Programming with Numerical Uncertainties

THÈSE N^{O} 6343 (2014)

PRÉSENTÉE LE 28 NOVEMBRE 2014

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE D'ANALYSE ET DE RAISONNEMENT AUTOMATISÉS
PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Eva DARULOVÁ

acceptée sur proposition du jury:

Prof. M. Odersky, président du jury
Prof. V. Kuncak, directeur de thèse
Prof. R. Bodik, rapporteur
Prof. C. Koch, rapporteur
Dr R. Majumdar, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2014

Dědečkovi

Acknowledgements

I would like to thank Viktor Kuncak for his guidance during my five years at EPFL. His wealth of knowledge, enthusiasm and perseverance are amazing and and have been very motivating. I have much appreciated the independence he gave me in pursuing my own research topic, while relentlessly pushing me in areas that do not come to me quite as easily. His attention to detail, while making paper writing a roller-coaster experience, was highly instructive and made the most of our work. I would further like to thank my thesis committee members Rupak Majumdar, Rastislav Bodik, Christoph Koch and Martin Odersky for their time and many valuable comments.

There are many people without whom the last five years would have been significantly less fun. I thank all past and present LARA lab member who contributed in one way or another to making my stay at EPFL a pleasant experience: Ruzica, Philippe, Hossein, Giuliano, Tihomir, Etienne, Swen, Andrej, Filip, Pierre-Emmanuel, Ivan, Regis, Mikael, Ravi and Manos. Philippe and Etienne in particular deserve a medal for helping me deal with the intricacies of version control and misbehaving installs in general. Fabien Salvi showed equally great patience when I managed to break my computer(s) in yet another creative way. Danielle Chamberlain and Yvette Gallay were very helpful when it came to paper work and obscure things like sending a fax. The LAMP group and especially Manohar, Nada, Vojin, Alex, Hubert and Eugene deserve a big thank you for being great discussion and coffee drinking buddies as well as world-class 'tech-support' when it came to all things Scala. Furthermore, Petr and Cristina made sure I had great company during lunch times and beyond, Djano provided for many entertaining bilingual coffee breaks and Stefan was a great climbing, running and moving partner.

I have also had great support from afar. I thank Patty for taking the time and being such an amazing mentor. Jenny and Felix were with me from the start and were always my link to, at times much needed, non-academic reality, even though I moved from one far-away place to another. Barry kept up the spirits with endless questions, and helped me remember the Irish way of saying 'Ah sure, it'll be grand!'. Sure enough, in the end it always was.

It is safe to say that without my parents' support this thesis would not have been possible. They showed me that working hard pays off, how to enjoy the small as well as the big things in life and let me try anything - whether it meant burnt pots or moving to a foreign country. And Jana made sure I was not feeling alone in Switzerland and was always there when I needed ice cream. Finally, I thank Sebastian for his unconditional support.

Lausanne, 12 Mars 2011 E. D.

Preface

Can computers compute? We all learn the notion of arithmetic computation in school and soon start to associate it with *real numbers* and operations on them. Unfortunately, our platforms including hardware, languages and environments, have no reliable support for computing with arbitrary real numbers. Numerous software packages perform impressive counts of approximate arithmetic operations per second, allowing us to understand complex phenomena in science and engineering. Moreover, embedded controllers are in charge of our cars, trains, and airplanes. Yet this essential infrastructure offers little guarantees that the computations follow real-numbered mathematical models by which they were inspired.

The floating-point arithmetic standard is helpful in ensuring that individual operations are performed according to precision requirements. These are, however, guarantees at the *machine instruction level.* The main challenge remains to estimate how roundoff errors of individual instructions compose into the error on the overall algorithm. Programming languages (since FORTRAN) have made great progress in abstracting *exact* machine instructions into larger computation chunks that we can reason about at the source code level. Unfortunately, *approximate* computations do not abstract and compose as easily. It is tempting to combine errors of individual operations independently, as in interval arithmetic, but this proves to be too pessimistic. Our ability to perform so many individual operations quickly bites us back, resulting in correct, but useless, estimates. Affine arithmetic computation turns out to be an improvement, and this thesis shows its first use that is compatible with floating points. Unfortunately, it is not only that the errors accumulate, but also that they are highly dependent on the ranges of values. Any precise analysis of error propagation needs to include the analysis of ranges of values.

This thesis presents analysis techniques that estimate the errors and sensitivity to errors in computations. The techniques involve much deeper reasoning about real number computations than what compilers perform today. Moreover, it proposes an ideal programming model that is much more friendly to users: programs can be written in terms of real numbers along with specifications of the desired error tolerance. The question of error bounds becomes the question of correct compilation, instead of a manual verification issue for users. Such a model also opens up optimization opportunities that rely on reasoning with real numbers. This includes fundamental properties such as associativity of addition, which does not hold for floating points, yet is needed for, e.g., parallelization of summations. The thesis shows that it is possible to automatically find equivalent algebraic expressions that improve the precision compared to the originally given expression.

Whether the implementation is derived automatically or manually, the fundamental problem of computing the worst-case error bound on the entire program remains. When we perform forward computation from initial values, we need to maintain the information about our result as precisely as possible, taking into account known algebraic properties of operations and any invariants that govern the computation process. To compute bounds automatically, it is natural to examine decision procedures for non-linear arithmetic, whose implementations within solvers such as Z3 have become increasingly usable. It is tempting to consider the entire problem of approximate computation as asking a theorem prover a series of questions. However, the automated provers use algorithms with super-exponential running times, so we must delimit the size of queries we ask, and must be prepared for a failure as an answer. In other words, these provers are only a subroutine in techniques tailored towards worst-case error estimation.

For self-stabilizing computations that find, e.g., zeros of a function, we can avoid tracking dependencies across loop iterations. In such situations, we can estimate how good our solution is if we know the bounds on the derivatives of functions we are computing with. The notion of derivatives of functions turns out to be as fundamental in automatically estimating errors as it is in real analysis. Together with bounding the ranges of values and performing non-linear constraint solving, derivatives become one of the main tools used for automated error computation presented in this thesis. Derivatives are crucial, for example, to compute propagation of errors in a modular fashion, or to find closed forms of errors after any number of loop iterations.

Programs handled by the techniques in this thesis need not denote continuous functions. Indeed, the widespread use of conditionals makes it very easy for software to contain discontinuities. Techniques introduced in the thesis can show the desired error bounds even for such programs with conditionals, where a real valued computation follows on one branch, whereas the approximate computation follows a different branch.

Real numbers, as their set-theoretic definitions suggest, are inherently more involved than integers. Approximation of real number operations seems necessary for computability. With emerging hardware models, approximation also presents an opportunity for more energy-efficient solutions. To reason and compile approximate computations we need new analysis techniques of the sort presented in this thesis: techniques that estimate variable ranges by solving possibly non-linear constraints and accounting for variable dependencies, that leverage the self-stabilizing nature of algorithms, and that use derivatives of algebraic expressions while soundly supporting conditionals. This thesis presents a comprehensive set of techniques that address these problems, from the point of view of both verification and synthesis of code. The resulting implementations are all publicly available in source code.

Thanks to these techniques, thinking in terms of real numbers is becoming closer to reality, even for software developers.

Lausanne, 15 October 2014 Viktor Kunčak

Abstract

Numerical software, common in scientific computing or embedded systems, inevitably uses an approximation of the real arithmetic in which most algorithms are designed. In many domains, roundoff errors are not the only source of inaccuracy and measurement as well as truncation errors further increase the uncertainty of the computed results. Adequate tools are needed to help users select suitable approximations (data types and algorithms) which satisfy their accuracy requirements, especially for safety- critical applications.

Determining that a computation produces accurate results is challenging. Roundoff errors and error propagation depend on the ranges of variables in complex and non-obvious ways; even determining these ranges accurately for nonlinear programs poses a significant challenge. In numerical loops, roundoff errors grow, in general, unboundedly. Finally, due to numerical errors, the control flow in the finite-precision implementation may diverge from the ideal real-valued one by taking a different branch and produce a result that is far-off of the expected one.

In this thesis, we present techniques and tools for *automated* and *sound* analysis, verification and synthesis of numerical programs. We focus on numerical errors due to roundoff from floating-point and fixed-point arithmetic, external input uncertainties or truncation errors. Our work uses interval or affine arithmetic together with Satisfiability Modulo Theories (SMT) technology as well as analytical properties of the underlying mathematical problems. This combination of techniques enables us to compute sound and yet accurate error bounds for nonlinear computations, determine closed-form symbolic invariants for unbounded loops and quantify the effects of discontinuities on numerical errors. We can furthermore certify the results of self-correcting iterative algorithms.

Accuracy usually comes at the expense of resource efficiency: more precise data types need more time, space and energy. We propose a programming model where the scientist writes his or her numerical program in a real-valued specification language with explicit error annotations. It is then the task of our verifying compiler to select a suitable floating-point or fixed-point data type which guarantees the needed accuracy. Sometimes accuracy can be gained by simply re-arranging the non-associative finite-precision computation. We present a scalable technique that searches for a more optimal evaluation order and show that the gains can be substantial.

We have implemented all our techniques and evaluated them on a number of benchmarks from scientific computing and embedded systems, with promising results.

