_x} \right\} \bigcup \varnothing \tag{6.8}$$

Afin de simplifier la notation, une boîte $\mathbb{B}$ peut être utilisée en exposant, *e.g.,* $x^{\mathbb{B}}$ indique que l'élément x se trouve dans la boîte $\mathbb{B}$. L est l'ensemble des boîtes qu'il reste à traiter.

6.2.1 Propriétés et limites

L'objectif principal de ce branch-and-bound est de calculer l'erreur maximale produite par un programme. Cette maximisation est atteinte lorsque la borne inférieure est égale à la borne supérieure, *i.e.,* $e^* = \overline{e}$. Cependant, cette condition est difficile à satisfaire en pratique.

Une première limite vient du *problème de dépendance* [Moore *et al.*, 2009] qui apparaît dans une expression avec plusieurs occurrences d'au moins une variable. Les occurrences multiples sont un problème critique en arithmétique des intervalles étant donné que chaque occurrence d'une variable est considérée comme une variable différente ayant le même domaine. Ce problème de

Entrée : $\langle X,C,D\rangle$ ** triplet de variables, contraintes, et domaines **
Entrée : $e \in [-\infty,+\infty]$ ** erreur à maximiser **
Sortie : $(e^*,\overline{e},S)$ ** bornes de l'erreur maximale et pile des solutions pour la borne inférieure **

$L \leftarrow \left\{ \prod_{x\in X} I_x \;\middle|\; I_x = (D_x,D_{e_x}) \right\}$ ** ensemble des boîtes à traiter **
$\overline{e} \leftarrow +\infty$ ** borne supérieure de l'erreur **
$e^* \leftarrow -\infty$ ** borne inférieure de l'erreur maximale **
$\overline{e}^D \leftarrow -\infty$ ** borne supérieure des boîtes écartées **
$S \leftarrow \varnothing$ ** pile des solutions pour la borne inférieure **
tant que $L \neq \varnothing$ **et** $e^* < \overline{e}$ **faire**
  choisir $\mathbb{B} \in L$ **et** $L \leftarrow L \setminus \mathbb{B}$
  $\overline{e}^{\mathbb{B}}_{old} \leftarrow \overline{e}^{\mathbb{B}}$
  $\mathbb{B} \leftarrow \Phi(X, C \wedge e > e^*, \mathbb{B})$
  si $\overline{e}^{\mathbb{B}}_{old} = \overline{e}$ **et** $(\mathbb{B} = \varnothing$ **ou** $\overline{e}^{\mathbb{B}} < \overline{e})$ **alors**
    si $\mathbb{B} \neq \varnothing$ **alors**
      $\overline{e} \leftarrow \overline{e}^{\mathbb{B}}$
    sinon
      $\overline{e} \leftarrow -\infty$
    fin si
    $\overline{e} \leftarrow \max\left(\{\overline{e}^{\mathbb{B}_i} \mid \forall \mathbb{B}_i \in L\} \cup \{\overline{e}, \overline{e}^D\}\right)$
  fin si
  si $\mathbb{B} \neq \varnothing$ **alors**
    si $\overline{e}^{\mathbb{B}} > e^*$ **alors**
      si $\forall (x^{\mathbb{B}},e_x^{\mathbb{B}}) \in \mathbb{B},\ \underline{x}^{\mathbb{B}} = \overline{x}^{\mathbb{B}}$ **alors**
        $(e^{\mathbb{B}}, s^{\mathbb{B}}) \leftarrow (e^{\mathbb{B}}, \mathbb{B})$
      sinon
        $(e^{\mathbb{B}}, s^{\mathbb{B}}) \leftarrow \text{calculerBorneInférieure}(X,C,\mathbb{B})$
      fin si
      si $e^{\mathbb{B}} > e^*$ **alors**
        $e^* \leftarrow e^{\mathbb{B}}$
        ajouter $(e^{\mathbb{B}}, s^{\mathbb{B}})$ **à** S
        $L \leftarrow L \setminus \{\mathbb{B}_i \in L \mid \overline{e}^{\mathbb{B}_i} \leq e^*\}$
      fin si
    fin si
    si $\overline{e}^{\mathbb{B}} > \overline{e}^D$ **et** $\overline{e}^{\mathbb{B}} > e^*$ **alors**
      si $\mathbb{B}$ **est écartée alors**
        $\overline{e}^D \leftarrow \max(\overline{e}^D, \overline{e}^{\mathbb{B}})$
      sinon si il existe $(\mathbf{x}^{\mathbb{B}}, \mathbf{e_x}^{\mathbb{B}}) \in \mathbb{B}$ **tel que** $\underline{\mathbf{x}}^{\mathbb{B}} < \overline{\mathbf{x}}^{\mathbb{B}}$ **est vrai alors**
        $\mathbb{B}_1 \leftarrow \mathbb{B}$ **et** $\mathbb{B}_2 \leftarrow \mathbb{B}$
        $D_x^{\mathbb{B}_1} \leftarrow \left[\underline{\mathbf{x}}^{\mathbb{B}}, \frac{\underline{\mathbf{x}}^{\mathbb{B}}+\overline{\mathbf{x}}^{\mathbb{B}}}{2}\right]$ **et** $D_x^{\mathbb{B}_2} \leftarrow \left[\left(\frac{\underline{\mathbf{x}}^{\mathbb{B}}+\overline{\mathbf{x}}^{\mathbb{B}}}{2}\right)^+, \overline{\mathbf{x}}^{\mathbb{B}}\right]$
        $L \leftarrow L \cup \{\mathbb{B}_1, \mathbb{B}_2\}$
      fin si
    fin si
  fin si
fin tant que
retourner $(e^*,\overline{e},S)$

Algorithme 6.2: Branch-and-Bound — encadrement de l'erreur maximale

dépendance produit une sur-approximation dans l'évaluation des valeurs possibles qu'une expression peut prendre. L'exemple 6.2.1 illustre ce problème sur une opération arithmétique.

Exemple 6.2.1 – Soit deux intervalles $\mathbf{x}$ et $\mathbf{y}$ tels que $\mathbf{y} = \mathbf{x} \times \mathbf{x}$ avec $\mathbf{x} \in [-1, 1]$. L'évaluation sur les intervalles donne $\mathbf{y} \in [-1, 1]$, alors que le résultat exact est $[0, 1]$. L'équation 6.9 donne la formule du produit [Moore *et al.*, 2009] de deux intervalles $\mathbf{a}$ et $\mathbf{b}$.

$$\mathbf{a} \times \mathbf{b} = [\min(\underline{\mathbf{a}} \times \underline{\mathbf{b}}, \underline{\mathbf{a}} \times \overline{\mathbf{b}}, \overline{\mathbf{a}} \times \underline{\mathbf{b}}, \overline{\mathbf{a}} \times \overline{\mathbf{b}}), \max(\underline{\mathbf{a}} \times \underline{\mathbf{b}}, \underline{\mathbf{a}} \times \overline{\mathbf{b}}, \overline{\mathbf{a}} \times \underline{\mathbf{b}}, \overline{\mathbf{a}} \times \overline{\mathbf{b}})] \tag{6.9}$$

Cette surestimation d'intervalles survient dans les fonctions de projections calculant les erreurs qui contiennent plusieurs occurrences, telles que pour la multiplication (voir équations 5.32 à 5.37) et pour la division (voir équations 5.38 à 5.46). Elle mène à une sur-approximation inutile des intervalles calculés. Une conséquence directe de ce problème est une sur-approximation de la borne supérieure, la rendant inatteignable.

Le second problème est causé par le modèle de l'erreur sur l'opération (voir équation 5.55). Pour une opération arithmétique sur les nombres à virgule flottante de la forme $\tilde{z} = \tilde{x} \odot \tilde{y}$, avec $\odot \in \{\oplus, \ominus, \otimes, \oslash\}$, l'erreur sur l'opération est bornée par $\frac{1}{2}\,\mathrm{ulp}(\tilde{z})$. Cette borne est fortement dépendante de la distribution des nombres à virgule flottante. Soit l'intervalle ouvert $]2^n, 2^{n+1}[$ avec $n \in \mathbb{Z}$, chaque nombre à virgule flottante dans l'intervalle est séparé de ses voisins par la même distance. En d'autres termes, chaque nombre à virgule flottante dans cet intervalle a le même ulp. Lorsque le domaine de valeurs du résultat d'une opération est réduit à un tel intervalle, les bornes de $\mathbf{e}_\odot$ sont fixées et ne peuvent plus être améliorées à partir des fonctions de projections du filtrage. Cette limite peut être généralisée au niveau d'un CSP. La proposition 6.2.1 formalise cette limite.

Proposition 6.2.1 (Borne irréductible pour l'erreur sur l'opération)**.** *Soit le CSP étendu aux erreurs $\langle X, C, D\rangle$ sans occurrences multiples de variables. Si pour chaque contrainte arithmétique de la forme $z = x \cdot y$, avec $\cdot \in \{+, -, \times, \div\}$ et $z, x, y \in X$, le domaine $\mathbf{z}$ est réduit à un intervalle de la forme $]2^n, 2^{n+1}[$ avec $n \in \mathbb{Z}$, fixant ainsi son domaine de l'erreur sur l'opération $\mathbf{e}_\odot$ au demi* ulp *de $\mathbf{z}$, alors aucun domaine du CSP ne peut être réduit sans énumérer ses valeurs.*

Autrement dit, même s'il n'y a pas d'occurrences multiples, alors la borne supérieure de l'erreur à maximiser ne peut plus être réduite à ce moment. C'est pourquoi l'algorithme arrête d'explorer une boîte lorsque la proposition 6.2.1 est vrai. À noter que cette réduction est possible, si et seulement si les intervalles ne sont pas dégénérés. Sinon il est possible d'inférer directement l'erreur à partir de l'unique valeur des domaines de valeurs des variables. Le cas des intervalles dégénérés correspond donc à un processus d'énumération.

6.2.2 Gestion des boîtes

L'algorithme gère une liste L de boîtes à traiter qui est initialisée avec la boîte $\mathbb{B} = \{\prod_{x \in X} I_x \mid I_x = (D_x, D_{e_x})\}$, où D_x et D_{e_x} sont respectivement le domaine de valeurs et le domaine d'erreurs d'une variable x du problème. Une boîte peut être dans trois états différents : *non explorée*, *éliminée*, ou *écartée*. Une boîte *non explorée* est une boîte dans L qui n'a pas encore été traitée. Une boîte *éliminée* est une boîte où la borne supérieure locale de l'erreur $\overline{e}^{\mathbb{B}}$ est plus petite que la borne inférieure de l'erreur e^*, telle que $\overline{e}^{\mathbb{B}} \leq e^*$. Une telle boîte ne contient aucune valeur permettant de calculer une meilleure borne inférieure e^* et est donc supprimée de L. Une boîte

écartée est une boîte qui satisfait la proposition 6.2.1. Cette boîte ne peut pas servir à améliorer l'erreur à moins que l'algorithme ait recours à l'énumération de valeurs pour les domaines de valeurs des variables [3]. Elle est donc enlevée de L. Une boîte *écartée* reste une boîte valide, *i.e.,* une boîte pouvant contenir une solution au problème. C'est pourquoi, en plus de la borne supérieure de l'erreur $\overline{e}$ et de la borne inférieure de l'erreur e^*, l'algorithme prend en compte la borne supérieure de l'erreur pour les boîtes *écartées*. Cette borne pour les boîtes *écartées* est notée $\overline{e}^D$ et rentre en compte lors de la mise à jour de la borne supérieure de l'erreur $\overline{e}$. La résolution s'arrête lorsqu'il ne reste plus de boîte à traiter dans L ou lorsque la borne inférieure de l'erreur e^* est égale à la borne supérieure de l'erreur $\overline{e}$, *i.e.,* lorsque l'erreur maximale est trouvée.

La boucle principale de notre algorithme est comme tout branch-and-bound découpée en plusieurs étapes : sélection d'une boîte, filtrage, mise à jour de la borne supérieure, mise à jour de la borne inférieure, et découpage de la boîte courante. Les lignes de l'algorithme 6.2 correspondantes à chaque étape sont données au début du paragraphe de l'étape.

Sélection d'une boîte (*ligne* 7) La boîte $\mathbb{B}$ sélectionnée dans la liste L est celle avec la plus grande borne supérieure locale de l'erreur. En effet, ce choix offre de plus grandes opportunités d'améliorer e^* et $\overline{e}$. Sachant que la valeur de $\overline{e}$ a son support dans cette boîte, il existe de meilleures chances de calculer une plus grande erreur atteignable e^*. Une fois la boîte choisie, elle est supprimée de L.

Filtrage (*lignes* 8 *et* 9) Le processus de filtrage (voir chapitre 5), noté Φ, est appliqué à $\mathbb{B}$ afin de réduire les domaines de valeurs et les domaines d'erreurs des variables du problème. Ce filtrage est mis en place sur C, l'ensemble des contraintes initiales du problème, auquel sont ajoutées les contraintes $e^* \leq e$ et $e \leq \overline{e}$, explicitant les bornes de l'erreur à maximiser. Si $\Phi(\mathbb{B}) = \varnothing$, alors la boîte ne contient aucune solution ; soit parce qu'elle ne satisfait pas une des contraintes initiales ou car l'intervalle inféré pour l'erreur ne respecte pas les contraintes additionnelles sur les bornes de l'erreur à maximiser. Dans les deux cas, l'algorithme jette la boîte $\mathbb{B}$ et passe directement à la prochaine itération de la boucle principale. Sinon, l'algorithme passe à la prochaine étape.

Mise à jour de la borne supérieure (*lignes* 10 *à* 17) Une fois les domaines dans $\mathbb{B}$ réduits, si la borne supérieure locale de l'erreur dans la boîte courante, notée $e^{\mathbb{B}}$, était support de $\overline{e}$ et ne l'est plus, alors $\overline{e}$ est mis à jour. La borne supérieure de l'erreur à maximiser, $\overline{e}$, est mise à jour en prenant le maximum entre les bornes supérieures locales de l'erreur dans la boîte courante, dans les boîtes restantes à traiter dans L, et dans l'ensemble des boîtes *écartées*.

Mise à jour de la borne inférieure (*lignes* 19 *à* 30) Une boîte non vide peut contenir une plus grande borne inférieure locale que la borne courante. Notre algorithme utilise une procédure, basée sur le principe du *générer-et-tester*, ou *Generate-and-Test*, et nommée calculerBorneInférieure. Cette procédure cherche à obtenir une meilleure borne inférieure en deux étapes. Une variable d'entrée est d'abord instanciée avec un nombre à virgule flottante choisi de manière aléatoire dans son domaine de valeurs grâce à la fonction générationAléatoire (voir ligne 20). Une autre valeur aléatoire est choisie dans son domaine d'erreurs pour fixer l'erreur associée à la variable. Afin de réduire la combinatoire de valeurs possibles, le choix de la valeur de l'erreur tire avantage du fait que, pour une valeur donnée de la variable d'entrée, si le signe de la dérivée ne change pas

3. À condition qu'il n'y ait pas d'occurrences multiples de variable dans l'expression.

sur le domaine d'erreurs, alors la distance maximale entre l'hyperplan défini par le résultat sur $\mathbb{F}$ et son équivalent sur $\mathbb{R}$ se trouve à une des extrémités du domaine d'erreurs correspondant. En pratique, cela limite l'erreur d'une variable d'entrée à $\pm\frac{1}{2}$ ulp de sa valeur choisie. Lorsque toutes les variables d'entrée sont instanciées, une fonction f représentant l'ensemble des opérations du programme est évaluée exactement sur $\mathbb{Q}$, tandis que son équivalent $\tilde{f}$ est évaluée en précision machine sur $\mathbb{F}$. Obtenir l'erreur produite par ces valeurs d'entrées revient simplement à évaluer $f - \tilde{f}$, en valeur absolue, exactement sur $\mathbb{Q}$. L'exactitude de cette erreur est garantie par le fait que les opérations acceptées par notre algorithme sont les quatre opérations arithmétiques $\oplus$, $\ominus$, $\otimes$, et $\oslash$. Ces opérations sont calculables exactement dans $\mathbb{Q}$ et le standard IEEE 754 [IEEE, 2008] donne des propriétés sur les erreurs produites par ces opérations. Ajouter le support pour d'autres opérations sur les nombres à virgule flottante introduirait une approximation nécessaire au calcul des erreurs.

L'erreur calculée est ensuite améliorée à l'aide d'une *recherche locale* [Martí *et al.*, 2018], ou *local search*. Cette méthode consiste à explorer les nombres à virgule flottante proches des valeurs choisies pour les variables d'entrée du programme, puis à évaluer de nouveau les fonctions f et $\tilde{f}$. Cette exploration est réalisée par la fonction perturber à la ligne 20. Ce processus est répété un nombre fixe de fois, jusqu'à ce que l'erreur ne puisse plus être améliorée, *i.e.,* lorsqu'un maximum local de l'erreur est atteint. Si l'erreur obtenue est meilleure que la borne inférieure courante, alors cette dernière est mise à jour. Chaque nouvelle borne inférieure atteignable est ajoutée à l'ensemble S avec l'ensemble des valeurs d'entrée qui l'exerce. L'algorithme 6.3 reprend cette procédure. En pratique, les nombres d'itérations I (voir ligne 4) et J (voir ligne 19) garantissant la terminaison de la procédure *générer-et-tester* et de la *recherche locale* respectivement sont fixés au lancement du branch-and-bound. Le nombre d'itérations pour cette étape de l'algorithme impacte directement le temps de résolution pour un problème, mais offre toutefois de plus grandes chances de calculer une meilleure borne inférieure e^*.

Découpage de la boîte courante (*lignes 31 à 39*) Une boîte n'est pas découpée, mais éliminée, si sa borne supérieure locale de l'erreur est plus petite où égale à $\overline{e}^D$, la borne supérieure des boîtes *écartées*, ou e^*. Supprimer une telle boîte accélère le temps de résolution. Comme aucune valeur de l'erreur contenue dans cette boîte ne peut améliorer la borne supérieure de l'erreur, explorer cette boîte n'est pas nécessaire. Le découpage de la boîte courante ne se produit que s'il existe au moins une variable dans $\mathbb{B}$ qui n'est pas instanciée, *i.e.,* son domaine de valeurs n'est pas réduit à une seule valeur, et que la boîte n'est pas *écartée*. Autrement, la boîte est *écartée*. La borne supérieure locale de l'erreur $e^{\mathbb{B}}$ pour la boîte écartée est utilisée pour mettre à jour la borne supérieure de l'erreur sur l'ensemble des boîtes écartées, si et seulement si $e^{\mathbb{B}}$ est plus grand que $\overline{e}^D$. La stratégie de recherche (voir la section 3.2 du chapitre 3) applique un **round-robin** pour sélectionner la prochaine variable à découper, assurant d'explorer le domaine de chaque variable de façon identique. Le domaine de la variable sélectionnée est ensuite découpé avec une **bissection**, génèrant deux sous-intervalles à partir de l'intervalle initial du domaine de valeurs de la variable. Les deux sous-boîtes produites sont ensuite ajoutées à la liste L des boîtes à traiter.

L'algorithme donne toujours un encadrement correct de l'erreur maximale et sa terminaison est garantie. En effet, dans le pire des cas, toutes les boîtes sont découpées jusqu'à ce qu'il ne reste que des boîtes *dégénérées*. Chaque boîte *dégénérée* avec une erreur $e^{\mathbb{B}}$ plus petite que la borne inférieure de l'erreur e^* est alors supprimée. Si $e^* \leq e^{\mathbb{B}} \leq \overline{e}$ est vrai, alors $e^{\mathbb{B}}$ est utilisé pour mettre à jour e^* et $\overline{e}$ avant de supprimer la boîte. Finalement, comme l'ensemble des nombres à

Entrée : $\langle X,C,\mathbb{B}\rangle$ ** triplet de variables, contraintes, et domaines **
Sortie : $(e^{\mathbb{B}}, s^{\mathbb{B}})$ ** borne inférieure obtenue et valeurs d'entrée l'exerçant **
1: $e^{\mathbb{B}} \leftarrow -\infty$ ** erreur atteignable calculée dans $\mathbb{B}$ **
2: $s^{\mathbb{B}} \leftarrow \varnothing$ ** ensemble des valeurs et erreurs exerçant $e^{\mathbb{B}}$ dans $\mathbb{B}$ **
3: $i \leftarrow 0$
4: **tant que** $i < I$ **faire**
5: $\quad\mathbb{B}' \leftarrow \mathbb{B}$
6: $\quad$**pour chaque** $(\mathbf{x}^{\mathbb{B}'},\mathbf{e_x}^{\mathbb{B}'}) \in \mathbb{B}'$ **faire**
7: $\quad\quad$**si** $\underline{x}^{\mathbb{B}'} < \overline{x}^{\mathbb{B}'}$ **alors**
8: $\quad\quad\quad\mathbf{x}^{\mathbb{B}'} \leftarrow \text{générationAléatoire}(\underline{x}^{\mathbb{B}'}, \overline{x}^{\mathbb{B}'})$
9: $\quad\quad\quad\mathbf{e_x}^{\mathbb{B}'} \leftarrow \text{prendreMeilleureErreur}(\underline{\mathbf{e_x}}^{\mathbb{B}'}, \overline{\mathbf{e_x}}^{\mathbb{B}'})$
10: $\quad\quad\quad\mathbb{B}' \leftarrow \Phi(X, C \wedge e > e^*, \mathbb{B}')$
11: $\quad\quad$**fin si**
12: $\quad$**fin pour**
13: $\quad$**si** $\mathbb{B}' \neq \varnothing$ **et** $\overline{e}^{\mathbb{B}'} > e^{\mathbb{B}}$ **alors**
14: $\quad\quad e^{\mathbb{B}} \leftarrow \overline{e}^{\mathbb{B}'}$
15: $\quad\quad s^{\mathbb{B}} \leftarrow \mathbb{B}'$
16: $\quad$**fin si**
17: $\quad i \leftarrow i + 1$
18: $\quad j \leftarrow 0$
19: $\quad$**tant que** $e^{\mathbb{B}} > -\infty$ **et** $j < J$ **faire**
20: $\quad\quad\mathbb{B}' \leftarrow \text{perturber}(\mathbb{B}')$
21: $\quad\quad\mathbb{B}' \leftarrow \Phi(X, C \wedge e > e^*, \mathbb{B}')$
22: $\quad\quad$**si** $\mathbb{B}' \neq \varnothing$ **et** $\overline{e}^{\mathbb{B}'} > e^{\mathbb{B}}$ **alors**
23: $\quad\quad\quad e^{\mathbb{B}} \leftarrow \overline{e}^{\mathbb{B}'}$
24: $\quad\quad\quad s^{\mathbb{B}} \leftarrow \mathbb{B}'$
25: $\quad\quad$**fin si**
26: $\quad\quad j \leftarrow j + 1$
27: $\quad$**fin tant que**
28: $\quad$**retourner** $(e^{\mathbb{B}}, s^{\mathbb{B}})$
29: **fin tant que**

Algorithme 6.3: calculerBorneInférieure — calculer une erreur atteignable

virgule flottante $\mathbb{F}$ est un ensemble fini, l'algorithme à besoin d'un nombre fini d'itérations pour explorer complètement l'espace de recherche et terminer.

La correction de l'encadrement de l'erreur maximale est assurée pour chaque borne par sa méthode de calcul. La borne supérieure, $\overline{e}$, sur-approximant l'erreur, est calculée par un filtrage en programmation par contraintes (voir chapitre 5). Les expressions pour l'erreur sont évaluées par arithmétique des intervalles sur $\mathbb{Q}$, assurant que les intervalles obtenus soient conservateurs de l'ensemble des solutions, au risque même parfois de surestimer les bornes des intervalles [4]. Ainsi, $e \leq \overline{e}$ est toujours satisfait. La borne inférieure, e^*, sous-approximant l'erreur, est une erreur atteignable et les valeurs d'entrée l'exerçant sont connues. Une erreur atteignable assure qu'elle est toujours plus petite ou égale à l'erreur maximale produite par le programme, une erreur plus grande que l'erreur maximale étant impossible en pratique. Cela permet d'affirmer que $e^* \leq e$ est toujours vrai. En combinant ces deux bornes, l'encadrement de l'erreur maximale, $e^* \leq e \leq \overline{e}$ est toujours correct par construction.

6.3 Conclusion

Dans ce chapitre, nous avons introduit un algorithme pour obtenir un encadrement rigoureux de l'erreur maximale produite par un programme sur les nombres à virgule flottante. Cette erreur maximale est exprimée en valeur absolue et représente la plus grande déviation possible entre l'exécution du programme en machine sur $\mathbb{F}$ et son équivalent mathématique exact sur $\mathbb{R}$. Notre algorithme est basé sur un branch-and-bound maximisant l'erreur. Il produit deux bornes pour l'erreur maximale : une sur-approximation correcte, obtenue à partir du filtrage de notre système de contraintes pour les erreurs d'arrondi, et une sous-approximation correcte et atteignable, calculée à l'aide d'une procédure générer-et-tester couplé à une recherche locale. L'originalité de notre borne inférieure est que les valeurs d'entrée permettant de l'exercer sont connues. À notre connaissance, notre algorithme est le premier à combiner une sur-approximation et une sous-approximation de l'erreur pour fournir un encadrement correct de l'erreur maximale. Les approches existantes en approximation d'erreurs ne s'intéressent qu'à une des deux bornes et ne permettent pas de quantifier la distance par rapport à l'erreur maximale (voir chapitre 7). Nous avons également proposé des critères d'arrêts supplémentaires ainsi que des définitions de cas particuliers afin d'accélérer la résolution du problème tout en calculant des bornes satisfaisantes pour l'erreur maximale.

4. Rappelons que cette surestimation, nommée problème de dépendance, est due aux occurrences multiples de variables dans une expression.

CHAPITRE 7

Implémentation et expérimentation

Ce chapitre présente les principaux éléments de l'implémentation d'un solveur de contraintes, dédié aux erreurs, basée sur les contributions des chapitres 5 et 6 dans un prototype appelé FErA, pour Floating-point Error Analyzer. Les performances de notre solveur, FErA, sont comparées aux autres outils de l'état de l'art. Les critères d'arrêts ainsi que la validité de l'encadrement produit par FErA sont également analysés.

Le système de contraintes pour les erreurs d'arrondi (voir chapitre 5) et l'algorithme pour encadrer rigoureusement l'erreur maximale (voir chapitre 6) sont implémentés dans un solveur de programmation par contraintes sur les nombres à virgule flottante appelé FErA, pour *Floating-point Error Analyzer*. Ce solveur est construit à partir d'Objective-CP [Hentenryck et Michel, 2013], un système d'optimisation, et FPCS, un solveur de contraintes sur les nombres à virgule flottante basé sur les fonctions de projections définies dans [Michel *et al.*, 2001, Michel, 2002, Botella *et al.*, 2006, Marre et Michel, 2010]. Objective-CP est un système d'optimisation au sens large et est capable de résoudre des problèmes de programmation linéaire (LP) et de programmation linéaire en nombres entiers (MIP) en plus des problèmes de programmation par contraintes. Dans ce système, un problème d'optimisation est vu comme la combinaison d'un modèle, d'une recherche, et d'un solveur. La modélisation utilise la notation classique en programmation par contraintes, avant d'être traduite dans celle du solveur utilisé pour la résolution. Un modèle est donc indépendant du solveur qui va le résoudre. La recherche est spécifiée à haut niveau et utilise un langage permettant d'écrire une procédure d'exploration générique et indépendante du solveur. L'implémentation de ce système repose sur une séquence de transformation de modèles, suivi d'une concrétisation dans le solveur cible. Le solveur de contraintes interne à ce système a une architecture en *micro-noyau*, ou *micro-kernel*. Une telle architecture facilite la maintenance et l'extensibilité du système.

Le solveur FPCS contient de nombreuses fonctions de projections pour les opérations sur les nombres à virgule flottante. Ces fonctions réduisent les domaines de valeurs, représentés par des intervalles dans $\mathbb{F}$, pour toutes les variables du problème. Les intervalles produits par ces fonctions sont conservateurs des solutions. Le solveur FPCS et les stratégies de choix de variables et de valeurs introduites dans [Zitoun, 2018] ont déjà été intégrés à Objective-CP.

Dans ce chapitre, l'implémentation des chapitres 5 et 6 est abordée afin d'illustrer les choix d'implémentation. Les performances de FErA sont ensuite évaluées sur un ensemble de programmes standards à la communauté de recherche sur les nombres à virgule flottante. Ces résultats sont comparés avec d'autres outils de l'état de l'art produisant une sur-approximation ou une sous-approximation de l'erreur (voir chapitre 4).

7.1 Système de contraintes pour les erreurs d'arrondi

FErA utilise la librairie GMP [Granlund et the GMP development team, 2016], ou *GNU Multiple Precision Arithmetic Library*, pour représenter une erreur dans $\mathbb{Q}$ et effectuer les opérations nécessaires sur les erreurs. Les opérateurs disponibles dans cette librairie sont naturellement étendus aux intervalles. Comme l'erreur étudiée par notre système est restreinte à celle produite par les opérations arithmétiques ($\oplus, \ominus, \otimes, \oslash$), l'ensemble des calculs peut être effectué sur $\mathbb{Q}$. L'ensemble des opérations disponible dans GMP ne permet pas de représenter les valeurs spéciales de l'erreur donnée par notre modèle (voir définition 5.3.4). C'est pourquoi l'ensemble des rationnels est étendu à $\{-\infty, +\infty, \mathrm{NaN}\}$.

Traitement du modèle La modélisation dans Objective-CP consiste à définir un modèle abstrait indépendant du solveur. Ce modèle abstrait est capable de représenter les contraintes d'un problème de satisfaction sur les nombres à virgule flottante. Le modèle abstrait est ensuite transformé de manière automatique afin de générer un nouveau modèle où les contraintes sont toutes décomposées en contraintes ternaires ou binaires. Ce nouveau modèle reste équivalent au modèle abstrait, et permet l'application d'une 2B-consistance (voir définition 3.1.4). Une fois ce modèle

transformé, il est concrétisé afin de l'adapter au solveur qui doit le résoudre. Le modèle concret conserve la séquence de transformations des contraintes du modèle abstrait, permettant ainsi d'exprimer des étapes de la recherche en fonction des variables du modèle abstrait. Une fonction de concrétisation associe une variable concrète à chaque variable du modèle abstrait. Elle fait aussi le lien entre une contrainte abstraite dans le modèle et sa concrétisation, couplée à un filtrage dédié, dans le modèle concret.

La notation utilisée dans le modèle abstrait est étendue aux erreurs à travers l'ajout de variables et de contraintes dédiées. La nature de ces contraintes est différente des contraintes existantes pour les variables dans $\mathbb{F}$. Une contrainte sur les erreurs est équivalente à une contrainte sur $\mathbb{Q}$. Afin d'être complet, notre solveur contient différentes contraintes sur $\mathbb{Q}$ telles que les contraintes arithmétiques $(+, -, \times, \div)$, les contraintes de comparaisons $(<, \leq, >, \geq, =, \neq)$, des contraintes unaires $(|x|, -x)$, ou bien des contraintes de conversions de $\mathbb{F}$ vers $\mathbb{Q}$. L'exemple 7.1.1 illustre la modélisation d'un problème de satisfaction de contraintes dans FErA.

Exemple 7.1.1 – Soit le programme sur les nombres à virgule flottante implémentant deux opérations arithmétiques et donné dans le programme 7.1 écrit en C.

```
double simple(double x)
{
  double a =  4;
  double b =  0.2;

  return (a*x)/(b+x);
 }
```

Programme 7.1: simple.

La traduction de ce programme sous forme de CSP est directe et est donnée dans 7.1. La variable d'entrée x est ici restreinte à l'intervalle $[1, 2]$.

$$\begin{aligned} x &\in [1, 2] \\ a &\leftarrow 4 \\ b &\leftarrow 0{,}2 \\ z &= \frac{a \times x}{b + x} \end{aligned} \tag{7.1}$$

Le modèle représentant le CSP de 7.1 dans FErA est donné dans le programme 7.2.

Les variables du problème sont toutes sur les nombres à virgule flottante en double précision et sont du type `ORDoubleVar`. La catégorie de la variable est spécifiée à la déclaration : `doubleInputVar` pour une variable d'entrée, `doubleVar` pour une variable intermédiaire, ou `doubleConstantVar` pour une constante.

```
/* Création du modèle */
id<ORModel> mdl = [ORFactory createModel];

/* Déclaration des variables du problème */
id<ORDoubleVar> x = [ORFactory doubleInputVar:mdl low:1.0 up:2.0 name:@"x"];
id<ORDoubleVar> a = [ORFactory doubleVar:mdl name:@"a"];
id<ORDoubleVar> b = [ORFactory doubleConstantVar:mdl value:0.2 string:@"2/10"
    name:@"b"];
id<ORDoubleVar> z = [ORFactory doubleVar:mdl name:@"z"];

/* Contrainte d'affectation pour la constante exacte */
[mdl add:[a set: @(4.0)]];

/* Contraintes arithmétiques */
[mdl add:[z set: [[a mul: x] div: [b plus: x]] ]];

/* Création du solveur */
id<CPProgram> cp = [ORFactory createCPProgram:mdl];

/* Résolution du problème de satisfaction de contraintes */
[cp solve];
```

Programme 7.2: Modélisation dans FErA d'un problème de satisfaction de contraintes.