Key words: floating-point arithmetic, fixed-point arithmetic, roundoff errors, numerical accuracy, static analysis, runtime verification, software synthesis

Zusammenfassung

Numerische Software, wie sie im wissenschaftlichen Rechnen oder eingebetteten Systemen verbreitet ist, nutzt zwangsläufig eine Approximation der reelen Zahlen in denen die meisten Algorithmen entwickelt werden. In vielen Bereichen sind Rundungsfehler nicht die einzige Quelle von Ungenauigkeiten und Mess- und Abschneidefehler erhöhen die Unsicherheit der berechneten Ergebnisse zusätzlich. Es braucht angemessene Tools, um Benutzern die Auswahl von geeigneten Approximationen (Datentypen und Algorithmen) zu erleichtern, die ihren Präzisionsanforderungen gerecht werden. Dies ist vor allem bei sicherheitskritische Anwendungen wichtig.

Ob die Ergebnisse einer Rechnung akkurat sind, ist allerdings schwierig festzustellen. Rundungsfehler und Fehlerfortpflanzung hängen auf komplexe Art und Weise von den Intervallen der Variablen ab, und schon die genaue Bestimmung dieser Intervalle stellt bei nichtlinearen Programmen eine erhebliche Herausforderung dar. In numerischen Schleifen wachsen Rundungsfehler im allgemeinen unbegrenzt. Hinzu kommt, dass durch numerische Fehler die Implementation in endlicher Arithmetik von der idealen reelen abweichen kann, indem sie einen anderen Weg durch die Kontrollstrukturen nimmt, und somit ein Ergebnis produziert, das weit vom Erwarteten ist.

In dieser Arbeit präsentieren wir Methoden und Tools für die *automatisierte* und *korrekte* Analyse, Verifikation und Synthese von numerischen Programmen. Unser Schwerpunkt liegt dabei auf numerischen Fehlern durch Rundungen von Gleitkomma- und Festkommaarithmetik, Unsicherheiten an externen Inputs oder Abschneidefehler. Unsere Arbeit verwendet Intervall- oder affine Arithmetik zusammen mit SMT Technologien (Satisfiability Modulo Theories) sowie die analytischen Eigenschaften der zugrunde liegenden mathematischen Probleme. Diese Kombination ermöglicht uns korrekte und dennoch präzise Fehlerschranken für nichtlineare Rechnungen und geschlossene symbolische Invarianten für unbegrenzte Schleifen zu finden und die Auswirkungen von Kontrollstrukturen auf numerische Fehler quantitativ zu bestimmen. Wir können darüber hinaus die Ergebnisse von selbst-korrigierenden iterativen Algorithmen zertifizieren.

Genauigkeit ist in der Regel nur auf Kosten der Ressourceneffizienz möglich: genauere Datentypen brauchen mehr Zeit, Platz und Energie. Wir stellen ein Programmiermodell vor, in dem der oder die Wissenschaftler/in numerische Programme in einer reellwertigen Spezifikationssprache mit expliziten Fehleranforderungen schreibt. Es ist dann die Aufgabe eines Verifizierungs-Compilers einen geeigneten Gleitkomma- oder Festkommadatentyp zu wählen, der die erforderliche Genauigkeit gewährleistet. Manchmal kann Präzision einfach durch eine

andere Rechenreihenfolge gewonnen werden, da Computerarithmetik nicht assoziativ ist. Wir stellen eine skalierbare Methode vor, die nach einer optimalen Reihenfolge sucht und zeigen, dass die Präzisionsgewinne erheblich sein können.
Wir haben alle unsere Methoden implementiert und sie an einer Reihe von Beispielen aus den Bereichen des wissenschaftlichen Rechnens und eingebetteter Systeme mit vielversprechenden Ergebnissen evaluiert.

Stichwörter: Gleitkommaarithmetik, Festkommaarithmetik, Rundungsfehler, numerische Präzision, Statische Analyze, Laufzeitverifikation, Softwaresynthese

Résumé

Des logiciels numériques, communs dans le calcul scientifique ou dans les systèmes embarqués, utilisent inévitablement une approximation de l'arithmétique réelle dans laquelle la plupart des algorithmes ont été conçus. Dans de nombreux domaines, les erreurs d'arrondi ne sont pas la seule source d'imprécision et les erreurs de mesure et les erreurs de troncature augmentent encore l'incertitude des résultats calculés. Des outils adéquats sont nécessaires pour aider les utilisateurs à sélectionner des approximations convenables (types de données et algorithmes) qui répondent à leurs exigences en matière de précision, en particulier pour les systèmes critiques.

Déterminer si le résultat d'un calcul est précis est difficile. Les erreurs d'arrondi et la propagation d'erreurs dépendent des intervalles de variables de façon complexe et non triviale ; rien que la détermination précise de ces intervalles pour les programmes non linéaires représente un défi considérable. Dans les boucles numériques les erreurs d'arrondi se développent, en général, sans limite. Enfin, à cause des erreurs numériques, le flux de contrôle d'une implémentation en précision finie peut s'écarter de l'implémentation réelle idéale en prenant une branche différente et le résultat obtenu peut être très différent de celui attendu.

Dans cette thèse, nous présentons les techniques et les outils pour l'analyse, la vérification et la synthèse de programmes numériques qui est sûre et automatisée. Nous mettons l'accent sur les erreurs numériques dues aux arrondis de l'arithmétique en virgule flottante et de l'arithmétique en virgule fixe, aux incertitudes externes d'entrée ou aux erreurs de troncature. Notre travail utilise l'arithmétique d'intervalles ou affine avec la technologie SMT (Satisfiability Modulo Theories) ainsi que des propriétés analytiques des problèmes mathématiques sous-jacents. Cette combinaison de techniques nous permet de calculer des limites d'erreur sûres et néanmoins précises pour les calculs non-linéaires. Cela nous permet aussi de déterminer des invariantes symboliques et fermées pour les boucles infinies et de quantifier les effets de discontinuités sur les erreurs numériques. Au-delà, nous pouvons certifier les résultats des algorithmes itératifs autocorrectif.

La précision est généralement au détriment de l'efficacité des ressources : de plus précis types de données ont besoin de plus de temps, d'espace et d'énergie. Nous proposons un modèle de programmation, où la/le scientifique écrit le programme numérique dans un langage de spécification en valeur réelle avec des annotations d'erreur explicites. C'est la tâche d'un compilateur vérificateur de sélectionner le type de données approprié, soit en arithmétique flottant ou en virgule fixe, qui garantie la précision nécessaire. De la précision peut, parfois, être gagnée simplement en réorganisant l'arithmétique non-associative en précision finie.

Nous présentons une technique évolutive qui cherche un ordre d'évaluation optimal et nous montrons que les gains peuvent être substantiels.
Nous avons implémenté nos techniques et nous les avons évaluées avec un certain nombre d'exemples de calculs scientifique ainsi que de systèmes embarqués, avec des résultats prometteurs.

Mots clefs : arithmétique flottante, arithmétique en virgule fixe, erreurs d'arrondi, précision numérique, analyse statique, vérification durant exécution, synthèse de logiciels

Contents

List of Figures

List of Tables

1 Introduction

Numerical errors are rare, rare enough
not to care about them all the time,
but yet not rare enough to ignore them.
— William M. Kahan

Numerical programs are widely used in mathematics, science and engineering. For example, scientific computing works with mathematical models which aim to explain the phenomena of the physical world. Since they usually cannot be solved analytically, numerical algorithms are needed to find approximate solutions. In computer science, many areas such as image and signal processing, graphics, vision and machine learning also rely on numerical computations. Finally, many of the devices we use on an every day basis are cyber-physical or embedded systems: they have a numerical control system that interacts with the physical world. Many of these applications are safety-critical, so it is very important to verify that they perform their function correctly.

One of the many aspects that makes their verification difficult is the inherent gap between the continuous nature of the mathematics and the physical processes and the discrete implementation on today's digital computers. Many numerical algorithms are naturally expressed in real arithmetic and often use algorithms which compute the exact answer only in the limit, for instance with infinitely many components of a sequence or with an infinitely small step size. Unfortunately, on a computer we only have finite resources available so that we need to discretize and approximate both the arithmetic and the infinite iterations. Finite-precision arithmetic can approximate real arithmetic, but needs to round the result of every computation step, committing a *round-off error*. When we stop an iteration after a finite number of steps, we introduce another difference to the ideal computation, which we call *truncation error*. Sensors or experimental equipment is not perfectly accurate either and contributes a *measurement error* on the inputs which then ultimately also affects the overall uncertainty on the result.

Altogether we need to consider a variety of errors when assessing the correctness of numerical results: measurement, roundoff and truncation errors. While the individual errors are usually small, they accumulate and can render the final result inaccurate or entirely meaningless. An N-version experiment [81] has shown that different implementations of an identical seismic data processing algorithm can produce vastly different results. Much of the discrepancy has been traced back to the numerical portion of the code. One important task of numerical program verification is thus understanding how the individual errors combine and, ideally, show that the overall error remains small enough.

Finite-Precision Roundoff Errors Because of the absence of real arithmetic, todays numerical algorithms usually use finite-precision arithmetic, such as floating-point or fixed-point arithmetic, for their computations. Base-two floating-point arithmetic has become popular also because it is nowadays implemented on dedicated hardware and the computations are thus very fast. Fixed-point arithmetic can be implemented entirely with integers and is mostly used when a dedicated floating-point unit is not available because of cost or energy constraints, as is the case for many embedded systems. Both arithmetics work on a rational subset of the reals with a limited number of digits for precision and with a limited range. For example, $1/10$ cannot be represented exactly in standard floating-point arithmetic because it is not a power of two, so if we then compute $0.1 * 9$ in single (32 bit) precision we get 0.90000004 instead of 0.9. Furthermore, finite-precision arithmetic is not associative and multiplication does not distribute so the usual mathematical equivalences do not hold any more. This implies that the order of computation matters and, for example, summing up a list of numbers in two different orders can produce two vastly different results if the results do not have the same orders of magnitude.