Filtrage Notre filtrage dédié aux erreurs s'intègre directement dans les mécanismes de résolution d'Objective-CP. Les fonctions de projections pour chaque contrainte arithmétique couplée à celles existantes dans FPCS pour les domaines de valeurs donnent directement les propagateurs des contraintes. Ces fonctions de projections s'écrivent facilement grâce aux opérateurs sur les intervalles dans $\mathbb{Q}$ implémentés à partir de GMP. La propagation des contraintes s'effectue grâce à un algorithme AC3 [Mackworth, 1977]. Les contraintes sur $\mathbb{Q}$, utilisées pour modéliser des relations entre erreurs, nécessitent un filtrage dédié. Ce filtrage repose sur des fonctions de projections directement issue de l'arithmétique des intervalles sur les réels [Moore *et al.*, 2009]. Les opérations sur $\mathbb{Q}$ étant exactes, l'écriture de ces fonctions est directe. L'exemple 7.1.2 donne la sortie produite par FErA lors de la résolution du modèle de l'exemple 7.1.1.

Exemple 7.1.2 – Soit le modèle donné dans l'exemple 7.1. La résolution de ce modèle avec FErA produit la sortie donnée ci-dessous.

```
a : [4.0000000000e+00;4.0000000000e+00]+[+0.0000000000e+00;+0.0000000000e+00]
b : [2.0000000000e-01;2.0000000000e-01]+[-1.1102230246e-17;-1.1102230246e-17]
x : [1.0000000000e+00;2.0000000000e+00]+[-1.1102230246e-16;+2.2204460493e-16]
z : [1.8181818182e+00;6.6666666667e+00]+[-3.5280420560e-15;+3.7130792268e-15]
```

Chaque ligne correspond à une variable du problème et respecte le format *nom de la variable* : *[domaine de valeurs]+[domaine d'erreurs]*. Une sur-approximation correcte de l'erreur d'après les domaines réduits produit par FErA est la borne supérieure, en valeur absolue, de l'erreur de la variable z, *i.e.*, $|e_z| \leq 3{,}713\,079\,226\,8 \times 10^{-15}$. Sachant que les erreurs sont calculées uniquement dans $\mathbb{Q}$, FErA est aussi capable d'afficher la valeur de l'erreur sous forme de fraction, comme montré dans l'équation 7.2.

$$|e_z| \leq \frac{4884004232559322171433223624589319}{1315351473597812151774931623995582559923354992640} \approx 3{,}713\,079\,226\,8 \times 10^{-15} \tag{7.2}$$

Recherche Notre système de contraintes ne nécessite pas de stratégie de recherche dédiée pour effectuer une simple réduction de domaine. Néanmoins, les stratégies existantes sur les nombres à virgule flottante [Zitoun, 2018] sont directement utilisables dans la modélisation d'un problème. Afin de pouvoir résoudre un problème de satisfaction de contraintes uniquement sur $\mathbb{Q}$, une **bissection** (voir chapitre 3) classique est implémentée pour les domaines sur les rationnels.

7.2 Algorithme pour encadrer rigoureusement les erreurs d'arrondi

Une procédure de recherche dans Objective-CP est spécifiée sous forme de constructions de haut-niveaux non-déterministes, de combinateurs de recherche, et de stratégies de sélection de nœuds. Cette spécification combine les avantages des contrôleurs de recherche et des continuations [Hentenryck et Michel, 2006] ainsi que des combinateurs compositionnels [Schrijvers *et al.*, 2013]. Notre algorithme pour encadrer rigoureusement l'erreur maximale produite par un programme s'intègre naturellement dans les mécanismes d'exploration d'Objective-CP. L'implémentation du branch-and-bound repose sur un contrôleur de recherche dédié et sur le langage de recherche offert par Objective-CP afin de créer une procédure de recherche spécifique.

La modélisation d'un problème d'optimisation sous contraintes pour produire un encadrement des erreurs d'arrondi est similaire à celle d'un problème de satisfaction de contraintes. La fonction objectif est ajoutée sous la forme d'une contrainte maximisant la valeur absolue d'une erreur pour une variable du problème. La stratégie de sélection de valeurs est spécifiée au niveau du modèle. Cela permet de profiter des différentes stratégies de coupe existantes sur $\mathbb{F}$ et issues de [Zitoun, 2018].

Une 3B-consistance (voir définition 3.1.5) est implémentée pour le filtrage des erreurs. Cette consistance, combinée à la 3B-consistance pour les domaines de valeurs, produit de meilleures réductions de domaines pour des variables du problème. Néanmoins, un tel filtrage est extrêmement couteux en temps [1] et ne laisse pas notre algorithme résoudre des problèmes dans un temps raisonnable. Les performances de la 3B-consistance dans FErA ne sont pas évaluées dans ce chapitre. Toutefois, la 3B-consistante est particulièrement efficace en présence d'occurences multiples. En effet, énumérer les valeurs aux bornes des domaines des variables permet d'attenuer le problème de dépendance et d'obtenir une sur-approximation plus fine des erreurs.

7.3 Expérimentation

Les performances de notre solveur FErA sont évaluées sur un ensemble de programmes issu de la suite FPBench [Damouche *et al.*, 2017b]. FPBench est une proposition de standard commun pour les problèmes sur les nombres à virgule flottante. En plus du large choix de programmes sur lesquels expérimenter, il propose de nombreux outils permettant de traduire un programme dans le langage source vers différents langages de programmation (C, Go, JavaScript, ...) ou bien dans des formats spécifiques à un outil (FPTaylor, Gappa, SMT-LIB2, ...). FPBench propose différentes méta-données pour les expérimentations permettant de représenter les propriétés des problèmes. Les programmes choisis dans FPBench sont restreints aux quatre opérations arithmétiques ($\oplus, \ominus, \otimes, \oslash$) afin de pouvoir les résoudre dans FErA. Ces programmes sont uniquement sur des nombres à virgule flottante en double précision, comme définis dans le standard IEEE 754 [IEEE,

1. L'infinité de valeurs dans les domaines d'erreurs sur $\mathbb{Q}$ ne permet pas d'énumerer directement les valeurs aux bornes des domaines et nécessite d'approximer le pas d'énumération.

2008]. Toutes les expérimentations sont réalisées sur un MacBook Pro avec un processeur 2,8 GHz Intel Core i7–7700HQ et 16 GB de mémoire vive sous le système macOS Catalina (10.15.4). Les temps sont exprimés en secondes et TO indique que la résolution est arrêtée automatiquement à 10 minutes.

7.3.1 Comparaison des critères d'arrêts

La table 7.1 compare le comportement de notre algorithme avec différents critères d'arrêts. Les résultats sont classés entre la **meilleure**, la **seconde** meilleure, et la **pire** sur-approximation $\overline{e}$ pour chaque programme. Le critère idéal $e^* = \overline{e}$ arrête l'algorithme si et seulement si la borne inférieure de l'erreur maximale atteint sa borne supérieure. Bien entendu, ce critère est difficile à satisfaire en pratique et la résolution est toujours stoppée à 10 minutes. [2]. Malgré l'arrêt brutal de l'algorithme, ce critère permet d'obtenir les meilleures valeurs de e^* et de $\overline{e}$ produites par notre solveur. Le critère $e^* = \overline{e}$ w. s. combine le critère idéal avec le concept de boîte *écartée* (voir chapitre 6), *i.e.,* une boîte qui est *écartée* une fois que toutes les contraintes arithmétiques du CSP valide la proposition 6.2.1. Cette combinaison de critères permet à FErA d'obtenir un encadrement de l'erreur maximale dans un temps raisonnable pour tous les programmes, excepté `kepler2`. Toutefois, cette réduction du temps de résolution est obtenue au prix d'une dégradation des bornes de l'erreur maximale. Le critère $\frac{\overline{e}}{e^*} \leq 2$, est une relaxation du critère d'arrêt idéal. L'idée est d'arrêter la résolution lorsque le ratio de la borne supérieure par la borne inférieure est plus petit ou égal à 2. Ici, ce critère nécessite de stopper la résolution de `sine` et `kepler2` à 10 minutes. Un tel comportement n'est pas surprenant. Les résultats avec le critère idéal montre que FErA a des difficultés à atteindre ce ratio sur ces programmes, probablement à cause de la grande quantité d'occurrences multiples au niveau des variables du problème. Le critère $\frac{\overline{e}}{e^*} \leq 2$ w. s. combine le ratio inférieur à 2 avec les boîtes *écartées*. La résolution termine pour l'ensemble des problèmes, mais ne produit pas des bornes pour l'erreur maximale aussi resserrées que les autres critères.

7.3.2 Comparaison avec les autres outils de l'état de l'art

Les performances de FErA sont comparées à plusieurs outils de l'état de l'art produisant une sur-approximation de l'erreur, tel que Fluctuat [Goubault et Putot, 2006, Goubault et Putot, 2011], Gappa [Daumas et Melquiond, 2010], PRECiSA [Moscato *et al.*, 2017, Titolo *et al.*, 2018], Real2Float [Magron *et al.*, 2017], Daisy [Izycheva et Darulova, 2017, Darulova *et al.*, 2018b, Darulova *et al.*, 2018a, Darulova et Volkova, 2019], Rosa [Darulova et Kuncak, 2014, Darulova et Kuncak, 2017], et FPTaylor [Solovyev *et al.*, 2015, Solovyev *et al.*, 2018], ou une sous-approximation de l'erreur, tel que S^{3}FP [Chiang *et al.*, 2014]. La table 7.2 regroupe les versions utilisées pour chaque outil lors des expérimentations.

À noter que les résultats pour S^{3}FP sont issus de [Solovyev *et al.*, 2018]. Les auteurs affirment dans [Chiang *et al.*, 2014] que l'outil est uniquement adapté aux nombres à virgule flottante en simple précision. Le code source de S^{3}FP ne permet donc pas de générer aléatoirement des nombres à virgule flottante en double précision afin d'obtenir l'erreur produite par un programme en double précision. Les temps de résolution pour S^{3}FP ne sont pas donnés dans [Solovyev *et al.*, 2018]. Néanmoins, dans l'article de l'outil [Chiang *et al.*, 2014], les auteurs indiquent

2. Excepté pour `sqroot` qui a une seule variable d'entrée $x \in [0, 1]$. FErA élimine la boîte pour la moitié inférieure de $\mathbf{x}$ rapidement, ce qui permet à l'algorithme de terminer après avoir exploré ou éliminé les autres boîtes de la moitié supérieure.

Table 7.1 – Comparaison des différents critères d'arrêts du branch-and-bound dans FErA

	filtrage	$e^* = \overline{e}$		$e^* = \overline{e}$ w. s.		$\frac{\overline{e}}{e^*} \leq 2$		$\frac{\overline{e}}{e^*} \leq 2$ w. s.	
		e^*	$\overline{e}$	e^*	$\overline{e}$	e^*	$\overline{e}$	e^*	$\overline{e}$
carbonGas	**4,24e-08**	4,28e-9	**6,01e-9**	2,95e-9	**7,02e-9**	3,60e-9	**7,02e-9**	3,63e-9	**7,02e-9**
	0,017s		TO		0,345s		1,419s		0,266s
verhulst	**4,20e-16**	2,44e-16	**2,83e-16**	2,19e-16	**2,87e-16**	1,82e-16	2,91e-16	1,64e-16	3e-16
	0,016s		TO		0,034s		0,024s		0,018s
predPrey	**1,84e-16**	1,54e-16	**1,66e-16**	1,03e-16	**1,68e-16**	9,31e-17	1,72e-16	9,88e-17	**1,84e-16**
	0,011s		TO		0,084s		0,041s		0,018s
rigidBody1	**2,95e-13**	2,88e-13	**2,95e-13**	1,95e-13	**2,95e-13**	1,48e-13	**2,95e-13**	1,49e-13	**2,95e-13**
	0,018s		TO		1,659s		0,370s		0,543s
rigidBody2	**3,61e-11**	3,13e-11	**3,61e-11**	2,52e-11	**3,61e-11**	1,83e-11	**3,61e-11**	2,11e-11	**3,61e-11**
	0,022s		TO		3,298s		1,367s		1,266s
doppler1	**4,97e-13**	1,18e-13	**1,49e-13**	7,34e-14	1,57e-13	1,03e-13	**1,52e-13**	7,76e-14	1,55e-13
	0,021s		TO		0,752s		1,099s		0,757s
doppler2	**1,34e-12**	2,16e-13	**2,71e-13**	1,12e-13	3,36e-13	1,36e-13	**2,72e-13**	1,41e-13	3,36e-13
	0,034s		TO		0,356s		1,416s		0,270s
doppler3	**1,92e-13**	6,26e-14	**8,44e-14**	4,09e-14	9e-14	4,46e-14	**8,68e-14**	3,93e-14	9e-14
	0,023s		TO		0,341s		0,455s		0,311s
turbine1	**2,17e-13**	1,34e-14	**1,74e-14**	1,05e-14	**1,77e-14**	1,07e-14	2,02e-14	9,35e-15	1,81e-14
	0,016s		TO		8,514s		2,289s		6,042s
turbine2	**3,05e-13**	1,56e-14	**2,32e-14**	1,32e-14	**2,36e-14**	1,39e-14	2,43e-14	1,17e-14	2,36e-14
	0,025s		TO		2,803s		1,581s		2,952s
turbine3	**1,56e-13**	6,40e-15	**1,11e-14**	4,76e-15	**1,11e-14**	5,73e-15	**1,11e-14**	5,61e-15	**1,11e-14**
	0,026s		TO		2,766s		3,961s		1,800s
sqroot	**5,78e-16**	4,53e-16	**5,33e-16**	4,23e-16	**5,33e-16**	3,46e-16	**5,78e-16**	3,70e-16	**5,78e-16**
	0,032s		3,200s		2,831s		0,050s		0,050s
sine	**7,42e-16**	2,91e-16	**7,04e-16**	2,85e-16	**7,04e-16**	2,89e-16	**7,04e-16**	2,79e-16	**7,04e-16**
	0,027s		TO		102,982s		TO		101,309s
sineOrder3	**1,12e-15**	3,25e-16	**6,36e-16**	3,28e-16	**6,36e-16**	3,19e-16	**6,36e-16**	3,30e-16	**6,36e-16**
	0,021s		TO		1,388s		1,433s		1,504s
kepler0	**1,19e-13**	5,90e-14	**9,60e-14**	5,78e-14	**9,62e-14**	5,01e-14	**9,82e-14**	4,97e-14	**9,82e-14**
	0,037s		TO		TO		1,937s		2,798s
kepler1	**4,95e-13**	1,68e-13	**3,10e-13**	1,41e-13	**3,11e-13**	1,63e-13	3,15e-13	1,64e-13	3,15e-13
	0,031s		TO		51,303s		15,136s		12,691s
kepler2	**2,43e-12**	7,99e-13	**1,83e-12**	6,53e-13	**1,84e-12**	6,98e-13	**1,83e-12**	6,73e-13	**1,84e-12**
	0,027s		TO		58,834s		TO		72,622s

Table 7.2 – Versions des outils de l'état de l'art utilisés pour les expérimentations

Outils	Versions
Fluctuat	version 3.1390 avec division d'intervalles
Gappa	version 1.3.5 avec indices avancés
PRECiSA	version 2.1.1
Real2Float	version 0.7
Daisy	master branch, commit 8f26766
Rosa	master branch, commit 68e58b8
FPTaylor	master branch, commit 147e1fe avec l'optimiseur Gelpia

Table 7.3 – Comparaison des bornes inférieures de l'erreur entre S^{3}FP et FErA

	S^{3}FP	FErA
carbonGas	4,20e-09	**4,29e-09**
verhulst	2,40e-16	**2,44e-16**
predPrey	1,50e-16	**1,54e-16**
rigidBody1	2,70e-13	**2,91e-13**
rigidBody2	3e-11	**3,30e-11**
doppler1	1e-13	**1,18e-13**
doppler2	1,90e-13	**2,16e-13**
doppler3	5,70e-14	**6,35e-14**
turbine1	1,10e-14	**1,42e-14**
turbine2	1,40e-14	**1,56e-14**
turbine3	6,20e-15	**6,60e-15**
sqroot	**4,70e-16**	4,63e-16
sine	2,90e-16	**2,94e-16**
sineOrder3	4,10e-16	**4,12e-16**
kepler0	5,30e-14	**5,90e-14**
kepler1	1,60e-13	**1,68e-13**
kepler2	**8,40e-13**	8,39e-13

une limite de temps de 1 heure. Cette limite semble réaliste par rapport aux mécanismes de l'outil. S^{3}FP réalise une exploration de l'espace de recherche guidée par un découpage des domaines initiaux des variables. L'instanciation aléatoire des variables du programme et l'absence de sur-approximation de l'erreur ne donnent aucune garantie de trouver une bonne erreur pouvant servir de sous-approximation dans un temps raisonnable. Le processus d'exploration de S^{3}FP peut également être poussé jusqu'à l'énumération complète de toutes les valeurs possibles pour chaque variable d'un programme [3]. Laisser FErA calculer une sous-approximation de l'erreur maximale avec une durée similaire, donne une borne inférieure atteignable meilleure pour la plupart des problèmes. La table 7.3 compare S^{3}FP et FErA pour le calcul de la borne inférieure de l'erreur maximale, avec une limite de temps de 1 heure [4]. FErA est exécuté avec le critère d'arrêt idéal $e^* = \overline{e}$, afin d'explorer l'espace de recherche en profondeur.