Unfortunately, round-off (and other) errors are propagated through a computation in complex and hard to predict ways. Even seemingly simple computations like calculating the area of a triangle can have surprising results. Recall the standard textbook formula for a triangle with sides a, b and c:

$$A = \sqrt{s(s-a)(s-b)(s-c)} \qquad \text{where } s = \frac{a+b+c}{2}$$

We have implemented this computation with double precision and with quadruple double floating-point arithmetic [16], and then randomly sampled inputs for a, b and c. If we compare the results, we obtain an estimate on the absolute round-off error of the double precision computation, which we plot in Figure 1.1. We observe that for some triangles whose area is small the errors grow. A careful examination of the computation reveals that the textbook formula is numerically unstable for flat triangles [93]. That is, for triangles where the sum of two sides is close to the length of the third (e.g. $a + b \approx c$), the subsequent difference $s - c$ becomes very small and magnifies the existing round-off errors. This effect is also called cancellation as many correct significant digits cancel out in the subtraction and leave mostly digits affected by errors behind. Spotting cancellation can be very hard as it requires ranges of all variables to be known.

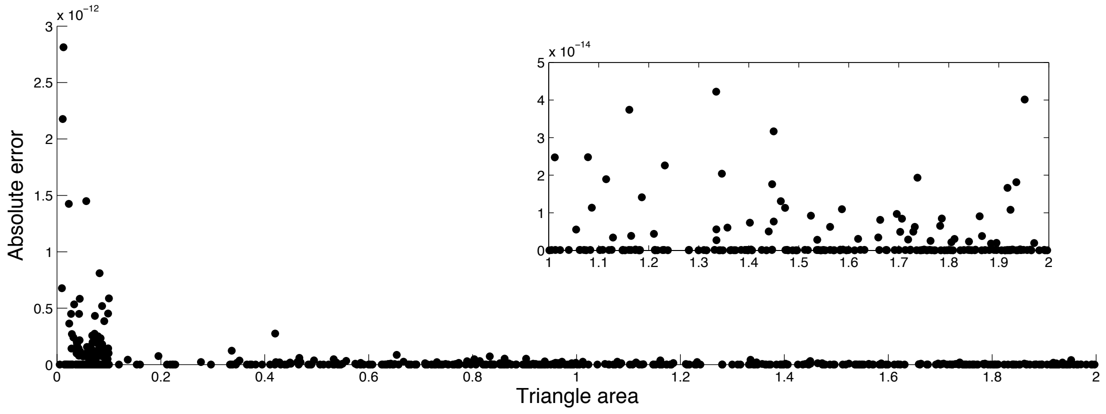


Figure 1.1 – Absolute errors committed for different triangle areas, determined by random sampling and comparing results obtained with double precision against those obtained with quadruple double precision. The inset shows a magnification of the triangle area range [1, 2].

Furthermore, the errors in Figure 1.1 are clearly not evenly spread. As the inset shows for ranges of triangle areas between 1 and 2, round-off errors appear to be rather randomly distributed. It is clear then that we cannot in general verify the result of a numerical computation by a simple manual inspection of either the source code or the result. Furthermore, approaches based on sampling as are used in testing [18, 39, 37] require a large number of samples to provide reasonable error estimates, but still cannot provide sound guarantees.

Accuracy vs Efficiency There is an inherent trade-off between accuracy and efficiency: the more precise data type we choose, the longer the computation is going to run and the more energy it is going to consume in general. One size does not fit all, however, and where a program is placed on this spectrum highly depends on the particular requirements and characteristics of the application. If the measurement noise is large and obscures any round-off errors, or the application can tolerate a certain error (e.g. a human observer), then we may not need the full 64 bit floating-point precision that is often the default choice [143]. In other cases, an (embedded) device may not even have a floating-point unit and the computation has to be implemented in fixed-point arithmetic, while being accurate enough to ensure stability of your controller [108]. Other computations, however, may have very high precision requirements, because of numerical instabilities or because a small difference can have a huge effect on decision making [33]. Similar considerations also apply to the choice of algorithms as well, where they influence the magnitudes of truncation errors for instance. Energy and performance concerns have recently also sparked interest in approximate hardware, which performs for example arithmetic operations less precisely [20]. However, in order to use this trade-off without sacrificing correct behavior of the application we need to have some confidence that our choice of data type (and algorithm) is appropriate. For this we have to be able to quantify the overall errors and convince ourselves that they are sufficiently small.

State of the Art Numerical error estimation is an integral part of numerical program verification and has always been a concern in traditional numerical analysis [95]. The programmer is being cautioned that round-off errors can corrupt the results, but very often it is hard to tell when this is actually the case and no general guidelines exist. Verification is also an important part of scientific computing, although it has a somewhat different meaning than in software verification. Verification and validation (V&V) is mostly concerned with ensuring that numerical computations conform to the physical reality or agree with analytical solutions sufficiently, if such exist [124]. Validated numerics are another example of verified computation, where interval methods are used to compute guaranteed enclosures of the ideal real-valued result [137]. While all these approaches are being successfully used, they are also very specific and need to be developed for each particular problem individually. In addition, interval arithmetic cannot be used blindly as it looses correlation information and for useful results the computation may need to be completely reformulated in a non-obvious way.

In software verification, tools that perform a sound analysis exist as well. For example, interactive theorem provers can prove very detailed and precise properties about floating-point programs [13, 25, 103], but require an expert user. Hardware verification on the other hand, due to its finite-state nature, has been very successful [80] and is now routinely used, but software verification tools still have a long way to go before they can be called mature or even merely practical for use by the average programmer [102]. Only Fluctuat [72] which is based on abstract interpretation [42] is an automated tool for soundly estimating round-off errors.

The goal of this dissertation is to advance the state of the art of numerical error analysis, verification and synthesis, by developing *general*, *automated* and *accurate* techniques and tools which are applicable to a wide range of applications and can also be used by non-experts in numerical analysis.

1.1 Challenges in Analysis and Verification

A central concern in numerical program verification is the determination of the **ranges** (or intervals) individual variables can take. This is important, for example, if an algorithm is only defined or valid on certain domains and we need to verify that all inputs, which may also be results of previous computations, respect this domain. Range computation is however also important for error estimation because the magnitude of round-off errors directly depends on the magnitude of the results as the number of maximum digits is fixed. Many numerical applications are nonlinear and feature correlations between its variables, characteristics that make range computation a challenge. The standard method to estimate the range of an arithmetic expression $f(x)$ given the range for the input(s) x has been interval arithmetic [120]. Unfortunately it does not keep track of any correlations and thus can produce greatly inaccurate results. Unlike in many domains, where linearization is a widely used and a suitable technique, naively applied in the context of numerical estimation, it produces results that are too approximate to be very useful.

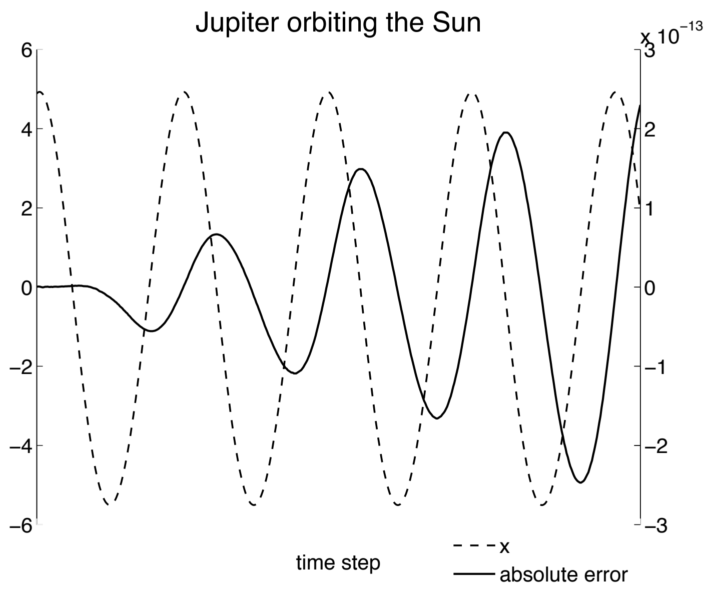


Figure 1.2 – Absolute actual round-off errors computed by comparison against quad double precision arithmetic in 300 iterations of a simple gravity simulation. Note that the actual error grows with the iteration number.

Conditional branches introduce discontinuities which can, in the presence of errors, result in diverging behavior between the ideal real-valued computation and its finite-precision implementation. We call the resulting difference *discontinuity error*. For example, suppose that we want to approximate $\sqrt{(1+x)}$ for small x:

```
if (x < 1e-4) 1 + 0.5 * x
else sqrt(1 + x)
```

If our application can tolerate a less accurate square root computation, the approximation is preferable as it is significantly faster. If in the preceding computation x has accumulated an error, and the finite-precision value is for example 0.00099 instead of 0.00012, the finite-precision computation will take the less accurate if branch, even though the ideal real computation would have taken the else-branch. Since the computation of this conditional is not continuous, the difference between the real and the finite-precision result due to round-offs is further exacerbated by the conditional. Quantifying discontinuity errors is hard due to nonlinearity and because the two branches of the conditional usually share variables and are thus tightly correlated.

Loops are also a common feature in numerical programs. In forward computations such as numerical integration of differential equations round-off errors grow in general unboundedly. As an illustration, consider a simulation of the planet Jupiter orbiting the Sun, for which we plot the absolute errors of one of the coordinates, x, in Figure 1.2. While the values of x stay bounded, the errors grow, making it impossible to find a constant absolute error bound. For this reason current error computation techniques do not work well and often return an error bound of $[-\infty, \infty]$.

Numerical errors in loops in many iterative algorithms behave differently however. These algorithms are often self-correcting in that the errors from one loop iteration are corrected in later ones. This means then that tracking round-off errors across iterations, as current approaches would do, is not very sensible. With each iteration, these algorithms gradually converge to the exact result, but truly reach it only in the limit. As we need to stop the algorithm after a finite number of iterations, the computed result is only approximate. We want to quantify the truncation error that has been committed, but do so in a way that is generally applicable to a large class of applications.

1.2 Synthesis

Too often, accuracy is an afterthought. For most applications today, the programmer picks a numerical data type for his or her program first, implements the algorithm, as if it was in reals and then (maybe) remembers to check that the results produced are close enough to what is expected. Such an approach limits the range of possible implementations by an up front choice of data type, but also limits applicable optimizations since finite-precision arithmetic does not obey many mathematical rules. Furthermore, this approach creates a gap between the ideal real-valued algorithm (on paper, in math) and the actual finite-precision implementation. The programmer has to deal with implicit low-level details of the finite-precision arithmetic. We argue that accuracy should ideally be made *explicit* and always be part of the program itself. One easy and light-weight way, which we also use in this thesis, is to include assertions which also check for numerical errors. Going further, we also propose a specification language which allows to write programs over a `Real` data type with explicit accuracy requirements. It is then up to a “verifying compiler” to select a suitable data type that satisfies the error specification. Thus, the compiler synthesizes a finite-precision implementation from its real-valued specification fully automatically.

Finite-precision arithmetic does not obey the usual arithmetic laws that real arithmetic does. In particular it is not associative, so different computation orders can produce different results. This is especially surprising if such a re-ordering is performed by a (traditional) compiler silently behind the scenes, even though in reals such a transformation is mathematically valid. We can however, exploit this ‘feature’ of finite- precision arithmetic to reduce round-off errors “for free” by searching among the possible mathematically equivalent formulations of our computation and generate the one which minimizes the overall round-off error. Such a rewriting is semantically permissible when the input language is real-valued, for example like our proposed specification language, and thus admits such transformations. Since we can do this optimization over an entire range of inputs at compile time, we can improve the accuracy of the computation without changing the data type.

1.3 Outline and Contributions

This thesis presents techniques for sound numerical error analysis, verification and synthesis. In particular, we have developed methods for accurately estimating errors in different kinds of numerical programs. Furthermore, we explore both static and runtime approaches and present different ways of integrating numerical verification into a programming language with data type libraries, macros and compiler plugins. We have implemented and experimentally evaluated all our techniques for their usability and effectiveness. This thesis is organized as follows:

Chapter 2 provides necessary background on finite-precision and range arithmetic and then describes our technique to compute and track round-off errors of floating-point and fixed-point arithmetic. Denoting by f the ideal real- valued function and by $\tilde{f}$ the corresponding finite-precision one, we develop a method to soundly bound the roundoff error $|f - \tilde{f}| \leq \rho$. Our method is based on affine arithmetic [51] which we adapt for tracking the errors of a single computation $\left(|f(a) - \tilde{f}(a)| \leq \rho\right)$ as well as for a range $[a, b]$ of user-defined inputs $\left(\forall x \in [a, b] \,.\, |f(x) - \tilde{f}(x)| \leq \rho\right)$. We can track round-off errors for all arithmetic operations, as well as a number of commonly used transcendental functions. The approach for tracking roundoff errors presented in this chapter forms the basis of all our techniques and is used throughout this thesis.

Chapter 3 presents the `AffineFloat` and `SmartFloat` data types which by wrapping our round-off error analysis replace the regular `Double` floating-point type and provide in addition to the usual result also a sound upper bound on the error. In this dynamic approach, we integrated our runtime library seamlessly into Scala, making it easily usable on existing code. Our experiments show that the affine arithmetic based error computation in general outperforms the traditional interval arithmetic based approach precision-wise, often by orders of magnitude. We further demonstrate the ability of our data types to identify problematic code such as the triangle area computation. Chapters 2 and 3 are based on the paper [44].

Chapter 4 uses our round-off error computation together with a genetic algorithm to search for fixed-point arithmetic implementations that minimize round-off errors. That is, given a real-valued expression t, our tool automatically searches for t' which is mathematically equivalent and satisfies: $\min_{\text{equiv. } t'} \quad \max_{x \in [a,b]} \left| t(x) - \tilde{t}'(x) \right|$. By exploiting the non-associativity of the arithmetic, our tool Xfp can find mathematically equivalent expressions which are often up to 50% more precise than the baseline formulation, as it may be generated by a tool like MATLAB [4]. This work is based on the paper [48].

Chapter 5 introduces our real-valued specification language for writing numerical programs, which explicitly includes errors. We present a compilation algorithm that takes this language over the **`Real`** data type and based on the given error requirements determines a suitable floating-point or fixed-point data type for the concrete implementation.

In order to handle interesting code including method calls, we introduce a range of approximations, which we have implemented in our “verifying compiler” tool called Rosa. This work has been presented in [47].

Chapter 6 presents our techniques for statically computing accurate range and error bounds inside Rosa on straight-line arithmetic expressions. Combining interval and affine arithmetic with nonlinear SMT solving allows us to use correlations between variables as well as additional constraints to determine range bounds accurately: ($[a, b] = \text{getRange}(P, \text{expr}) \Rightarrow \forall x, \text{res}. P(x) \wedge \text{res} = f(x) \rightarrow (a \leq \text{res} \wedge \text{res} \leq b)$, where P captures initial ranges and additional constraints). Then we separate numerical errors into propagated initial errors and newly committed round-off errors and use this separation to derive a new method for error propagation ($\forall x, y \in [a, b] \,.\, |x - y| \leq \text{init.error} \rightarrow |f(x) - f(y)| \leq \text{prop.error}$), which improves error bounds significantly over plain affine arithmetic.

Chapter 7 presents two techniques for statically computing discontinuity errors due to conditional branches which trade off accuracy for scalability in different ways ($\forall x, y \in [a, b] \,.\, |x - y| \leq \text{init.error} \rightarrow |f_1(x) - \tilde{f}_2(y)| \leq \text{disc.error}$, where f_1 and f_2 represent the two branches). Then, we apply our separation of errors idea and error propagation technique to (unbounded) loops and derive an error specification which is a function of the number of iterations. Beyond this inductive specification, we also show that this approach allows us to compute error bounds for application which were out of reach for state-of- the-art tools. That is, we compute a bound on $|f^m(x) - \tilde{f}^m(y)|$ valid for all $x, y \in [a, b]$ with $|x - y| \leq \text{init.error}$. Chapter 6 and 7 are based on [47] as well as recent work under submission [46].

Chapter 8 presents a certification procedure, implemented in the library called Cassia, for dynamically verifying the accuracy of solutions of systems of nonlinear equations. These applications are usually solved with self-correcting biterative methods which are highly optimized. By integrating theorems from validated numerics with an assertion-based approach we can leverage the efficiency of these methods, yet provide guarantees on the results. We have presented this work in [45].

1.4 Usage Scenarios

This thesis presents four open-source implementations of our techniques. While they are, for now, separate, we view them as building blocks which can be combined into one overall system, handling different types of programs and applications.

For instance, our static analyzer *Rosa* can be used to soundly verify that numerical errors in (smaller) computation kernels stay below a required bound. The verification effort may fail, due to the limited scalability of the static approach, or because the asserted property is simply not correct. In this case, the programmer can identify a trace of interest and debug or verify it with our `SmartFloat` runtime library. If the code is using an external library for solving systems of equations, its untrusted result can be certified to be close enough to the true roots by our *Cassia* runtime library. Finally, the programmer can use our tool *Xfp* to not only verify, but also to improve a fixed-point implementation, by automatically rewriting the arithmetic expressions such that roundoff errors are reduced.

Our tools are “push-button” in the sense that they only require the user to provide bounds on the input ranges and, where applicable, measurement errors on inputs in addition to the program itself. We believe that it is reasonable to expect that a domain expert can provide such a specification. Each tool then performs the error estimation or search fully automatically.

We imagine our tools to be used, for example, by embedded systems engineers to soundly verify that roundoff errors do not invalidate stability properties established for a real-valued world. For the programmer of a scientific computing application, our tools can help him or her to find numerical issues in the computation kernel, or determine which of several possible alternative implementations is preferable from the perspective of numerical stability.

In this spirit, we have tested and evaluated all our tools on real-world benchmarks such as embedded controllers, physical simulations and problems from biology as well as chemistry. In addition, we have applied our tools in two case studies, where we helped astronomers and researchers at EPFL gain confidence in their implementations. In subsection 3.2.6, we apply our `SmartFloat` library to an computationally involved astronomical program and verify that the result is accurate enough for the needed purpose. In subsection 8.5.2, we are able to certify that solutions of an optimization problem from the energy domain are indeed correct up to the required error threshold.