Dans la table 7.4 [5], les résultats sont classés entre la **meilleure**, la **seconde** meilleure, et la **pire** sur-approximation de l'erreur pour chaque programme. Les lignes en gris indiquent le temps pour calculer la borne donnée au-dessus. Les colonnes S^{3}FP et e^* donnent une borne inférieure pour l'erreur maximale, tandis que toutes les autres colonnes donnent une borne supérieure de l'erreur maximale. Pour FErA, la colonne *filtrage* donne la sur-approximation de l'erreur obtenue après application d'un simple filtrage de notre système de contraintes, tandis que les colonnes e^* et $\overline{e}$ donnent respectivement la meilleure erreur atteignable, ou sous-approximation, et la plus

3. Une telle énumération n'est pas réalisable en pratique et demande donc à S^{3}FP de fixer une limite de temps afin de garantir la terminaison de son algorithme.

4. Cette limite de temps pour S^{3}FP est estimée à partir du discours des auteurs dans [Chiang *et al.*, 2014].

5. La sur-approximation de l'erreur calculée par PRECiSA, pour `sqroot`, est plus petite que la borne inférieure de l'erreur calculée par FErA et S^{3}FP. Nous supposons qu'il s'agit d'un bug dans PRECiSA lors la concrétisation numérique de l'erreur via le solveur Kodiak.

Table 7.4 – Comparaison des approximations d'erreurs entre FErA et les autres outils

	Fluctuat	Gappa	PRECiSA	Real2Float	Daisy	Rosa	FPTaylor	S^{3}FP	FErA		
									filtrage	e^*	$\overline{e}$
carbonGas	1,17e-08	**6,03e-09**	7,10e-09	2,21e-08	**3,92e-08**	1,60e-08	**4,97e-09**	4,20e-09	4,24e-08	3,63e-9	7,02e-9
	0,123s	3,445s	0,034s	6,887s	37,750s	37,581s	0,320s		0,017s		0,266s
verhulst	4,81e-16	**2,85e-16**	**5,14e-16**	4,67e-16	3,72e-16	4,67e-16	**2,48e-16**	2,40e-16	4,20e-16	1,64e-16	3e-16
	0,108s	0,619s	0,023s	4,675s	28,250s	15,762s	0,290s		0,016s		0,018s
predPrey	2,36e-16	**1,68e-16**	2,09e-16	**2,52e-16**	1,75e-16	1,98e-16	**1,59e-16**	1,50e-16	1,84e-16	9,88e-17	1,84e-16
	0,107s	2,166s	0,020s	7,269s	29,500s	33,220s	0,410s		0,011s		0,018s
rigidBody1	**3,22e-13**	**2,95e-13**	3,24e-13	**5,33e-13**	**2,95e-13**	**3,22e-13**	**2,95e-13**	2,70e-13	2,95e-13	1,49e-13	**2,95e-13**
	2,794s	2,359s	0,033s	3,230s	27,983s	7,505s	0,280s		0,018s		0,543s
rigidBody2	**3,65e-11**	**3,61e-11**	**3,65e-11**	**6,48e-11**	**3,61e-11**	**3,65e-11**	**3,61e-11**	3e-11	3,61e-11	2,11e-11	**3,61e-11**
	5,090s	3,657s	0,370s	3,698s	32,683s	10,377s	0,310s		0,022s		1,266s
doppler1	**1,28e-13**	1,61e-13	2,09e-13	**7,64e-12**	4,20e-13	2,69e-13	**1,22e-13**	1e-13	4,97e-13	7,76e-14	1,55e-13
	8,347s	5,542s	0,044s	26,821s	30,817s	24,298s	1,450s		0,021s		0,757s
doppler2	**2,36e-13**	2,86e-13	3,08e-13	**8,85e-12**	1,05e-12	6,46e-13	**2,23e-13**	1,90e-13	1,34e-12	1,41e-13	3,36e-13
	8,244s	5,634s	0,041s	26,731s	34,000s	24,073s	1,730s		0,034s		0,270s
doppler3	**7,13e-14**	8,76e-14	9,50e-14	**4,07e-12**	1,69e-13	1,01e-13	**6,62e-14**	5,70e-14	1,92e-13	3,93e-14	9e-14
	9,028s	5,476s	0,044s	26,057s	32,250s	31,442s	1,330s		0,023s		0,311s
turbine1	3,09e-14	2,41e-14	2,52e-14	**2,47e-11**	8,65e-14	5,99e-14	**1,67e-14**	1,10e-14	2,17e-13	9,35e-15	**1,81e-14**
	7,555s	9,816s	0,144s	127,911s	32,950s	31,400s	0,450s		0,016s		6,042s
turbine2	2,60e-14	3,33e-14	3,01e-14	**2,08e-12**	1,31e-13	7,68e-14	**2e-14**	1,40e-14	3,05e-13	1,17e-14	**2,36e-14**
	5,562s	7,395s	0,132s	22,225s	30,183s	14,890s	0,560s		0,025s		2,952s
turbine3	1,34e-14	**0,36**	1,83e-14	1,71e-11	6,24e-14	4,63e-14	**9,58e-15**	6,20e-15	1,56e-13	5,61e-15	**1,11e-14**
	7,342s	11,256s	0,193s	150,653s	31,050s	31,224s	0,520s		0,026s		1,800s
sqroot	6,84e-16	5,35e-16	**4,30e-16**	**1,29e-15**	5,71e-16	6,18e-16	**5,02e-16**	4,70e-16	5,78e-16	3,70e-16	5,78e-16
	0,120s	7,937s	0,035s	13,840s	28,000s	8,414s	0,320s		0,032s		0,050s
sine	7,42e-16	6,96e-16	**7,49e-16**	6,03e-16	1,13e-15	**5,19e-16**	**4,44e-16**	2,90e-16	7,42e-16	2,79e-16	7,04e-16
	0,126s	40,351s	0,132s	13,138s	27,933s	14,265s	0,450s		0,027s		101,309s
sineOrder3	1,09e-15	6,54e-16	1,23e-15	1,19e-15	**1,46e-15**	9,97e-16	**5,94e-16**	4,10e-16	1,12e-15	3,30e-16	**6,36e-16**
	0,117s	3,177s	0,021s	4,241s	25,867s	6,974s	0,290s		0,021s		1,504s
kepler0	1,03e-13	1,10e-13	1,10e-13	**1,20e-13**	1,05e-13	**8,28e-14**	**7,47e-14**	5,30e-14	1,19e-13	4,97e-14	9,82e-14
	12,611s	12,187s	0,230s	2,120s	28,033s	11,113s	0,690s		0,037s		2,798s
kepler1	3,52e-13	4,69e-13	4,04e-13	4,68e-13	**4,81e-13**	4,14e-13	**2,87e-13**	1,60e-13	4,95e-13	1,64e-13	**3,15e-13**
	252,468s	19,785s	0,683s	93,202s	28,933s	134,149s	1,710s		0,031s		12,691s
kepler2	2,24e-12	2,41e-12	**1,67e-12**	2,10e-12	**2,47e-12**	2,16e-12	**1,58e-12**	8,40e-13	2,43e-12	6,73e-13	1,84e-12
	33,600s	39,048s	31,235s	59,881s	30,483s	72,847s	0,580s		0,027s		72,622s

petite sur-approximation de l'erreur calculée par le branch-and-bound de FErA. Ici, FErA est exécuté avec le critère d'arrêt $\frac{\overline{e}}{e^*} \leq 2$ w. s. afin d'obtenir un temps de résolution comparable aux autres outils et de conserver des bornes satisfaisantes pour l'erreur maximale. Sur ces expérimentations, FErA se classe premier deux fois (`rigidBody1` et `rigidBody2`), deuxième cinq fois (`turbine1`, `turbine2`, `turbine3`, `sineOrder3`, et `kepler1`), et troisième cinq fois (`carbonGas`, `verhulst`, `doppler1`, `kepler0`, `kepler2`) en terme de meilleure sur-approximation. À noter que FErA n'est jamais dernier, *i.e.,* il ne produit jamais la pire sur-approximation de l'erreur. Les deux bornes produites par notre outil sont du même ordre de grandeur pour la majorité des problèmes. Le manque d'un traitement dédié aux occurrences multiples dans FErA est mis en avant par la sur-approximation de la borne supérieure obtenue pour le problème `sine`. Dans ce cas, le processus de coupe utilisé dans le branch-and-bound n'est pas suffisant pour faire baisser la valeur de cette borne. FErA résout la plupart des programmes dans un temps raisonnable à l'exception de `kepler2`. Les programmes Kepler sont les problèmes avec le plus grand nombre de variables d'entrée et notre solveur est plus performant sur des petits problèmes. Toutefois, un des avantages de FErA est qu'il utilise un algorithme *anytime*, *i.e.,* un algorithme qui peut être arrêté à la fin de n'importe quelle itération et qui produit toujours un encadrement correct de l'erreur maximale. Ceci permet d'assurer un encadrement de l'erreur maximale, même grossier, en un temps raisonnable de résolution.

7.3.3 Évolution des bornes pendant la résolution

Un suivi de l'évolution des bornes inférieures et supérieures de l'erreur maximale au cours de la résolution donne des indications sur l'efficacité de notre encadrement. Les figures 7.1, 7.2, 7.4, et 7.4 regroupent [6] l'évolution des bornes pour chaque problème. Ces expérimentations sont réalisées avec le critère d'arrêt idéal $e^* = \overline{e}$ et avec une limite de temps de 10 minutes. L'intérêt ici n'est pas d'obtenir des bornes dans un temps raisonnable, mais de visualiser l'évolution des valeurs des bornes pendant la résolution. Pour chaque graphique, la courbe rose représente la borne supérieure de l'erreur maximale tandis que la courbe verte représente la borne inférieure de l'erreur maximale. L'axe des abscisses donne le temps en secondes avec une échelle logarithmique, tandis que l'axe des ordonnées donne la valeur des bornes avec une échelle linéaire. À la fin de l'axe des ordonnées $\cdot 10^{-n}$ avec $n \in \mathbb{N}$ indique la magnitude de l'ensemble des valeurs sur cet axe.

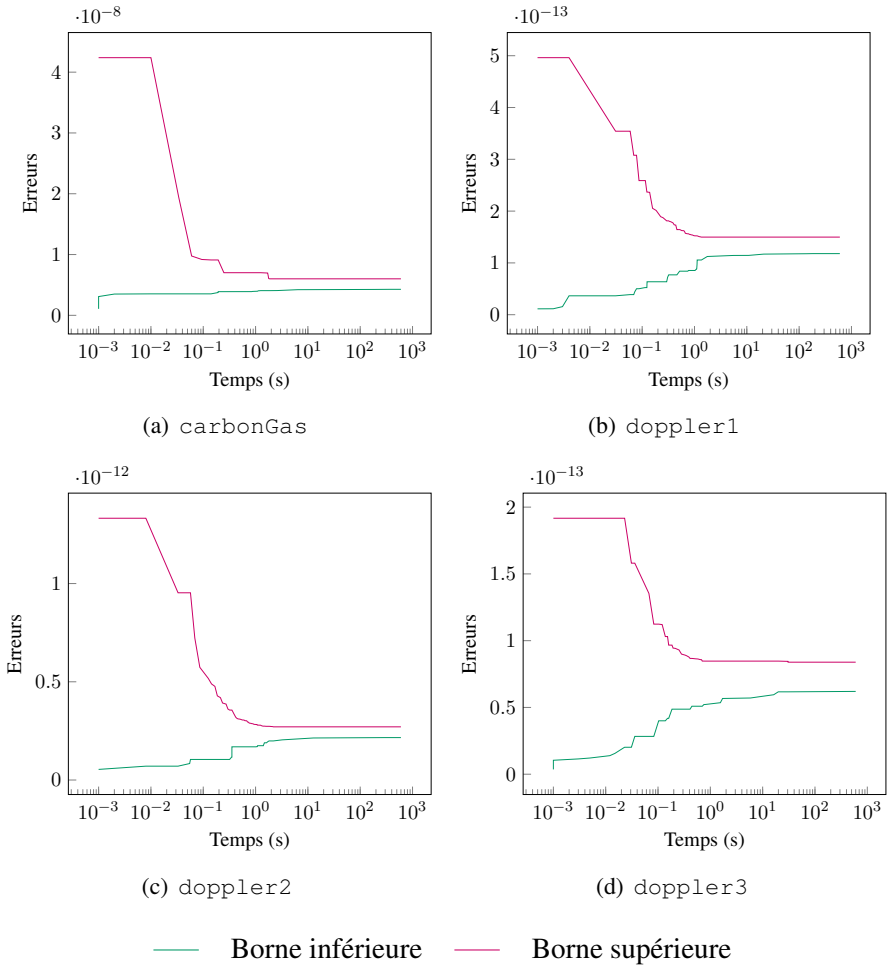


Figure 7.1 – Évolution des bornes de l'encadrement de l'erreur maximal

Pour la plupart des expérimentations, une réduction (resp. augmentation) de la borne supérieure (resp. inférieure) se produit assez tôt dans le processus de résolution. Cette amélioration des

6. La séparation en trois figures et le regroupement de problèmes dans celles-ci n'a pas de signification particulière.

bornes est due au découpage de l'espace de recherche effectuée par FErA. Ce découpage à lieu sur les domaines de valeurs des variables d'entrées du problème. Un tel découpage, associé au choix de la boîte la plus prometteuse, offre de plus grandes chances de calculer une meilleure borne inférieure. En effet, les domaines de valeurs dans lesquels une valeur est choisie pour chaque variable sont plus petits. Cela aide également le processus de filtrage, reposant sur l'arithmétique des intervalles, à obtenir des bornes plus précises pour les domaines d'erreurs des variables. Les graphiques de nombreux problèmes, tels que `carbonGas`, `rigidBody` (1 et 2), `doppler` (1, 2, et 3), ou encore `turbine` (1, 2, et 3) affichent un encadrement de l'erreur maximale serré en fin de résolution. Pour `rigidBody1` et `rigidBody2`, la borne supérieure obtenue par filtrage n'évolue jamais pendant la résolution. Ceci n'est pas surprenant, la borne supérieure calculée par simple filtrage se classe déjà première par rapport aux autres outils de l'état de l'art (voir table 7.4). De telles bornes supérieures sont donc satisfaisantes. De plus, l'évolution de la borne inférieure pour chaque problème assure que l'erreur maximale se trouve proche de la borne supérieure. Améliorer ces encadrements ou même trouver l'erreur maximale, nécessite une énumération des domaines de valeurs des variables du problème (voir la sous-section 6.2.1 du chapitre 6).