2 Bounding Finite-Precision Roundoff Errors

MATLAB's creator Dr. Cleve Moler used to advise foreign visitors
not to miss the country's two most awesome spectacles:
the Grand Canyon, and meetings of IEEE p754.
— William M. Kahan [145]

One of the main challenges in writing numerical programs is finite-precision arithmetic (floating-point or fixed-point), because it inevitably introduces round-off errors. While the round-off from one arithmetic operation may appear tiny, these errors can accumulate and even render results entirely meaningless [81]. In the area of embedded systems, it has been shown that the region of stability for embedded controllers directly depends on the round-off errors [9]. It is thus important to understand how these errors accumulate and propagate through a computation.

This chapter begins with necessary background on floating-point, fixed-point, interval and affine arithmetic (AA) [51]. We then introduce our method for computing and tracking round-off errors which forms an important building block for our techniques in the rest of this thesis.

Our method for computing and tracking round-off errors is based on affine arithmetic. Standard affine arithmetic cannot be used as-is for a sound error computation, so we introduce two adaptations: for tracking round-off errors for a single computations and for all computations whose inputs are in given ranges. In addition to arithmetic operations, we also implement AA for commonly used transcendental functions. It turns that this is non-trivial to implement because of round-off errors in the internal computation. We describe our solution which nonetheless provides sound and accurate bounds. We will use the error computation presented here as a building block for

Our error computation tracks *worst-case absolute errors* of floating-point or fixed-point arithmetic. That is, if x is the ideal real value we want to compute and $\tilde{x}$ is the actually computed number in finite arithmetic, then our goal is to determine a sound upper bound on $|x - \tilde{x}|$.

Naturally, we want this bound to be as tight as possible. In order to achieve soundness, we assume worst-case round-off errors at each operation, even though they will not be reached in practice at every computation step. Our analysis tracks absolute and not relative errors ($|(x-\tilde{x})/x|$) as this is more natural, but relative errors can be computed from the absolute error bound, if needed. When the range of x includes zero, computation of relative errors becomes problematic, however, due to division by zero. This can happen e.g. when tracking a range of positive and negative values even if the zero value is never obtained in practice. Choosing absolute errors avoids this problem, so we report these.

The material in this chapter is mostly based on [44].

2.1 Finite-Precision Arithmetic

We begin with a review of floating-point and fixed-point arithmetic, which we denote by $\mathbb{F}$.

2.1.1 IEEE Floating-Point Arithmetic

We assume throughout that floating-point arithmetic conforms to the IEEE 754 floating-point standard [84]. Recent CPUs generally follow the standard fully, and most programming languages respect the minimal set of requirements that we rely on and which we present here. Our tools and examples all run on the Java Virtual Machine (JVM), whose (partial) conformance to the IEEE 754 standard is documented in [104]. The standard defines, among others, the single and double precision floating-point data types (with 32 and 64 bits respectively), arithmetic operations on these data types and various rounding modes.

The five rounding modes available in IEEE 754 are rounding to nearest (ties to even or ties away from zero) and directed rounding (towards zero, $+\infty, -\infty$). We will assume that the code we are interested in uses rounding to nearest, ties to even. This is the default rounding mode in most programming languages, and also the only one available on the JVM. Our tools also use the directed rounding modes towards $+\infty, -\infty$ internally, namely for interval arithmetic, which we implement through a native library. In those cases, we will denote by $x\downarrow$ and $x\uparrow$ the value of x when rounded towards $-\infty$ and ∞ respectively.

The standard specifies that the basic arithmetic operations $\{+,-,*,/,\sqrt{}\}$ be computed as if first an exact infinite-precision result was computed and then rounded to the specified precision. For the rounding-to-nearest rounding mode this means that results are rounded correctly, that is, the result from any such operation must be the closest representable floating-point number. Provided there is no overflow and underflow, the result of an arithmetic operation in floating-point arithmetic $\{\oplus,\ominus,\otimes,\oslash,\not\sqrt{}\}$ then satisfies

$$\begin{aligned} x \odot y &= (x \circ y)(1+\delta), \qquad \circ \in \{+,-,*,/\},\ \ \odot \in \{\oplus,\ominus,\otimes,\oslash\} \\ \not\sqrt{x} &= (\sqrt{x})(1+\delta), \qquad |\delta| \leq \epsilon_M \end{aligned} \tag{2.1}$$

where ϵ_M is the so-called *machine epsilon* that determines the upper bound on the relative error. For instance, for single precision $\epsilon_M = 2^{-24}$ and for double precision $\epsilon_M = 2^{-53}$. We will use this abstraction throughout to compute round-off errors. These can also be measured in terms of the *unit in the last place* (ulp), which is the value of the least significant bit of a floating-point number. Arithmetic operations are then rounded to within a $\frac{1}{2}$ ulp.

Underflow occurs when a computed value is too small to be represented by a regular floating-point number (called normalized) and is handled by one of two cases. The first option is flushing, i.e. by discarding all significant digits. The roundoff error is then bounded by the smallest positive normalized number. The second, more standard, option is gradual underflow, where denormalized numbers are used to fill the gap between the smallest normalized number and zero. The roundoff error is bounded by half the ulp of the smallest positive normalized number in this case. In this thesis, we consider a range containing only denormals to be an error and will thus be working only with Equation 2.1. Note that for systems with gradual overflow this bound on the round-off error soundly over-estimates also the round-off for any denormal results contained in a larger range. For more (detailed) information on floating-points see [71].

The techniques described in this thesis can be ported to other languages, such as C/C++, or to languages specifically targeted for instance for GPU's or parallel architectures as long as the computation order is not changed by the compiler and provided the implementation of floating-point arithmetic adheres to the IEEE 754 rounding-to-nearest semantics (Equation 2.1), and uses denormal floating-point numbers for underflow.

While our examples use single- and double-precision floating-point numbers, our techniques are parametric in the machine epsilon. This implies that we can analyze code in any floating-point implementation, as long as it can be abstracted by Equation 2.1. Changing the number of exponent and mantissa bits may be useful for specific applications with programmable hardware platforms or with dynamic (software) allocation of exponent and mantissa bits.

2.1.2 Higher-Precision Floating-Point Arithmetic

One possible way to avoid or reduce round-off errors is to use higher-precision or arbitrary precision arithmetic. These are usually provided through software libraries, for instance, the QuadDouble library uses two or four double floating-point numbers to obtain 32 and 64 decimal digit precisions [82]. The GNU MPFR Library [62] implements multiple-precision floating-point computations, and many other similar libraries exist. For certain applications, for instance geometric computations, such a high precision is indeed crucial. For example, the library LEDA keeps track of the computation history in order to determine geometric predicates [116]. For all other applications, however, we want to benefit from the performance the hardware support of single and double-precision IEEE floating-point provides.

[77] proposes a new data type with a variable number of mantissa and exponent bits in order to make arithmetic more memory efficient. Using this new data type, however, requires to specify up front a limit on the maximum number of these bits and it is not clear how to estimate it soundly ahead of time, so it remains to be seen how applicable it will be to real-world problems.

2.1.3 Fixed-Point Arithmetic

An alternative for representing a subset of the rational numbers on a computer is fixed-point arithmetic. It presumes integer arithmetic hardware support only and is thus a common choice on embedded devices where a floating-point unit is not a good option due to resource bounds. This naturally entails a trade-off, as we cannot rely anymore on the dynamic properties of floating-point arithmetic. Instead, as the name suggests, the exponent range is fixed for every intermediate variable (but not necessarily the same) and we have to implement shifts aligning them for arithmetic operations explicitly.

We assume a constant bit length for an entire computation, for example 16 or 32 bits. Every intermediate value in an arithmetic computation is assigned a *fixed-point format* which determines how many *integer bits* are required to cover the range of the value, and how many *fractional bits* f remain for the fractional digits. An integer value x can then be interpreted as the rational number $2^{-f}x$. We assume signed integer arithmetic and truncation semantics, which implies that the maximum absolute round-off error for a value is given by 2^{-f}.

An implementation of a real-valued expression into fixed-point arithmetic consists of first determining the ranges of all values in the program, including inputs, constants and intermediate results. This determines the minimum number of integer bits. Then, arithmetic operations are implemented with regular integer arithmetic and bit shifts to align the implicit decimal points [9].

Fixed-point formats are not unique and can result in different overall round-off errors. For a given arithmetic expression, this error is minimal when the round-off error at each operation is minimal. This is achieved when the formats are chosen such that the number of integer bits is sufficient but not larger than necessary, i.e. we maximize the number of fractional bits. Thus, to obtain an optimal fixed-point implementation, we need to evaluate the ranges of (intermediate) values as tightly as possible. Note that the round-off errors depend on the ranges of values, which is similar to the round-off error abstraction we use for floating-point arithmetic.

2.2 Interval Arithmetic as a Baseline

The standard approach to guaranteed computation is interval arithmetic (IA) [120], and we will use it as the baseline for our techniques and experiments. IA associates with each variable

x a closed interval $[a,b]$ such that $x \in [a,b] \Leftrightarrow a \leq x \wedge x \leq b$ where x, a, b can be elements of $\mathbb{R}, \mathbb{Q}, \mathbb{Z}$ or $\mathbb{F}$. When the lower and upper bounds are equal, we call the interval a *point interval*. We will also use the notation $[x]$ to denote the interval represented by the variable x.

Given a real-valued function $f(x_1, \dots, x_n)$ and ranges for its inputs $[a_i, b_i]$ such that $x_i \in [a_i, b_i]$, then interval arithmetic computes an over-approximation $[c,d]$ of the range of f over the input domain, i.e.

$$\forall x_i \,.\, x_i \in [a_i, b_i] \rightarrow f(x_1, \dots, x_n) \in [c,d]$$

If X and Y are two intervals, then binary arithmetic operations are defined (over $\mathbb{R}$) as

$$X \circ Y = \{z \mid \exists x \in X, y \in Y \,.\, z = x \circ y\}$$

and similarly for other functions such as square root. We can implement interval arithmetic straight-forwardly with rationals or integers, however when using floating-point arithmetic directed rounding is needed to obtain sound interval bounds.

Interval arithmetic can be used to obtain sound bounds on finite-precision round-off errors by interpreting the width of the interval as the error on the corresponding variable. Input intervals are point intervals, if the number can be represented exacly in finite precision, or have the lower and upper bound be the next smaller and bigger representable finite-precision number respectively. After performing the computation in standard interval arithmetic, we can read off the bound on the round-off error from the width of the result's interval.

Unfortunately intervals give too pessimistic estimates in many cases. The problem is easy to demonstrate: if X is an interval $[0,a]$ then interval arithmetic approximates the expression $X - X$ with $[-a,a]$, although it is, in fact, always equal to zero. Essentially, interval arithmetic ignores correlations between variables, i.e. it approximates $x - x$ in the same way as it would approximate $x - y$ when x and y are unrelated variables that both belong to $[0,a]$.

Furthermore, we would like to not only track a single computation (with initially point intervals), but a range of inputs, and obtain roundoff error bounds which are sound for all possible executions on those inputs. In other words, we want the intervals to track two sources of uncertainty:

- roundoff errors due to the difference between the ideal and the finite-precision value
- uncertainty on the inputs (which will in general be larger in magnitude)

Any approach that conflates these two sources of uncertainty will quickly become inaccurate in distinguishing the (smaller) roundoff errors. That said, intervals can be very useful in certain carefully chosen cases, and we also use them in later chapters.

2.3 Affine Arithmetic

Affine arithmetic (AA) addresses the difficulty of interval arithmetic in handling correlations between variables. Affine arithmetic was originally introduced in [51] and developed to compute ranges of functions over the domain of reals, with the actual calculations implemented in double floating-point precision. A possible application of affine arithmetic, as originally proposed, is finding roots of a function in a given initial interval by a branch-and-bound approach.

Affine arithmetic represents possible values of variables as affine forms

$$\hat{x} = x_\circ + \sum_{i=1}^{n} x_i \epsilon_i, \qquad \epsilon_i \in [-1,1]$$

Using the terminology from [51], $x_\circ$ denotes the *central value* (the mid point of the represented interval) and each *noise term* $x_i\epsilon_i$ represents a deviation from the central value with maximum magnitude x_i. We will call the ϵ_i's *noise symbols*. Note that the sign of x_i does not matter in isolation, it does, however, reflect the relative dependence between values. For example, take $x = x_\circ + x_1\epsilon_1$, then in real number semantics,

$$x - x = x_\circ + x_1\epsilon_1 - (x_\circ + x_1\epsilon_1) = x_\circ - x_\circ + x_1\epsilon_1 - x_1\epsilon_1 = 0$$

In contrast, if we do not have a correlation (no shared noise symbols), the resulting interval has width $2 * x_1$ and not zero: $x - x' = (x_\circ + x_1\epsilon_1) - (x_\circ + x_1\epsilon_2) = x_1\epsilon_1 - x_1\epsilon_2 \neq 0$.

The range represented by an affine form, denoted by $[\hat{x}]$, is computed as

$$[\hat{x}] = [x_\circ - \mathsf{radius}(\hat{x}),\ x_\circ + \mathsf{radius}(\hat{x})], \qquad \text{where} \quad \mathsf{radius}(\hat{x}) = \sum_{i=1}^{n} |x_i|$$

A general affine operation $\alpha\hat{x} + \beta\hat{y} + \zeta$ consists of addition, subtraction, addition of a constant (ζ) or multiplication by a constant (α, β). Expanding the affine forms $\hat{x}$ and $\hat{y}$ we get

$$\alpha\hat{x} + \beta\hat{y} + \zeta = (\alpha x_\circ + \beta y_\circ + \zeta) + \sum_{i=1}^{n} (\alpha x_i + \beta y_i)\epsilon_i \tag{2.2}$$

Implementation in Finite Precision When implementing affine forms in floating-point arithmetic, we need to take into account that some operations are not performed exactly because the central value and the coefficients need to be represented in some finite (e.g. double) precision. As suggested in [51], the round-off errors committed during the computation, here called *internal errors* ι, can be added with a new fresh noise symbol to the final affine form. Each operation carries a round-off error and all of them must be taken into account when computing a rigorous bound for ι. The challenge hereby consists of accounting for all round-off errors, but still creating a tight approximation. While for the basic arithmetic operations the round-off can be computed with Equation 2.1, there is no such simple formula for calculating the round-off for composed expressions (e.g. $\alpha * x_\circ + \zeta$). We determine the maximum round-off error of an expression $f(v_1, \ldots, v_m)$ using the following procedure [51]:

$$\begin{aligned} z = &\ f(v_1, v_2, \ldots, v_m) \\ z_{-\infty} = &\ f(v_1, v_2, \ldots, v_m)\downarrow \\ z_{+\infty} = &\ f(v_1, v_2, \ldots, v_m)\uparrow \\ \iota = &\ \max(z_{+\infty} - z, z - z_{-\infty}) \end{aligned}$$

where $\downarrow, \uparrow$ denotes rounding towards $-\infty$ and ∞ respectively. That is, the program computes three values: (1) the floating-point result z using rounding to-nearest, (2) the result $z_{-\infty}$ assuming worst-case round-off errors when rounding towards $-\infty$, and the analogous result $z_{+\infty}$ with rounding towards $+\infty$ at each step. We determine the worst-case committed round-off error ι as the maximum difference between these values.

An alternative is to use a rational data type backed by arbitrary precision integers (e.g. Java's `BigInteger`) to avoid having to deal with round-off errors from the internal computation. The improved readability and maintainability of the implementation comes at the expense of efficiency, as the integers in the numerator and denominator grow quickly. Chapter 3 and chapter 8 present an application of affine arithmetic in a runtime technique, and use the floating-point implementation. The remaining work which is of static nature and considers shorter arithmetic expressions at a time employs the rational version. In the rest of this chapter we will discuss the floating-point implementation, the rational one follows straight-forwardly.

2.3.1 Nonlinear Computations

Affine operations are computed by Equation 2.2. For nonlinear operations like multiplication, inverse or square root, this formula is not applicable and the operations have to be approximated. Multiplication is derived from expanding and multiplying two affine forms:

$$\hat{x}\hat{y} = x_\circ y_\circ + \sum_{i=1}^{n} (x_\circ y_i + y_\circ x_i)\epsilon_i + (\eta + \iota)\epsilon_{n+1}$$

where ι collects the internal errors when applicable and η is an over- approximation of the nonlinear contribution. To compute the latter, several possibilities exist of varying degree of accuracy. The simplest way is to compute η as $\eta = \mathsf{radius}(\hat{x}) \cdot \mathsf{radius}(\hat{y})$. When $\mathsf{radius}(\hat{x}), \mathsf{radius}(\hat{y}) \ll 1$, this is sufficiently accurate as the nonlinear contribution η will then also be small. For larger ranges, the over-approximation of the nonlinear part becomes significant. Our implementation computes η as:

$$\begin{aligned}\eta &= \max_{\epsilon_i\in[-1,1]}\left|\sum_{1\le i,j,\le n} x_i y_j \epsilon_i \epsilon_j\right| \\ &= \sum_{1\le i,j,\le n} |x_i y_j| \\ &= \sum_{i=1}^{n} |x_i y_i| + \sum_{i<j}^{n} |x_i y_j + x_j y_i|\end{aligned}$$

Division $\hat{x}/\hat{y}$ is computed as $\hat{x}\cdot(1/\hat{y})$. For the approximation of unary functions, the problem is the following: given $f(x)$, find α, ζ, δ such that

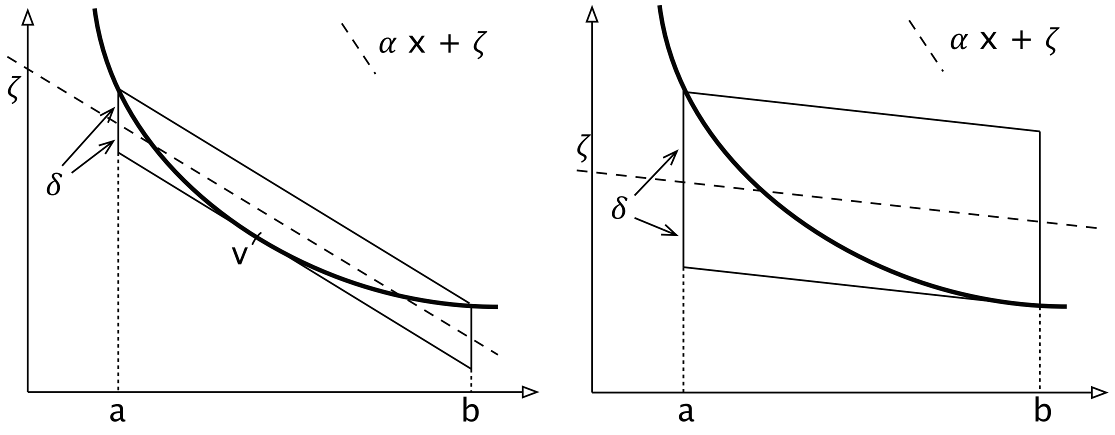
$$[f(x)] \subset [\alpha x + \zeta \pm \delta]$$

where $[f(x)]$ denotes the sound bound (interval) of the range of f, given a range for its input x. α and ζ are constants determined by a linear approximation of the function f and δ represents all (round-off and approximation) errors committed, thus yielding a rigorous bound. [51] suggests two approximations for computing α, ζ, and δ: a Chebyshev (min-max) or a min-range approximation. Figure 2.1 illustrates these two on an example function. For both approximations, the algorithm first computes the interval represented by $[\hat{x}] = [a, b]$ and then works with its endpoints a and b. In both cases we want to compute a bounding box around the result, by computing the slope (α) of the dashed line, its intersection with the y-axis (ζ) and the maximum deviation from this middle line (δ). In the following we assume that the function is monotone over the interval of approximation $[a, b]$ and does not cross any inflection or extreme points. Where this is not the case, our library resorts to computing the result in interval arithmetic and converting it back into an affine form.