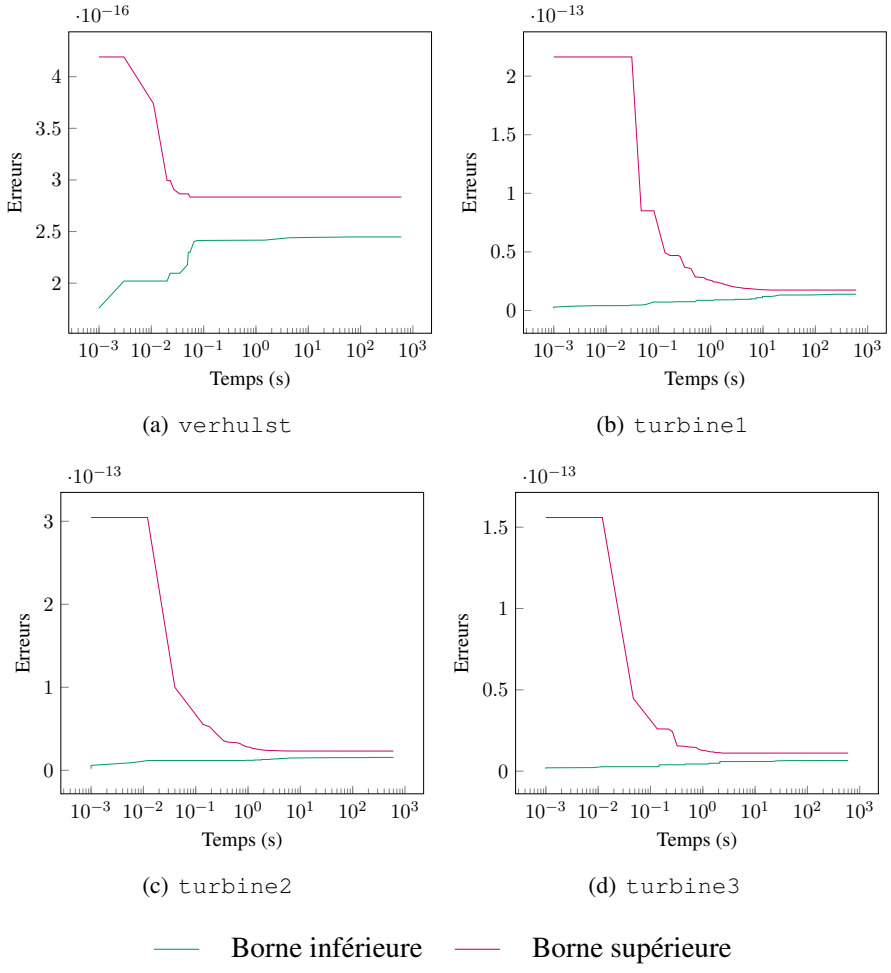
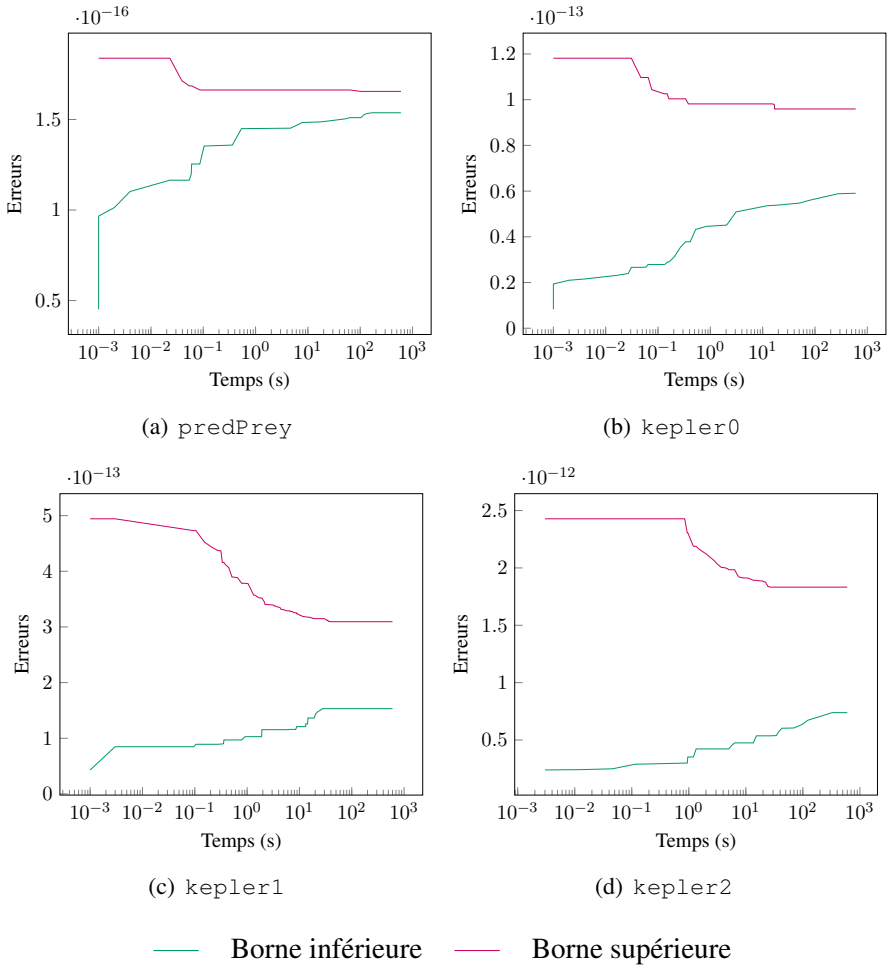


Figure 7.2 – Évolution des bornes de l'encadrement de l'erreur maximal

Le problème de dépendances, lié aux occurrences multiples de variables, est clairement visible sur certains graphiques. Notamment pour `sine`, `sineOrder3`, ou encore `sqroot`. Ces problèmes ne contiennent qu'une seule variable, x, et des constantes. La variable x est donc utilisée dans toutes les contraintes arithmétiques du problème. Ces nombreuses occurrences multiples impactent directement le calcul de la borne supérieure, qui n'est presque pas réduite lors de la résolution. La faible augmentation de la borne inférieure est également liée au maintien de la borne supérieure. Comme les valeurs des bornes sont utilisées pour évacuer des boîtes ne contenant pas de solution, cela permet d'accélérer la résolution et d'obtenir plus rapidement un meilleur encadrement de l'erreur maximale. Ici, les boîtes ne sont pas évacuées et le processus de calcul de la borne inférieure cherche à obtenir une plus grande erreur dans des boîtes où ce n'est pas possible.



(a) `predPrey` (b) `kepler0`

(c) `kepler1` (d) `kepler2`

Figure 7.3 – Évolution des bornes de l'encadrement de l'erreur maximal

Une mise à jour de la borne supérieure entraine souvent une mise à jour de la borne inférieure, et inversement. Cela est possible car notre algorithme établit des liens clairs entre les deux bornes afin d'accélérer la résolution d'un problème. Cette dépendance des bornes est visible sur les graphiques de `kepler0`, `kepler1`, et `kepler2`. Ces mises à jour s'expliquent par plusieurs règles de notre algorithme. Une borne inférieure plus grande va évacuer les boîtes ayant une borne

supérieure locale plus petite. Jeter ces boîtes aide l'algorithme à se concentrer sur celles pouvant contenir une solution. Néanmoins, la résolution pour ces problèmes stagne rapidement. Cela est dû à une combinaison du problème de dépendance et à la combinatoire importante des domaines des variables. En effet, les problèmes `kepler` sont ceux avec le plus grand nombre de variables d'entrée. L'application d'une consistance plus forte, comme la 3B-consistance, ou sa généralisation la kB-consistance [Lhomme, 1993], permet d'atténuer le problème de dépendance et d'effectuer de meilleures réductions de domaines. Toutefois, les calculs nécessaires à une telle consistance sont très couteux en temps et ne sont donc pas applicables ici pour améliorer la résolution de ces problèmes.

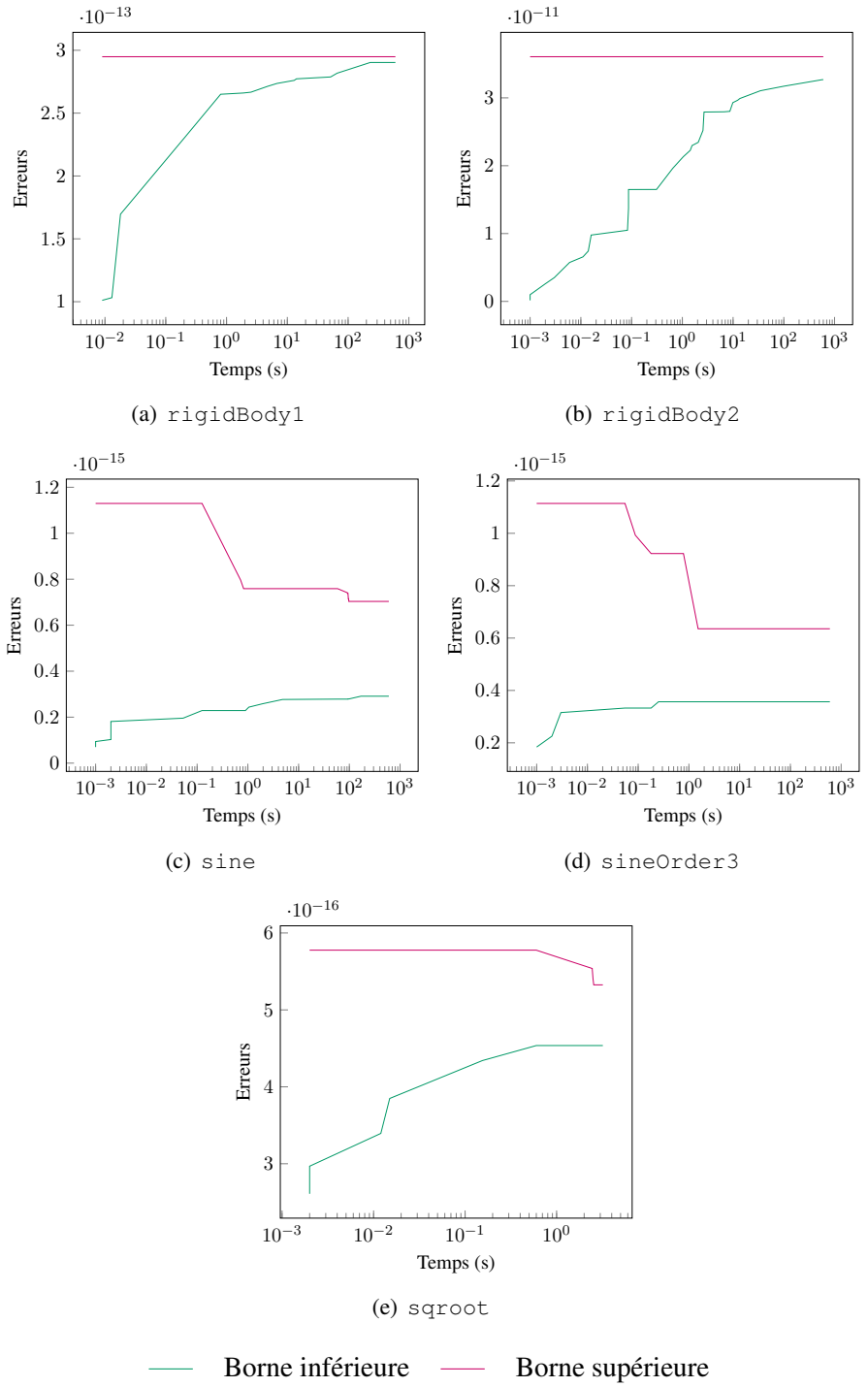
La table 7.5 donne des informations sur les données de chaque problème, telles que le nombre et la nature des contraintes, le nombre de variables, ou la quantité d'occurences pour chaque variable d'un problème.

Table 7.5 – Mesures sur les problèmes issus de FPBench

	$\#C$	C				$\#X$	occurrences
		$+$	$-$	$\times$	$\div$		
carbonGas	11	1	2	6	2	1	$\#v = 3$
verhulst	4	1	0	1	2	1	$\#x = 2$
predPrey	7	1	0	3	3	1	$\#x = 4$
rigidBody1	7	0	4	3	0	3	$\#x_1 = 2, \#x_2 = 2, \#x_3 = 2$
rigidBody2	14	2	2	10	0	3	$\#x_1 = 2, \#x_2 = 4, \#x_3 = 6$
doppler1	8	3	0	4	1	3	$\#u = 2, \#v = 1, \#T = 1$
doppler2	8	3	0	4	1	3	$\#u = 2, \#v = 1, \#T = 1$
doppler3	8	3	0	4	1	3	$\#u = 2, \#v = 1, \#T = 1$
turbine1	14	1	4	7	2	3	$\#v = 2, \#w = 2, \#r = 4$
turbine2	10	0	3	6	1	3	$\#v = 3, \#w = 2, \#r = 2$
turbine3	14	1	4	7	2	3	$\#v = 2, \#w = 2, \#r = 4$
sqroot	14	2	2	10	0	1	$\#x = 10$
sine	17	1	2	11	3	1	$\#x = 16$
sineOrder3	5	0	1	4	0	1	$\#x = 4$
kepler0	15	6	4	5	0	6	$\#x_1 = 2, \#x_2 = 3, \#x_3 = 3$ $\#x_4 = 1, \#x_5 = 3, \#x_6 = 3$
kepler1	24	8	8	8	0	4	$\#x_1 = 6, \#x_2 = 6, \#x_3 = 6$ $\#x_4 = 6$
kepler2	36	12	10	14	0	6	$\#x_1 = 6, \#x_2 = 6, \#x_3 = 6$ $\#x_4 = 6, \#x_5 = 6, \#x_6 = 6$

En général, l'encadrement de l'erreur maximale fourni par FErA est précis. Le manque de traitement des occurrences multiples limite néanmoins les réductions possibles pour la borne supérieure, ce qui impacte directement le calcul de la borne inférieure.

L'encadrement de l'erreur est proche de l'erreur maximale pour de nombreux problèmes. Obtenir un meilleur encadrement, voire trouver l'erreur maximale, peut nécessiter d'énumérer les domaines de valeurs des variables d'entrée.



(a) `rigidBody1` (b) `rigidBody2`

(c) `sine` (d) `sineOrder3`

(e) `sqroot`

Figure 7.4 – Évolution des bornes de l'encadrement de l'erreur maximal

7.4 Conclusion

Dans ce chapitre, nous avons présenté l'implémentation de nos méthodes et algorithmes présentés dans les chapitres 5 et 6 dans un solveur appelé FErA. Ce solveur est construit à partir d'Objective-CP, un système d'optimisation, et FPCS, un solveur pour les contraintes sur les nombres à virgule flottante. Nos contributions s'insèrent naturellement dans la structure d'Objective-CP et profitent de l'architecture et des fonctionnalités existantes pour la résolution de problème. Les performances de FErA sont évaluées sur un ensemble de problèmes standard de la communauté d'analyse de programme sur les nombres à virgule flottante. FErA produit un encadrement de l'erreur maximale satisfaisant et compétitif avec les bornes supérieures ou inférieures produites par d'autres outils de l'état de l'art. Les expérimentations illustrent également le comportement aux limites de notre algorithme et montrent la précision de l'encadrement produit par rapport à l'erreur maximale.

CHAPITRE 8

Conclusion et Perspectives

8.1 Conclusion

Dans cette thèse, nous nous intéressons à l'analyse de programme sur les nombres à virgule flottante et en particulier à l'analyse d'erreurs d'arrondi. Les erreurs d'arrondi sont un problème majeur pour les programmes numériques. Elles impactent l'exactitude des valeurs calculées par le programme et peuvent modifier directement son comportement. Connaître ces erreurs et pouvoir les quantifier est donc essentiel afin de mieux connaître le comportement d'un programme numérique en pratique.

Dans un premier temps, nous proposons une approche, basée sur la programmation par contraintes, afin de représenter, calculer, et raisonner sur les erreurs d'arrondi produites par un programme sur les nombres à virgule flottante. Pour cela, nous étendons la notion de problème de satisfaction de contraintes aux erreurs et proposons une modélisation de l'erreur d'arrondi. Cette extension nécessite l'introduction d'un nouveau domaine, associé à chaque variable représentant un nombre à virgule flottante, appelé domaine d'erreurs. Un tel domaine capture l'erreur associée à chaque variable du problème et la représente grâce à un intervalle sur les rationnels. En plus de ce domaine d'erreurs, un domaine de l'erreur sur l'opération est défini pour chaque contrainte arithmétique du problème. Ce domaine capture l'erreur introduite à chaque opération arithmétique dans le programme. La résolution d'un tel problème nécessite un filtrage dédié. Nous introduisons donc un filtrage pour les domaines d'erreurs afin d'obtenir une 2B-consistance. Ce filtrage tire pleinement avantage des spécificités de l'arithmétique des nombres à virgule flottante pour borner l'erreur. Afin d'exprimer des relations entre les erreurs au niveau de la modélisation, nous introduisons des contraintes sur les erreurs. Contrairement aux contraintes impliquant les domaines de valeurs de variables du problème, ces nouvelles contraintes expriment des relations sur les domaines d'erreurs. Les contraintes sur les erreurs peuvent être mixtes et exprimer une relation entre domaines de valeurs et domaines d'erreurs. L'intérêt de ces contraintes est d'introduire un raisonnement sur les erreurs afin de résoudre de nouveaux problèmes.

Dans un second temps, nous nous sommes intéressés au problème de l'erreur maximale, *i.e.,* l'erreur la plus grande, en valeur absolue, qu'un programme peut produire pour un ensemble de valeurs d'entrée donné. Nous proposons un algorithme afin d'encadrer rigoureusement l'erreur maximale. Cet algorithme se base sur un branch-and-bound cherchant à maximiser l'erreur produite. Il produit deux bornes pour l'erreur maximale : une borne supérieure, sous forme de sur-approximation correcte, et une borne inférieure, sous forme de sous-approximation atteignable de l'erreur. La borne supérieure est obtenue à partir du processus de filtrage de notre système de

contraintes pour les erreurs. La borne inférieure est calculée à l'aide d'une procédure générer-et-tester couplé à une recherche locale. Les valeurs d'entrée permettant d'exercer la borne inférieure sont connues et peuvent donc servir à la vérifier dans un oracle externe. À notre connaissance, notre algorithme est le premier à combiner une sur-approximation et une sous-approximation de l'erreur pour produire un encadrement correct de l'erreur maximale. Les bornes de notre encadrement tirent avantage l'une de l'autre afin de s'améliorer et accélérer la résolution de l'algorithme. Nous proposons également une analyse des limites et cas particuliers de notre algorithme, ainsi que des critères d'arrêts supplémentaires afin de réduire le temps de résolution tout en obtenant un encadrement satisfaisant. Un des intérêts de notre algorithme est qu'il est *anytime* : il peut être arrêté à la fin de n'importe quelle itération et produit toujours un encadrement correct de l'erreur maximale.

Les contributions présentées dans cette thèse ont été implémentées dans un solveur, appelé FErA. Ce solveur est construit à partir d'Objective-CP, un système d'optimisation, et de FPCS, un solveur pour les contraintes sur les nombres à virgule flottante. Les performances de ce prototype sont évaluées sur un ensemble de programmes communs à la communauté d'analyse de programmes sur les nombres à virgule flottante. Les bornes produites par FErA sont également comparées à plusieurs autres outils de l'état de l'art. Ces expérimentations illustrent la validité de l'encadrement de l'erreur maximale produit et l'efficacité de notre algorithme par rapport à d'autres approches.

8.2 Perspectives

Cette thèse ouvre plusieurs perspectives de recherches. En pratique, une meilleure analyse et un modèle plus précis de l'erreur d'arrondi permettraient de réduire l'effet négatif du problème de dépendance lors du filtrage de l'erreur. En effet, le modèle existant montre certaines limites pour l'encadrement de l'erreur maximale. Dans un premier temps, une analyse de l'erreur pour les opérations arithmétiques, similaire à celle proposée pour les valeurs dans [Gallois-Wong *et al.*, 2020], donnerait une borne plus précise pour certains cas. Ceci nécessiterait également une analyse fine de la distribution de l'erreur pour les opérations arithmétiques de base.

La recherche locale utilisée pour améliorer la borne inférieure de l'encadrement de l'erreur maximale est un élément essentiel de notre algorithme. Le paramétrage de cette recherche locale impacte directement le temps global de résolution. Il est donc essentiel de pouvoir guider au mieux cette recherche afin de trouver une plus grande erreur locale pouvant servir de borne inférieure. Cette amélioration pourrait être guidée par la distribution de l'erreur, comme dans [Xia *et al.*, 2020]. Le paramétrage précis de la recherche locale serait aussi amélioré par une analyse empirique de son comportement lors de la résolution.

Une autre direction de recherche serait d'ajouter le support pour les fonctions transcendantes, *e.g.,* $\sin$, $\cos$, ou encore $\tan$, afin de pouvoir traiter plus de problèmes. Néanmoins, l'ajout de ces fonctions vient au prix de l'exactitude du calcul de l'erreur. En effet, ces fonctions ne sont pas calculables dans $\mathbb{Q}$. Il est toutefois possible d'écrire des fonctions de projections dédiées à ces fonctions sur les intervalles en introduisant une sur-approximation supplémentaire. De plus, le manque des propriétés pour ces fonctions au niveau du standard IEEE 754 [IEEE, 2008] rend d'autant plus dépendante une approche théorique à l'implémentation des fonctions transcendantes. Au niveau de l'implémentation, de telles fonctions de projections ne peuvent donc pas être calculée dans $\mathbb{Q}$. Une alternative serait d'utiliser des nombres à virgule flottante dans une précision

arbitrairement *grande*, comme dans MPFR [Fousse *et al.*, 2007], ainsi que des intervalles. Cette approche produit une sur-approximation des erreurs mais reste conservatrice des solutions grâce à un arrondi extérieur au bornes de l'intervalle.

Finalement, une expérimentation approfondie avec différentes stratégies de recherches, comme celles introduites dans [Zitoun, 2018], servirait à améliorer la résolution de problème. De nouvelles stratégies de recherche, dédiées aux erreurs, pourraient également être utiles à l'accélération du processus de résolution. Ces stratégies pourraient s'inspirer de la distribution des erreurs et mettre en avant certaines propriétés liant les domaines de valeurs et les domaines d'erreurs des variables du problème.

Mes publications

[Garcia *et al.*, 2018a] Garcia, R., Michel, C., Pelleau, M. et Rueher, M. (2018a). Towards a constraint system for round-off error analysis of floating-point computation. *In 24th International Conference on Principles and Practice of Constraint Programming : Doctoral Program*, Lille, France.

[Garcia *et al.*, 2018b] Garcia, R., Michel, C., Pelleau, M. et Rueher, M. (2018b). Vers un système de contraintes pour l'analyse des erreurs de précision des calculs sur les flottants. *In JFPC 2018 - Actes des 14es Journees Francophones de Programmation par Contraintes, Amiens, France*, page 55.

[Garcia *et al.*, 2020a] Garcia, R., Michel, C. et Rueher, M. (2020a). A Branch-and-bound Algorithm to Rigorously Enclose the Round-Off Errors. *In* Simonis, H., éditeur : *Principles and Practice of Constraint Programming - 26th International Conference, CP 2020, Louvain-la-Neuve, Belgium, September 7-11, 2020, Proceedings*, volume 12333 de *Lecture Notes in Computer Science*, pages 637–653. Springer.

[Garcia *et al.*, 2020b] Garcia, R., Michel, C. et Rueher, M. (2020b). Rigorous Enclosure of Round-Off Errors in Floating-Point Computations. *In* Christakis, M., Polikarpova, N., Duggirala, P. S. et Schrammel, P., éditeurs : *Software Verification - 12th International Conference, VSTTE 2020, and 13th International Workshop, NSV 2020, Los Angeles, CA, USA, July 20-21, 2020, Revised Selected Papers*, volume 12549 de *Lecture Notes in Computer Science*, pages 196–212. Springer.

Bibliographie

[Aggoun et Beldiceanu, 1992] Aggoun, A. et Beldiceanu, N. (1992). Extending CHIP in order to solve complex scheduling and placement problems. *In* Delahaye, J., Devienne, P., Mathieu, P. et Yim, P., éditeurs : *JFPL'92, 1ères Journées Francophones de Programmation Logique, 25-27 Mai 1992, Lille, France*, page 51.

[Andrlon *et al.*, 2019] Andrlon, M., Schachte, P., Søndergaard, H. et Stuckey, P. J. (2019). Optimal Bounds for Floating-Point Addition in Constant Time. *In* Takagi, N., Boldo, S. et Langhammer, M., éditeurs : *26th IEEE Symposium on Computer Arithmetic, ARITH 2019, Kyoto, Japan, June 10-12, 2019*, pages 159–166. IEEE.

[Apt, 1999] Apt, K. R. (1999). The Rough Guide to Constraint Propagation. *In* Jaffar, J., éditeur : *Principles and Practice of Constraint Programming - CP'99, 5th International Conference, Alexandria, Virginia, USA, October 11-14, 1999, Proceedings*, volume 1713 de *Lecture Notes in Computer Science*, pages 1–23. Springer.

[Bäck *et al.*, 2000] Bäck, T., Fogel, D. B. et Michalewicz, Z. (2000). *Evolutionary computation 1 : Basic algorithms and operators*. CRC press.

[Baptiste *et al.*, 2012] Baptiste, P., Le Pape, C. et Nuijten, W. (2012). *Constraint-based scheduling : applying constraint programming to scheduling problems*, volume 39. Springer Science & Business Media.

[Baranowski et Briggs, 2017] Baranowski, M. S. et Briggs, I. (2017). Gelpia : A Global Optimizer for Real Functions.

[Becker *et al.*, 2018] Becker, H., Panchekha, P., Darulova, E. et Tatlock, Z. (2018). Combining Tools for Optimization and Analysis of Floating-Point Computations. *In* Havelund, K., Peleska, J., Roscoe, B. et de Vink, E. P., éditeurs : *Formal Methods - 22nd International Symposium, FM 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 15-17, 2018, Proceedings*, volume 10951 de *Lecture Notes in Computer Science*, pages 355–363. Springer.

[Benhamou, 1996] Benhamou, F. (1996). Heterogeneous Constraint Solving. *In* Hanus, M. et Rodríguez-Artalejo, M., éditeurs : *Algebraic and Logic Programming, 5th International Conference, ALP'96, Aachen, Germany, September 25-27, 1996, Proceedings*, volume 1139 de *Lecture Notes in Computer Science*, pages 62–76. Springer.

[Benhamou *et al.*, 1994] Benhamou, F., McAllester, D. A. et Hentenryck, P. V. (1994). CLP(Intervals) Revisited. *In* Bruynooghe, M., éditeur : *Logic Programming, Proceedings of the 1994 International Symposium, Ithaca, New York, USA, November 13-17, 1994*, pages 124–138. MIT Press.

[Benhamou et Older, 1997] Benhamou, F. et Older, W. J. (1997). Applying Interval Arithmetic to Real, Integer, and Boolean Constraints. *J. Log. Program.*, 32(1):1–24.

[Bergman *et al.*, 2014] Bergman, D., Ciré, A. A., Sabharwal, A., Samulowitz, H., Saraswat, V. A. et van Hoeve, W. J. (2014). Parallel Combinatorial Optimization with Decision Diagrams. *In* Simonis, H., éditeur : *Integration of AI and OR Techniques in Constraint Programming -*

11th International Conference, CPAIOR 2014, Cork, Ireland, May 19-23, 2014. Proceedings, volume 8451 de *Lecture Notes in Computer Science*, pages 351–367. Springer.

[Bessière, 1994] Bessière, C. (1994). Arc-Consistency and Arc-Consistency Again. *Artif. Intell.*, 65(1):179–190.

[Bessière et Régin, 1996] Bessière, C. et Régin, J. (1996). MAC and Combined Heuristics : Two Reasons to Forsake FC (and CBJ ?) on Hard Problems. *In* Freuder, E. C., éditeur : *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming, Cambridge, Massachusetts, USA, August 19-22, 1996*, volume 1118 de *Lecture Notes in Computer Science*, pages 61–75. Springer.

[Bessière et Régin, 2001] Bessière, C. et Régin, J. (2001). Refining the Basic Constraint Propagation Algorithm. *In* Nebel, B., éditeur : *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001, Seattle, Washington, USA, August 4-10, 2001*, pages 309–315. Morgan Kaufmann.

[Biere *et al.*, 2009] Biere, A., Heule, M., van Maaren, H. et Walsh, T., éditeurs (2009). *Handbook of Satisfiability*, volume 185 de *Frontiers in Artificial Intelligence and Applications*. IOS Press.

[Boldo *et al.*, 2013] Boldo, S., Clément, F., Filliâtre, J., Mayero, M., Melquiond, G. et Weis, P. (2013). Wave Equation Numerical Resolution : A Comprehensive Mechanized Proof of a C Program. *J. Autom. Reason.*, 50(4):423–456.

[Boldo *et al.*, 2009] Boldo, S., Filliâtre, J. et Melquiond, G. (2009). Combining Coq and Gappa for Certifying Floating-Point Programs. *In* Carette, J., Dixon, L., Coen, C. S. et Watt, S. M., éditeurs : *Intelligent Computer Mathematics, 16th Symposium, Calculemus 2009, 8th International Conference, MKM 2009, Held as Part of CICM 2009, Grand Bend, Canada, July 6-12, 2009. Proceedings*, volume 5625 de *Lecture Notes in Computer Science*, pages 59–74. Springer.

[Boldo *et al.*, 2015] Boldo, S., Jourdan, J., Leroy, X. et Melquiond, G. (2015). Verified Compilation of Floating-Point Computations. *J. Autom. Reason.*, 54(2):135–163.

[Boldo et Melquiond, 2011] Boldo, S. et Melquiond, G. (2011). Flocq : A Unified Library for Proving Floating-Point Algorithms in Coq. *In* Antelo, E., Hough, D. et Ienne, P., éditeurs : *20th IEEE Symposium on Computer Arithmetic, ARITH 2011, Tübingen, Germany, 25-27 July 2011*, pages 243–252. IEEE Computer Society.

[Botella *et al.*, 2006] Botella, B., Gotlieb, A. et Michel, C. (2006). Symbolic execution of floating-point computations. *Software Testing, Verification and Reliability*, 16(2):97–121.

[Boussemart *et al.*, 2004] Boussemart, F., Hemery, F., Lecoutre, C. et Sais, L. (2004). Boosting Systematic Search by Weighting Constraints. *In* de Mántaras, R. L. et Saitta, L., éditeurs : *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*, pages 146–150. IOS Press.

[Brain *et al.*, 2015] Brain, M., Tinelli, C., Rümmer, P. et Wahl, T. (2015). An Automatable Formal Semantics for IEEE-754 Floating-Point Arithmetic. *In 22nd IEEE Symposium on Computer Arithmetic, ARITH 2015, Lyon, France, June 22-24, 2015*, pages 160–167. IEEE.

[Brélaz, 1979] Brélaz, D. (1979). New Methods to Color the Vertices of a Graph. *Commun. ACM*, 22(4):251–256.

[Bureau d'Enquêtes et d'Analyse, 2012] Bureau d'Enquêtes et d'Analyse (2012). Rapport final-Accident survenu le 1er juin 2009 à l'Airbus A330-203 immatriculé F-GZCP exploité par Air France vol AF 447 Rio de Janeiro-Paris. *Bureau d'Enquête et d'Analyse, Paris*.

[Chen *et al.*, 2008] Chen, L., Miné, A. et Cousot, P. (2008). A Sound Floating-Point Polyhedra Abstract Domain. *In* Ramalingam, G., éditeur : *Programming Languages and Systems, 6th Asian Symposium, APLAS 2008, Bangalore, India, December 9-11, 2008. Proceedings*, volume 5356 de *Lecture Notes in Computer Science*, pages 3–18. Springer.

[Chiang *et al.*, 2014] Chiang, W., Gopalakrishnan, G., Rakamaric, Z. et Solovyev, A. (2014). Efficient search for inputs causing high floating-point errors. *In* Moreira, J. E. et Larus, J. R., éditeurs : *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*, pages 43–52. ACM.

[Collavizza *et al.*, 1998] Collavizza, H., Delobel, F. et Rueher, M. (1998). A Note on Partial Consistencies over Continuous Domains. *In* Maher, M. J. et Puget, J., éditeurs : *Principles and Practice of Constraint Programming - CP98, 4th International Conference, Pisa, Italy, October 26-30, 1998, Proceedings*, volume 1520 de *Lecture Notes in Computer Science*, pages 147–161. Springer.

[Collavizza *et al.*, 2016] Collavizza, H., Michel, C. et Rueher, M. (2016). Searching Critical Values for Floating-Point Programs. *In* Wotawa, F., Nica, M. et Kushik, N., éditeurs : *Testing Software and Systems - 28th IFIP WG 6.1 International Conference, ICTSS 2016, Graz, Austria, October 17-19, 2016, Proceedings*, volume 9976 de *Lecture Notes in Computer Science*, pages 209–217.

[Collavizza et Rueher, 2007] Collavizza, H. et Rueher, M. (2007). Exploring Different Constraint-Based Modelings for Program Verification. *In* Bessiere, C., éditeur : *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, volume 4741 de *Lecture Notes in Computer Science*, pages 49–63. Springer.

[Collavizza *et al.*, 2010] Collavizza, H., Rueher, M. et Hentenryck, P. V. (2010). CPBPV : a constraint-programming framework for bounded program verification. *Constraints An Int. J.*, 15(2):238–264.

[Cousot et Cousot, 1977a] Cousot, P. et Cousot, R. (1977a). Abstract Interpretation : A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. *In Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA. ACM.

[Cousot et Cousot, 1977b] Cousot, P. et Cousot, R. (1977b). Static Determination of Dynamic Properties of Generalized Type Unions. *SIGPLAN Not.*, 12(3):77–94.

[Cousot *et al.*, 2006] Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D. et Rival, X. (2006). Combination of Abstractions in the ASTRÉE Static Analyzer. *In* Okada, M. et Satoh, I., éditeurs : *Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues, 11th Asian Computing Science Conference, Tokyo, Japan, December 6-8, 2006, Revised Selected Papers*, volume 4435 de *Lecture Notes in Computer Science*, pages 272–300. Springer.