Min-range Compute the slope α of the function f (which we want to approximate) at one of the endpoints a or b. Compute two lines with slope α that go through the points $(a, f(a))$ and $(b, f(b))$ respectively. Fix ζ to be the average of the y-intercepts of the two lines. Compute δ as the maximum difference between f and the line with slope α and going through ζ, which occurs at either a or b, because the sign of the derivative of f does not change over $[a, b]$ by assumption.

Chebyshev Compute the slope α of the line through both $f(a)$ and $f(b)$. This gives one bounding side of the wanted 'box' (parallelepiped). To find the opposite side, compute the point where the curve takes on the same slope again. Again, compute ζ as the average of the intersections of the two lines and δ as the maximum deviation at either the middle point v, a or b.

In general, the Chebyshev approximation computes smaller parallelepipeds, especially if the slope is significantly different at a and b. However, it also needs the additional computation of the middle point. Especially for transcendental functions like arccos, arcsin, etc., this



(a) Chebyshev (b) Min-range

Figure 2.1 – Linear approximations of the inverse function

can involve quite complex computations which are all committing internal round-off errors. On large input intervals, like the ones considered in [51, 52], these are (probably) not very significant. However, when keeping track of round-off errors, our library deals with intervals on the order of machine epsilon. In particular, the computation of v introduces roundoff errors, but because it lies between $[a, b]$ it is not clear whether it should be rounded up or down for soundness. In fact, when we used rounding to nearest, which is closest to the true point, Chebyshev approximations kept returning unexpected and wrong results. We thus concluded that min-range is the better choice. As discussed in [52], the Chebyshev approximation would be the more accurate choice in long running computations, however we simply found it to be too numerically unstable for our purpose. To our knowledge, this problem has not been acknowledged before.

Error estimation for nonlinear library functions like $\mathsf{log}, \mathsf{exp}, \mathsf{cos}$, etc. requires specialized rounding, because the returned results are correct to 1 ulp only for the standard Java/Scala `math` library [88], and hence are less accurate than arithmetic operations, which are correct to within 1/2 ulp. The directed rounding procedure is thus adapted in this case to produce larger error bounds to make it is possible to analyze code with the usual Scala mathematical library functions without modifications.

Soft Policy to Avoid Too Many False Warnings

Our solution follows the 'soft' policy advocated in [51], whereby slight domain breaches for functions that work only on restricted domains are attributed to the inaccuracy of our over-approximations and are ignored. For example, with a 'hard' policy computing the square root of $[-1, 4]$ results in a run-time error, as the square root function is not defined on all of the input interval. It is possible, however, that the true interval (in a real semantics) is $[0, 4]$ and the domain problem is just a result of a previous over-approximation. In order to not interrupt computations unnecessarily with false alarms, a 'soft' policy computation will give the result $[0,2]$. Note, that our library nonetheless generates warnings in these cases, so the policy only affects the tool's ability to continue a computation in ambiguous cases, but not its rigorousness. The techniques in chapter 6 take a different approach, and report such breaches as (potential) errors.

2.4 Tracking Roundoff Errors with Affine Arithmetic

Affine arithmetic as described in section 2.3 can be used to compute sound range bounds. We can also use it to track round-off errors by computing the maximum absolute round-off at each arithmetic operation and adding it to the affine form with a fresh noise symbol. Note the difference between the internal errors, which are due to computation of the range, and these added round-off errors, which stem from modeling the finite-precision computation. In fact, there exist different ways of using affine arithmetic for tracking round-off errors and we identify and implement two of them.

2.4.1 Different Interpretations of Computations

When using a range-based method like interval or affine arithmetic, it is possible to have different interpretations of what such a range represents. We consider the following three different interpretations of affine arithmetic. The first one is also the interpretation from [51].

Interpretation 2.1 (Original Affine Arithmetic) *In the original affine arithmetic, an affine form $\hat{x}$ represents a range of real values, that is $[\hat{x}]$ denotes an interval $[a,b]$ for $a,b \in \mathbb{R}$. Note that this interpretation does not consider roundoff errors.*

Interpretation 2.2 (Exact Affine Arithmetic) *In exact affine arithmetic $\hat{x}$ represents* **one** *finite-precision value and its deviation from an ideal real value due to roundoff. That is, if a real-valued computation computed $x \in \mathbb{R}$ as the result, then for the corresponding computation in finite precision it holds that $x \in [\hat{x}]$.*

We realize interpretation 2.1 by requiring that the central value $x_\circ$ is equal to the actually computed finite-precision value at all times, in order to guarantee sound computation of round-off errors. This interpretation is the basis for our `AffineFloat` data type, which we present in chapter 3.

Interpretation 2.3 (Global Affine Arithmetic) *In global affine arithmetic $\hat{x}$ represents a range of finite-precision values, that is $[\hat{x}]$ denotes an interval $[a,b]$ for $a,b \in \mathbb{F}$.*

Interpretation 2.3 is used in our `SmartFloat` data type. Note the difference between this interpretation and interpretation 2.1: we are computing a sound bound of the result of a finite-precision computation, with committed round-off errors, whereas in the standard formulation we are computing a sound range of a real-valued computation without committed roundoff errors (but may be *computing* it with e.g. floating-points).

Usually implementation issues are of minor interest, however in the case of finite precision computations they are an important aspect: our implementation itself uses floating-point arithmetic to compute round-off errors, and we are faced with the very same problems in our own implementation that we are trying to quantify. We have identified two main challenges when implementing an affine arithmetic-based round-off error analysis:

- Implementing interpretation 2.2 with standard affine arithmetic as presented in section 2.3 is unsound.
- The resulting accuracy is unsatisfactory if affine arithmetic is implemented directly as defined previously.

In the following, we will discuss how to adapt affine arithmetic for our two interpretations, and our solution to the accuracy challenge.

2.4.2 Tracking the Computation from a Single Input Value

A possible use of affine arithmetic for keeping track of round-off errors is to represent each finite precision number by an affine form. The central value denotes the actually computed finite precision value and the noise terms collect the accumulated round-off errors. That is, the central value $x_\circ$ is exactly the finite-precision value and the noise symbols x_i represent the deviation due to round-off errors and approximation inaccuracies from non-affine operations. One expects to obtain tighter bounds than with interval arithmetic, especially when a computation exhibits many correlations between variables. However, a straightforward application of affine arithmetic in the original formulation is not always sound with respect to this interpretation. Namely, standard affine arithmetic takes the liberty of choosing a convenient central value in a range and does not necessarily preserve the correspondence with `Double`. In particular, non-affine operations such as division or trigonometric functions can shift the central value away from the actually computed finite-precision value. This correspondence is important, however, as the round-off for floating-points is computed according to Equation 2.1, i.e. by multiplication of the *new* central value by the machine epsilon ϵ_M. If the central value does not equal the actual floating-point value, the computed round-off will be that of a different result. For fixed-point arithmetic, this dependence is also present, although a difference is only visible when the two values happen to have different fixed-point formats. Affine operations maintain the invariant that the computed finite-precision value is equal to the central value. However, non-affine operations defined by computing α, ζ and δ such that the new affine form is $\hat{z} = \alpha \cdot \hat{x} + \zeta + \delta\epsilon_{n+1}$ do not necessarily enforce this. That is, in general (and in most cases), the new $z_\circ$ will be slightly shifted ($z_\circ \neq \alpha \cdot x_\circ + \zeta$). Usually, this shift is not large, however soundness cannot be guaranteed any more.

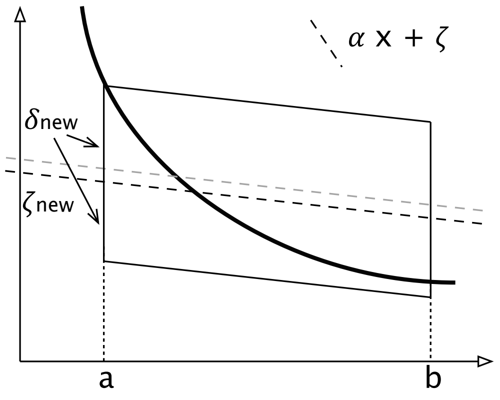


Figure 2.2 – Modified min-range approximation

Our implementation therefore provides a modified version of affine arithmetic that ensures the correspondence between the central value and the computed finite-precision value and thus ensures soundness.