[Cousot et Halbwachs, 1978] Cousot, P. et Halbwachs, N. (1978). Automatic Discovery of Linear Restraints Among Variables of a Program. *In* Aho, A. V., Zilles, S. N. et Szymanski, T. G., éditeurs : *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, pages 84–96. ACM Press.

[Damouche *et al.*, 2017a] Damouche, N., Martel, M. et Chapoutot, A. (2017a). Improving the numerical accuracy of programs by automatic transformation. *Int. J. Softw. Tools Technol. Transf.*, 19(4):427–448.

[Damouche *et al.*, 2017b] Damouche, N., Martel, M., Panchekha, P., Qiu, C., Sanchez-Stern, A. et Tatlock, Z. (2017b). Toward a Standard Benchmark Format and Suite for Floating-Point Analysis. *In 9th International Workshop on Numerical Software Verification (NSV2017)*, pages 63–77.

[Darulova *et al.*, 2018a] Darulova, E., Horn, E. et Sharma, S. (2018a). Sound mixed-precision optimization with rewriting. *In* Gill, C., Sinopoli, B., Liu, X. et Tabuada, P., éditeurs : *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems, ICCPS 2018, Porto, Portugal, April 11-13, 2018*, pages 208–219. IEEE Computer Society / ACM.

[Darulova *et al.*, 2018b] Darulova, E., Izycheva, A., Nasir, F., Ritter, F., Becker, H. et Bastian, R. (2018b). Daisy - Framework for Analysis and Optimization of Numerical Programs (Tool Paper). *In* Beyer, D. et Huisman, M., éditeurs : *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I*, volume 10805 de *Lecture Notes in Computer Science*, pages 270–287. Springer.

[Darulova et Kuncak, 2011] Darulova, E. et Kuncak, V. (2011). Trustworthy numerical computation in Scala. *In* Lopes, C. V. et Fisher, K., éditeurs : *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, pages 325–344. ACM.

[Darulova et Kuncak, 2014] Darulova, E. et Kuncak, V. (2014). Sound compilation of reals. *In The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 235–248. ACM.

[Darulova et Kuncak, 2017] Darulova, E. et Kuncak, V. (2017). Towards a Compiler for Reals. *ACM Trans. Program. Lang. Syst.*, 39(2):8 :1–8 :28.

[Darulova et Volkova, 2019] Darulova, E. et Volkova, A. (2019). Sound Approximation of Programs with Elementary Functions. *In* Dillig, I. et Tasiran, S., éditeurs : *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II*, volume 11562 de *Lecture Notes in Computer Science*, pages 174–183. Springer.

[Das *et al.*, 2020] Das, A., Briggs, I., Gopalakrishnan, G., Krishnamoorthy, S. et Panchekha, P. (2020). Scalable yet rigorous floating-point error analysis. *In* Cuicchi, C., Qualters, I. et Kramer, W. T., éditeurs : *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, page 51. IEEE/ACM.

[Daumas et Melquiond, 2010] Daumas, M. et Melquiond, G. (2010). Certification of bounds on expressions involving rounded operators. *ACM Trans. Math. Softw.*, 37(1):2 :1–2 :20.

[de Figueiredo et Stolfi, 2004] de Figueiredo, L. H. et Stolfi, J. (2004). Affine Arithmetic : Concepts and Applications. *Numer. Algorithms*, 37(1-4):147–158.

[de Moura et Bjørner, 2008] de Moura, L. M. et Bjørner, N. (2008). Z3 : An Efficient SMT Solver. *In* Ramakrishnan, C. R. et Rehof, J., éditeurs : *Tools and Algorithms for the Construction*

and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings, volume 4963 de *Lecture Notes in Computer Science*, pages 337–340. Springer.

[Denmat *et al.*, 2005] Denmat, T., Gotlieb, A. et Ducassé, M. (2005). Proving or Disproving likely Invariants with Constraint Reasoning. *In* Serebrenik, A. et Muñoz-Hernández, S., éditeurs : *Proceedings of the 15th International Workshop on Logic Programming Environments, Sitges (Barcelona), Spain, October 5, 2005*, pages 1–13.

[D'Silva *et al.*, 2012] D'Silva, V., Haller, L., Kroening, D. et Tautschnig, M. (2012). Numeric Bounds Analysis with Conflict-Driven Learning. *In* Flanagan, C. et König, B., éditeurs : *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7214 de *Lecture Notes in Computer Science*, pages 48–63. Springer.

[Fousse *et al.*, 2007] Fousse, L., Hanrot, G., Lefèvre, V., Pélissier, P. et Zimmermann, P. (2007). MPFR : A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.*, 33(2):13.

[Fu *et al.*, 2015] Fu, Z., Bai, Z. et Su, Z. (2015). Automated backward error analysis for numerical code. *In* Aldrich, J. et Eugster, P., éditeurs : *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 639–654. ACM.

[Gadelha *et al.*, 2018] Gadelha, M. Y. R., Monteiro, F. R., Morse, J., Cordeiro, L. C., Fischer, B. et Nicole, D. A. (2018). ESBMC 5.0 : an industrial-strength C model checker. *In* Huchard, M., Kästner, C. et Fraser, G., éditeurs : *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 888–891. ACM.

[Galassi *et al.*, 2009] Galassi, M., Davies, J., Theiler, J., Gough, B. et Jungman, G. (2009). *GNU Scientific Library - Reference Manual, Third Edition, for GSL Version 1.12*. Network Theory Ltd.

[Gallois-Wong *et al.*, 2020] Gallois-Wong, D., Boldo, S. et Cuoq, P. (2020). Optimal inverse projection of floating-point addition. *Numer. Algorithms*, 83(3):957–986.

[Gao *et al.*, 2013] Gao, S., Kong, S. et Clarke, E. M. (2013). dReal : An SMT Solver for Nonlinear Theories over the Reals. *In* Bonacina, M. P., éditeur : *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, volume 7898 de *Lecture Notes in Computer Science*, pages 208–214. Springer.

[García-Martín *et al.*, 2013] García-Martín, J. A., Clote, P. et Dotú, I. (2013). Rnaifold : a Constraint Programming Algorithm for RNA inverse Folding and molecular Design. *J. Bioinform. Comput. Biol.*, 11(2).

[General Accounting Office, 1992] General Accounting Office, U. S. (1992). Patriot Missile Defense : Software Problem Led to System Failure at Dhahran, Saudi Arabia.

[Geuvers, 2009] Geuvers, H. (2009). Proof assistants : History, ideas and future. *Sadhana*, 34(1): 3–25.

[Ghorbal *et al.*, 2009] Ghorbal, K., Goubault, E. et Putot, S. (2009). The Zonotope Abstract Domain Taylor1+. *In Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 de *Lecture Notes in Computer Science*, pages 627–633.

[Ghorbal *et al.*, 2010] Ghorbal, K., Goubault, E. et Putot, S. (2010). A Logical Product Approach to Zonotope Intersection. *In Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19,*, volume 6174 de *LNCS*, pages 212–226.

[Goldberg, 1991] Goldberg, D. (1991). What Every Computer Scientist Should Know About Floating-Point Arithmetic. *ACM Comput. Surv.*, 23(1):5–48.

[Golomb et Baumert, 1965] Golomb, S. W. et Baumert, L. D. (1965). Backtrack Programming. *J. ACM*, 12(4):516–524.

[Goodloe *et al.*, 2013] Goodloe, A., Muñoz, C. A., Kirchner, F. et Correnson, L. (2013). Verification of Numerical Programs : From Real Numbers to Floating Point Numbers. *In* Brat, G., Rungta, N. et Venet, A., éditeurs : *NASA Formal Methods, 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings*, volume 7871 de *Lecture Notes in Computer Science*, pages 441–446. Springer.

[Gotlieb et Botella, 2003] Gotlieb, A. et Botella, B. (2003). Automated Metamorphic Testing. *In 27th International Computer Software and Applications Conference (COMPSAC 2003) : Design and Assessment of Trustworthy Software-Based Systems, 3-6 November 2003, Dallas, TX, USA, Proceedings*, pages 34–40. IEEE Computer Society.

[Gotlieb *et al.*, 2000] Gotlieb, A., Botella, B. et Rueher, M. (2000). A CLP Framework for Computing Structural Test Data. *In* Lloyd, J. W., Dahl, V., Furbach, U., Kerber, M., Lau, K., Palamidessi, C., Pereira, L. M., Sagiv, Y. et Stuckey, P. J., éditeurs : *Computational Logic - CL 2000, First International Conference, London, UK, 24-28 July, 2000, Proceedings*, volume 1861 de *Lecture Notes in Computer Science*, pages 399–413. Springer.

[Goualard, 2017] Goualard, F. (2017). GAOL : Not Just Another Interval Library.

[Goubault et Putot, 2006] Goubault, E. et Putot, S. (2006). Static Analysis of Numerical Algorithms. *In Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings*, volume 4134 de *Lecture Notes in Computer Science*, pages 18–34.

[Goubault et Putot, 2011] Goubault, E. et Putot, S. (2011). Static Analysis of Finite Precision Computations. *In 12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2011)*, pages 232–247.

[Graillat *et al.*, 2019] Graillat, S., Jézéquel, F., Picot, R., Févotte, F. et Lathuilière, B. (2019). Auto-tuning for floating-point precision with Discrete Stochastic Arithmetic. *J. Comput. Sci.*, 36.

[Granlund et the GMP development team, 2016] Granlund, T. et the GMP development team (2016). GNU MP : The GNU Multiple Precision Arithmetic Library.

[Haralick et Elliott, 1979] Haralick, R. M. et Elliott, G. L. (1979). Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *In* Buchanan, B. G., éditeur : *Proceedings of the Sixth International Joint Conference on Artificial Intelligence, IJCAI 79, Tokyo, Japan, August 20-23, 1979, 2 Volumes*, pages 356–364. William Kaufmann.

[Harrison, 1999] Harrison, J. (1999). A Machine-Checked Theory of Floating Point Arithmetic. *In* Bertot, Y., Dowek, G., Hirschowitz, A., Paulin-Mohring, C. et Théry, L., éditeurs : *Theorem*

Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, Nice, France, September, 1999, Proceedings, volume 1690 de *Lecture Notes in Computer Science*, pages 113–130. Springer.

[Harrison, 2006] Harrison, J. (2006). Floating-Point Verification Using Theorem Proving. *In* Bernardo, M. et Cimatti, A., éditeurs : *Formal Methods for Hardware Verification, 6th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2006, Bertinoro, Italy, May 22-27, 2006, Advanced Lectures*, volume 3965 de *Lecture Notes in Computer Science*, pages 211–242. Springer.

[Harvey, 1995] Harvey, W. D. (1995). *Nonsystematic Backtracking Search*. Thèse de doctorat, Stanford, CA, USA.

[Hauser, 1996] Hauser, J. R. (1996). Handling Floating-point Exceptions in Numeric Programs. *ACM Trans. Program. Lang. Syst.*, 18(2):139–174.

[Hentenryck et Michel, 2006] Hentenryck, P. V. et Michel, L. (2006). Nondeterministic Control for Hybrid Search. *Constraints An Int. J.*, 11(4):353–373.

[Hentenryck et Michel, 2013] Hentenryck, P. V. et Michel, L. (2013). The Objective-CP Optimization System. *In 19th International Conference on Principles and Practice of Constraint Programming (CP 2013)*, pages 8–29.

[Hentenryck *et al.*, 1994] Hentenryck, P. V., Saraswat, V. A. et Deville, Y. (1994). Design, Implementation, and Evaluation of the Constraint Language cc(FD). *In* Podelski, A., éditeur : *Constraint Programming : Basics and Trends, Châtillon Spring School, Châtillon-sur-Seine, France, May 16 - 20, 1994, Selected Papers*, volume 910 de *Lecture Notes in Computer Science*, pages 293–316. Springer.

[Higham, 2002] Higham, N. J. (2002). *Accuracy and stability of numerical algorithms, Second Edition*. SIAM.

[IEEE, 2008] IEEE (2008). *754-2008 - IEEE Standard for floating point arithmethic*.

[Izycheva et Darulova, 2017] Izycheva, A. et Darulova, E. (2017). On sound relative error bounds for floating-point arithmetic. *In* Stewart, D. et Weissenbacher, G., éditeurs : *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*, pages 15–22. IEEE.

[Jacquemin *et al.*, 2018] Jacquemin, M., Putot, S. et Védrine, F. (2018). A Reduced Product of Absolute and Relative Error Bounds for Floating-Point Analysis. *In* Podelski, A., éditeur : *Static Analysis - 25th International Symposium, SAS 2018, Freiburg, Germany, August 29-31, 2018, Proceedings*, volume 11002 de *Lecture Notes in Computer Science*, pages 223–242. Springer.

[Jeannerod, 2015] Jeannerod, C. (2015). Exploiting Structure in Floating-Point Arithmetic. *In* Kotsireas, I. S., Rump, S. M. et Yap, C. K., éditeurs : *Mathematical Aspects of Computer and Information Sciences - 6th International Conference, MACIS 2015, Berlin, Germany, November 11-13, 2015, Revised Selected Papers*, volume 9582 de *Lecture Notes in Computer Science*, pages 25–34. Springer.

[Jézéquel et Chesneaux, 2008] Jézéquel, F. et Chesneaux, J. M. (2008). CADNA : a library for estimating round-off error propagation. *Comput. Phys. Commun.*, 178(12):933–955.

[Johnson, 2017] Johnson, S. G. (2017). The NLopt nonlinear-optimization package.

[Kahan, 2004] Kahan, W. (2004). A logarithm too clever by half.

[Khedker *et al.*, 2009] Khedker, U. P., Sanyal, A. et Sathe, B. (2009). *Data Flow Analysis - Theory and Practice*. CRC Press.

[Kirchner *et al.*, 2015] Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J. et Yakobowski, B. (2015). Frama-C : A software analysis perspective. *Formal Aspects Comput.*, 27(3):573–609.

[Köster *et al.*, 2017] Köster, U., Webb, T., Wang, X., Nassar, M., Bansal, A. K., Constable, W., Elibol, O., Hall, S., Hornof, L., Khosrowshahi, A., Kloss, C., Pai, R. J. et Rao, N. (2017). Flexpoint : An Adaptive Numerical Format for Efficient Training of Deep Neural Networks. *In* Guyon, I., von Luxburg, U., Bengio, S., Wallach, H. M., Fergus, R., Vishwanathan, S. V. N. et Garnett, R., éditeurs : *Advances in Neural Information Processing Systems 30 : Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, pages 1742–1752.

[Lasserre, 2006] Lasserre, J. B. (2006). Convergent SDP-Relaxations in Polynomial Optimization with Sparsity. *SIAM J. Optim.*, 17(3):822–843.

[Lasserre, 2011] Lasserre, J. B. (2011). A New Look at Nonnegativity on Closed Sets and Polynomial Optimization. *SIAM J. Optim.*, 21(3):864–885.

[Lebbah *et al.*, 2002] Lebbah, Y., Rueher, M. et Michel, C. (2002). A Global Filtering Algorithm for Handling Systems of Quadratic Equations and Inequations. *In* Hentenryck, P. V., éditeur : *Principles and Practice of Constraint Programming - CP 2002, 8th International Conference, CP 2002, Ithaca, NY, USA, September 9-13, 2002, Proceedings*, volume 2470 de *Lecture Notes in Computer Science*, pages 109–123. Springer.

[Lee, 2014] Lee, E. A. (2014). Constructive Models of Discrete and Continuous Physical Phenomena. *IEEE Access*, 2:797–821.

[Lhomme, 1993] Lhomme, O. (1993). Consistency Techniques for Numeric CSPs. *In* Bajcsy, R., éditeur : *Proceedings of the 13th International Joint Conference on Artificial Intelligence. Chambéry, France, August 28 - September 3, 1993*, pages 232–238. Morgan Kaufmann.

[Lions *et al.*, 1996] Lions, J.-L. *et al.* (1996). Flight 501 failure. *Report by the Inquiry Board*, 190.

[Lorentz, 1986] Lorentz, G. (1986). *Bernstein Polynomials*. AMS Chelsea Publishing Series. Chelsea Publishing Company.

[Mackworth, 1977] Mackworth, A. K. (1977). Consistency in networks of relations. *Journal of Artificial Intelligence*, pages 8(1) :99–118.

[Magron, 2018] Magron, V. (2018). Interval Enclosures of Upper Bounds of Roundoff Errors Using Semidefinite Programming. *ACM Trans. Math. Softw.*, 44(4):41 :1–41 :18.

[Magron *et al.*, 2017] Magron, V., Constantinides, G. A. et Donaldson, A. F. (2017). Certified Roundoff Error Bounds Using Semidefinite Programming. *ACM Trans. Math. Softw.*, 43(4): 34 :1–34 :31.

[Marre et Michel, 2010] Marre, B. et Michel, C. (2010). Improving the floating point addition and subtraction constraints. *In Proceedings of the 16th international conference on Principles and practice of constraint programming (CP'10)*, LNCS 6308, pages 360–367, St. Andrews, Scotland.

[Martí *et al.*, 2018] Martí, R., Pardalos, P. M. et Resende, M. G. C., éditeurs (2018). *Handbook of Heuristics*. Springer.

[Melquiond, 2012] Melquiond, G. (2012). Floating-point arithmetic in the Coq system. *Inf. Comput.*, 216:14–23.

[Meudec, 2001] Meudec, C. (2001). ATGen : automatic test data generation using constraint logic programming and symbolic execution. *Softw. Test. Verification Reliab.*, 11(2):81–96.

[Michel, 2002] Michel, C. (2002). Exact projection functions for floating point number constraints. *In AI&M 1-2002, Seventh international symposium on Artificial Intelligence and Mathematics (7th ISAIM)*, Fort Lauderdale, Floride (US).

[Michel *et al.*, 2001] Michel, C., Rueher, M. et Lebbah, Y. (2001). Solving Constraints over Floating-Point Numbers. *In 7th International Conference on Principles and Practice of Constraint Programming (CP 2001)*, pages 524–538.

[Miné, 2004] Miné, A. (2004). Relational Abstract Domains for the Detection of Floating-Point Run-Time Errors. *In* Schmidt, D. A., éditeur : *Programming Languages and Systems, 13th European Symposium on Programming, ESOP 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, volume 2986 de *Lecture Notes in Computer Science*, pages 3–17. Springer.

[Miné, 2006] Miné, A. (2006). The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100.

[Mohr et Henderson, 1986] Mohr, R. et Henderson, T. C. (1986). Arc and Path Consistency Revisited. *Artif. Intell.*, 28(2):225–233.

[Montanari, 1974] Montanari, U. (1974). Networks of constraints : Fundamental properties and applications to picture processing. *Inf. Sci.*, 7:95–132.

[Moore *et al.*, 2009] Moore, R. E., Kearfott, R. B. et Cloud, M. J. (2009). *Introduction to Interval Analysis*. SIAM.

[Moscato *et al.*, 2017] Moscato, M., Titolo, L., Dutle, A. et Muñoz, C. A. (2017). Automatic Estimation of Verified Floating-Point Round-Off Errors via Static Analysis. *In Computer Safety, Reliability, and Security*, pages 213–229.

[Muller *et al.*, 2018] Muller, J., Brunie, N., de Dinechin, F., Jeannerod, C., Joldes, M., Lefèvre, V., Melquiond, G., Revol, N. et Torres, S. (2018). *Handbook of Floating-Point Arithmetic (2nd Ed.)*. Springer.

[Muller, 2005] Muller, J.-M. (2005). On the definition of ulp(x). Research Report RR-5504, LIP RR-2005-09, INRIA, LIP.

[Myers, 2004] Myers, G. J. (2004). *The art of software testing (2. ed.)*. Wiley.

[Narkawicz et Muñoz, 2013] Narkawicz, A. et Muñoz, C. A. (2013). A Formally Verified Generic Branching Algorithm for Global Optimization. *In* Cohen, E. et Rybalchenko, A., éditeurs : *Verified Software : Theories, Tools, Experiments - 5th International Conference, VSTTE 2013, Menlo Park, CA, USA, May 17-19, 2013, Revised Selected Papers*, volume 8164 de *Lecture Notes in Computer Science*, pages 326–343. Springer.

[Owre *et al.*, 1992] Owre, S., Rushby, J. M. et Shankar, N. (1992). PVS : A Prototype Verification System. *In* Kapur, D., éditeur : *Automated Deduction - CADE-11, 11th International Conference on Automated Deduction, Saratoga Springs, NY, USA, June 15-18, 1992, Proceedings*, volume 607 de *Lecture Notes in Computer Science*, pages 748–752. Springer.

[Panchekha *et al.*, 2015] Panchekha, P., Sanchez-Stern, A., Wilcox, J. R. et Tatlock, Z. (2015). Automatically improving accuracy for floating point expressions. *In* Grove, D. et Blackburn, S. M., éditeurs : *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 1–11. ACM.

[Parrilo et Thomas, 2020] Parrilo, P. A. et Thomas, R. R. (2020). *Sum of Squares : Theory and Applications*, volume 77 de *Proceedings of symposia in applied mathematics*. American Mathematical Society.

[Perron, 1999] Perron, L. (1999). Search Procedures and Parallelism in Constraint Programming. *In* Jaffar, J., éditeur : *Principles and Practice of Constraint Programming - CP'99, 5th International Conference, Alexandria, Virginia, USA, October 11-14, 1999, Proceedings*, volume 1713 de *Lecture Notes in Computer Science*, pages 346–360. Springer.

[Ponsini *et al.*, 2016] Ponsini, O., Michel, C. et Rueher, M. (2016). Verifying floating-point programs with constraint programming and abstract interpretation techniques. *Automated Software Engineering*, 23(2):191–217.

[Quinn, 1983] Quinn, K. (1983). Ever Had Problems Rounding Off Figures ? This Stock Exchange Has. *The Wall Street Journal*, page 37.

[Rabbouch *et al.*, 2019] Rabbouch, B., Saâdaoui, F. et Mraihi, R. (2019). Constraint Programming Based Algorithm for Solving Large-Scale Vehicle Routing Problems. *In* García, H. P., Sánchez-González, L., Limas, M. C., Quintián-Pardo, H. et Rodríguez, E. S. C., éditeurs : *Hybrid Artificial Intelligent Systems - 14th International Conference, HAIS 2019, León, Spain, September 4-6, 2019, Proceedings*, volume 11734 de *Lecture Notes in Computer Science*, pages 526–539. Springer.

[Ratz, 1994] Ratz, D. (1994). Box-Splitting strategies for the interval Gauss-Seidel step in a global optimization method. *Computing*, 53(3-4):337–353.

[Régin, 1994] Régin, J. (1994). A Filtering Algorithm for Constraints of Difference in CSPs. *In* Hayes-Roth, B. et Korf, R. E., éditeurs : *Proceedings of the 12th National Conference on Artificial Intelligence, Seattle, WA, USA, July 31 - August 4, 1994, Volume 1*, pages 362–367. AAAI Press / The MIT Press.

[Régin *et al.*, 2013] Régin, J., Rezgui, M. et Malapert, A. (2013). Embarrassingly Parallel Search. *In* Schulte, C., éditeur : *Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings*, volume 8124 de *Lecture Notes in Computer Science*, pages 596–610. Springer.

[Rice, 1953] Rice, H. G. (1953). Classes of Recursively Enumerable Sets and Their Decision Problems. *Transactions of the American Mathematical Society*, 74(2):358–366.

[Rosen *et al.*, 1988] Rosen, B. K., Wegman, M. N. et Zadeck, F. K. (1988). Global Value Numbers and Redundant Computations. *In* Ferrante, J. et Mager, P., éditeurs : *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, pages 12–27. ACM Press.

[Rump, 1988] Rump, S. (1988). Algorithms for Verified Inclusions : Theory and Practice. *In* Moore, R. E., éditeur : *Reliability in Computing : The Role of Interval Methods in Scientific Computing*, pages 109–126. Academic Press Professional, Inc., San Diego, CA, USA.

[Rump *et al.*, 2008] Rump, S. M., Ogita, T. et Oishi, S. (2008). Accurate Floating-Point Summation Part I : Faithful Rounding. *SIAM J. Sci. Comput.*, 31(1):189–224.

[Schrammel *et al.*, 2017] Schrammel, P., Kroening, D., Brain, M., Martins, R., Teige, T. et Bienmüller, T. (2017). Incremental bounded model checking for embedded software. *Formal Aspects Comput.*, 29(5):911–931.

[Schrijvers *et al.*, 2013] Schrijvers, T., Tack, G., Wuille, P., Samulowitz, H. et Stuckey, P. J. (2013). Search combinators. *Constraints An Int. J.*, 18(2):269–305.

[Slabodkin, 1998] Slabodkin, G. (1998). Software glitches leave Navy Smart Ship dead in the water. https://gcn.com/Articles/1998/07/13/Software-glitches-leave-Navy-Smart-Ship-dead-in-the-water.aspx.

[Smith *et al.*, 2015] Smith, A. P., Muñoz, C. A., Narkawicz, A. J. et Markevicius, M. (2015). A Rigorous Generic Branch and Bound Solver for Nonlinear Problems. *In* Kovács, L., Negru, V., Ida, T., Jebelean, T., Petcu, D., Watt, S. M. et Zaharie, D., éditeurs : *17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2015, Timisoara, Romania, September 21-24, 2015*, pages 71–78. IEEE Computer Society.

[Solovyev *et al.*, 2018] Solovyev, A., Baranowski, M. S., Briggs, I., Jacobsen, C., Rakamarić, Z. et Gopalakrishnan, G. (2018). Rigorous Estimation of Floating-Point Round-Off Errors with Symbolic Taylor Expansions. *ACM Trans. Program. Lang. Syst.*, 41(1):2 :1–2 :39.

[Solovyev *et al.*, 2015] Solovyev, A., Jacobsen, C., Rakamarić, Z. et Gopalakrishnan, G. (2015). Rigorous Estimation of Floating-Point Round-off Errors with Symbolic Taylor Expansions. *In* Bjørner, N. et de Boer, F., éditeurs : *FM 2015 : Formal Methods*, pages 532–550, Cham. Springer International Publishing.

[Stallman et Sussman, 1977] Stallman, R. M. et Sussman, G. J. (1977). Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis. *Artif. Intell.*, 9(2):135–196.

[Sterbenz, 1974] Sterbenz, P. H. (1974). *Floating Point Computation*. Prentice-Hall.

[Sy et Deville, 2003] Sy, N. T. et Deville, Y. (2003). Consistency techniques for interprocedural test data generation. *In* Paakki, J. et Inverardi, P., éditeurs : *Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering 2003 held jointly with 9th European Software Engineering Conference, ESEC/FSE 2003, Helsinki, Finland, September 1-5, 2003*, pages 108–117. ACM.

[The Coq Development Team, 2020] The Coq Development Team (2020). *The Coq proof assistant reference manual*. Version 8.11.2.

[Tinelli, 2012] Tinelli, C. (2012). SMT-Based Model Checking. *In* Goodloe, A. et Person, S., éditeurs : *NASA Formal Methods - 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings*, volume 7226 de *Lecture Notes in Computer Science*, page 1. Springer.

[Titolo *et al.*, 2018] Titolo, L., Feliú, M. A., Moscato, M. M. et Muñoz, C. A. (2018). An Abstract Interpretation Framework for the Round-Off Error Analysis of Floating-Point Programs. *In Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9*, pages 516–537.

[Van Hentenryck *et al.*, 1997] Van Hentenryck, P., McAllester, D. et Kapur, D. (1997). Solving polynomial systems using a branch and prune approach. *SIAM Journal on Numerical Analysis*, 34(2):797–827.

[Vignes, 1993] Vignes, J. (1993). A Stochastic Arithmetic for Reliable Scientific Computation. *Math. Comput. Simul.*, 35(3):233–261.

[Wolkowicz *et al.*, 2000] Wolkowicz, H., Saigal, R. et Vandenberghe, L., éditeurs (2000). *Handbook of Semidefinite Programming*. Springer US.

[Wotawa et Nica, 2007] Wotawa, F. et Nica, M. (2007). Converting Programs into Constraint Satisfaction Problems. *In* Badica, C. et Paprzycki, M., éditeurs : *Advances in Intelligent and Distributed Computing, Proceedings of the 1st International Symposium on Intelligent and Distributed Computing, IDC 2007, Craiova, Romania, October 2007*, volume 78 de *Studies in Computational Intelligence*, pages 228–236. Springer.

[Xia *et al.*, 2020] Xia, Y., Guo, S., Hao, J., Liu, D. et Xu, J. (2020). Error detection of arithmetic expressions. *J. Supercomput.*, 76(1):1–18.

[Yamada, 1998] Yamada, Y. (1998). Floating-Point Number for Automotive Control Systems. Rapport technique, SAE Technical Paper.

[Yunes, 2002] Yunes, T. H. (2002). On the Sum Constraint : Relaxation and Applications. *In* Hentenryck, P. V., éditeur : *Principles and Practice of Constraint Programming - CP 2002, 8th International Conference, CP 2002, Ithaca, NY, USA, September 9-13, 2002, Proceedings*, volume 2470 de *Lecture Notes in Computer Science*, pages 80–92. Springer.

[Zeller, 2009] Zeller, A. (2009). *Why Programs Fail - A Guide to Systematic Debugging, 2nd Edition*. Academic Press.

[Zitoun, 2018] Zitoun, H. (2018). *Search strategies for solving constraint systems over floats for program verification*. Theses, Université Côte d'Azur.

[Zou *et al.*, 2020] Zou, D., Zeng, M., Xiong, Y., Fu, Z., Zhang, L. et Su, Z. (2020). Detecting floating-point errors via atomic conditions. *Proc. ACM Program. Lang.*, 4(POPL):60 :1–60 :27.

Liste des figures

Liste des tableaux

Liste des définitions

Liste des exemples

Listes des algorithmes

Analyse des erreurs d'arrondi sur les nombres à virgule flottante par programmation par contraintes

Rémy GARCIA

Résumé

Les nombres à virgule flottante sont utilisés dans de nombreuses applications pour effectuer des calculs, souvent à l'insu de l'utilisateur. Les modèles mathématiques de ces applications utilisent des nombres réels qui ne sont souvent pas représentables sur un ordinateur. En effet, une représentation binaire finie n'est pas suffisante pour représenter l'ensemble continu et infini des nombres réels. Le problème est que le calcul avec des nombres à virgule flottante introduit souvent une erreur d'arrondi par rapport à son équivalent sur les nombres réels. Connaître l'ordre de grandeur de cette erreur est essentiel afin de comprendre correctement le comportement d'un programme. De nombreux outils en analyse d'erreurs calculent une sur-approximation des erreurs. Ces sur-approximations sont souvent trop grossières pour évaluer efficacement l'impact de l'erreur sur le comportement du programme. D'autres outils calculent une sous-approximation de l'erreur maximale, *i.e.,* la plus grande erreur possible en valeur absolue. Ces sous-approximations sont soit incorrectes, soit inatteignables. Dans cette thèse, nous proposons un système de contraintes capable de capturer et de raisonner sur l'erreur produite par un programme qui effectue des calculs avec des nombres à virgule flottante. Nous proposons également un algorithme afin de chercher l'erreur maximale. Pour cela, notre algorithme calcule à la fois une sur-approximation et une sous-approximation rigoureuses de l'erreur maximale. Une sur-approximation est obtenue à partir du système de contraintes pour les erreurs, tandis qu'une sous-approximation atteignable est produite à l'aide d'une procédure générer-et-tester et d'une recherche locale. Notre algorithme est le premier à combiner à la fois une sur-approximation et une sous-approximation de l'erreur. Nos méthodes sont implémentées dans un solveur, appelé FErA. Les performances sur un ensemble de problèmes communs sont compétitives : l'encadrement rigoureux produit est précis et se compare bien par rapport aux autres outils de l'état de l'art.

Mots-clés : programmation par contraintes, nombres à virgule flottante, erreur d'arrondi, analyse d'erreurs, contraintes sur les erreurs, optimisation

Abstract

Floating-point numbers are used in many applications to perform computations, often without the user's knowledge. The mathematical models of these applications use real numbers that are often not representable on a computer. Indeed, a finite binary representation is not sufficient to represent the continuous and infinite set of real numbers. The problem is that computing with floating-point numbers often introduces a rounding error compared to its equivalent over real numbers. Knowing the order of magnitude of this error is essential in order to correctly understand the behaviour of a program. Many error analysis tools calculate an over-approximation of the errors. These over-approximations are often too coarse to effectively assess the impact of the error on the behaviour of the program. Other tools calculate an under-approximation of the maximum error, *i.e.,* the largest possible error in absolute value. These under-approximations are either incorrect or unreachable. In this thesis, we propose a constraint system capable of capturing and reasoning about the error produced by a program that performs computations with floating-point numbers. We also propose an algorithm to search for the maximum error. For this purpose, our algorithm computes both a rigorous over-approximation and a rigorous under-approximation of the maximum error. An over-approximation is obtained from the constraint system for the errors, while a reachable under-approximation is produced using a generate-and-test procedure and a local search. Our algorithm is the first to combine both an over-approximation and an under-approximation of the error. Our methods are implemented in a solver, called FErA. Performance on a set of common problems is competitive: the rigorous enclosure produced is accurate and compares well with other state-of-the-art tools.

Keywords: constraint programming, floating-point numbers, round-off error, error analysis, constraints over errors, optimization