Guaranteeing Soundness of Error Estimates

Our solution is illustrated in Figure 2.2. For non-linear operations, the new central value is computed as $z_\circ = \alpha \cdot x_\circ + \zeta$ and we want $f(x_\circ) = \alpha \cdot x_\circ + \zeta$, where $f(x_\circ)$ is the value actually computed in finite precision. Hence, our library computes ζ as

$$\zeta_{new} = f(x_\circ) - \alpha \cdot x_\circ$$

instead of as the average y-intercept as before. δ is then correspondingly computed with respect to this ζ_{new}. The min-range approximation computes for an input range $[a, b]$ an enclosing parallelepiped of a function as $\alpha \cdot x + \zeta_{new} \pm \delta$ that is guaranteed to contain the image of the nonlinear function from this interval as computed in finite precision.

Suppose that $\zeta_{new} = f(x_\circ) - \alpha \cdot x_\circ$, with α computed at one of the endpoints of the interval. Because we compute the deviation δ with outwards rounding at both endpoints and keep the maximum, we soundly over-approximate the function f in finite-precision semantics. Clearly, this approach only works for input ranges where the function in question is monotonic and does not have extreme or inflection points. We implement this procedure for floating-point arithmetic as part of a runtime library (see chapter 3), where we use that by the Java API [88] the implemented floating-point library functions are guaranteed to be semi-monotonic, i.e. whenever the real function is non-decreasing, so is the floating-point one. This technique applies also for fixed-point arithmetic division. It is clear from Figure 2.2 that our modified approximation computes a bigger parallelepiped than the original min-range approximation. However, in this case, the intervals are very small to begin with, so the over-approximations do not have a big effect on the accuracy of our library.

```
def /(other: AffineForm): AffineForm = other match {
  case AffineForm(y0, ynoise) =>
    val (yloD, yhiD) = other.interval
    val (ylo$_{DD}$, yhi$_{DD}$) = other.interval$_{DD}$
    if(yloD <= 0.0 && yhiD >= 0.0) return FullForm //division by zero

    if(ynoise.size == 0.0) { //exact
      val inv = 1.0/y0
      if((1.0 $/^{\downarrow}$ y0) == (1.0 $/^{\uparrow}$ y0))
        // multiplication with hint: x0/y0
        return this * (AffineForm(inv, Queue.empty), x0/y0)
      else
        return this * (AffineForm(inv, Queue(roundOff(inv))) , x0/y0)
    }

    /* Calculate the inverse. */
    val (a, b) = (min(|ylo|, |yhi|), max(|ylo|, |yhi|))
    val (ad, bd) = (min(|yloD|, |yhiD|), max(|yloD|, |yhiD|))

    // slope at right endpoint
    val alpha = -1.0 $/_{DD}$ ( b $*_{DD}$ b )

    // deviation of approximation from true value at end points of interval
    val dmax = $\mathbb{DD}$(1.0 $/^{\uparrow}$ ad) $-^{\uparrow}_{DD}$ (alpha $*^{\downarrow}_{DD}$ a))
    val dmin = $\mathbb{DD}$(1.0 $/^{\downarrow}$ bd) $-^{\downarrow}_{DD}$ (alpha $*^{\uparrow}_{DD}$ b))

    // use hint to compute zeta
    var (zeta, rdoff) = computeZeta(1.0/y0, alpha, y0)
    if(yloD < 0.0) zeta = -zeta

    // total deviation
    val delta = max( zeta $-^{\uparrow}$ dmin, dmax $-^{\uparrow}$ zeta ) $+^{\uparrow}_{DD}$ rdoff

    // iota includes all internal roundoffs from the mult. by alpha
    val inverse = AffineForm(1.0/y0, alpha * ynoise + iota)

    /* Multiply x * 1/y */
    return this * (inverse, x0/y0)

  case EmptyForm => EmptyForm
  case FullForm => FullForm
}
```

Figure 2.3 – Division operation for tracking one operation

Figure 2.3 shows the pseudocode for the division operation which is representative of our nonlinear approximation computation. We further support the following mathematical functions for floating-point arithmetic: sqrt, log, exp, cos, sin, tan, arccos, arcsin, arctan. The code shows the floating-point implementation of affine arithmetic for the case of tracking a double precision floating-point computation. The methods are implemented inside the AffineForm class

```
    class AffineForm(x0: Double, xnoise: Queue)
```

where `x0` is the central value and `xnoise` is the list of noise terms. We call these `y0` and `ynoise` respectively for the second operand in the binary division operation. `EmptyForm` is the result of an invalid operation and is thus the affine arithmetic equivalent of NaN. `FullForm` represents the interval $[-\infty, \infty]$. The subscript DD denotes arithmetic in double-double precision, which we discuss further later. We also use directed rounding denoted by $^{\downarrow}$ and $^{\uparrow}$ for rounding towards $-\infty$ and ∞ respectively. Division is computed as $x * \frac{1}{y}$. Our algorithm calls a special multiplication method that takes a "hint" to ensure that the central value equals the double value that would be computed. This is necessary as $x * \frac{1}{y}$ is not necessarily equal to $\frac{x}{y}$ in floating-point arithmetic (although the difference is in general very small). `computeZeta` computes ζ such that $z_\circ = \alpha \cdot x_\circ + \zeta$ holds, and also returns the internal round-off error committed from this computation.

2.4.3 Tracking Ranges of Inputs

The instantiation of affine arithmetic just described provides a way to estimate round-off errors for one single computation. It provides reasonably tight bounds for the most common mathematical operations and is fast enough to be applied in applications such as LU decompositions and Fast Fourier transforms (see subsection 3.2.1). It can be used to provide some intuition about the behavior of a calculation and is a better alternative for interval arithmetic which is a common choice in validated computation. It does not provide, however, any guarantee as to how large the errors would be if one chose (even slightly) different input values or constants.

Unfortunately, a straightforward reinterpretation of neither the original affine arithmetic, nor the modified version for `AffineFloat` provide sound error estimates when tracking a finite-precision computation for a range of inputs. When tracking a *range* of finite-precision numbers and computing the worst-case round-off errors of each computation, we need to consider the round-off errors for *all* values in the range, not only for the central value as was the case for `AffineFloat`. In addition, the non-linear approximation algorithm does not explicitly compute the round-off errors, they are implicitly included in the computed δ. If we now have input values given by (possibly wide) ranges, the computed δ will be so large that no round-off estimate from them is meaningful.

Our solution is the following. We represent every value by the following pair:

$$(x_\circ + \sum x_i \epsilon_i + \sum x_i u_i, \sum r_i \rho_i) \tag{2.3}$$

The first element of the tuple is an affine form representing a range of finite-precision values and the second element tracks the maximum committed round-off errors for all values in that range.

$x_\circ \in \mathbb{F}$ is the central value as before, but we use two different noise terms to capture the range: $x_i \epsilon_i$ and $x_i u_i$. We mark noise that comes from user-defined errors by special noise symbols u_i, which we call *uncertainties.* For instance, when the user specifies an input value as 3.14 ± 0.5, $x_\circ = 3.14$ and the single noise term is $0.5u_1$ such that the represented range is $[2.65, 3.65]$. ϵ_i's capture computation artifacts, for example from nonlinear approximations. Long running computations accumulate many noise terms, which we "compact" for performance reasons by essentially combining several noise terms into one. Since during this process we loose correlations, keeping the user defined uncertainties separate allows us to distinguish and preferably preserve these. The details of this procedure are given in subsection 2.4.4. $r_i \rho_i$ is a zero-centered affine form whose *error terms* keep track of the round-off errors committed. The sum $\sum |r_i|$ gives a sound estimate on the current maximum committed round-off error for all values within the range represented by the first part of the triple.

For the affine form represented by the first two elements of the tuple, the calculations are performed as described in section 2.3, except that the round-off errors are computed as discussed below. In addition, we need to define how to propagate the already computed round-off errors. For the original and our exact affine arithmetic this propagation has been automatically taken care of, but for tracking a range of values soundly we need to modify this procedure.

Computation of Roundoffs

To compute the maximum round-off error of an operation which is sound for a range of values, our library first determines the new range $[a, b]$. It uses either Equation 2.2 for affine operations or the min-range approximation for nonlinear ones. For floating-point arithmetic, we use the fact that arithmetic operations $+, -, *, /, \surd$ are rounded correctly, that is, the round-off is computed as

$$\frac{1}{2}\mathsf{ulp}(\max(|a|, |b|))$$

where ulp is a function provided by the `java.lang.Math` package. For other operations, correct to within 1 ulp, the round-off error is simply $\mathsf{ulp}(\max(|a|, |b|))$. This round-off error computation can also easily be adapted to other rounding modes, but we only consider rounding to nearest in this thesis as it is the the commonly used default rounding mode and the only one available on the JVM. For fixed-point arithmetic, the range $[a, b]$ and the given bit length determine the best fixed-point format. The round-off error is then the quantization error of this format.

Propagation of Roundoffs

Already committed and accumulated errors $\sum_{i=1}^{n} r_i \rho_i$ have to be propagated correctly for each operation. We denote the error terms of an affine form $\hat{x}$ by $\hat{\rho_x}$ and similarly for the other noise terms.

Affine The propagation is given straightforwardly by Equation 2.2. That is, if the operation involves the computation $\alpha\hat{x} + \beta\hat{y} + \zeta$, the errors are transformed as $\alpha\hat{\rho_x} + \beta\hat{\rho_y} + (\iota + \kappa)\rho_{n+1}$, where ι corresponds to the internal errors committed and κ to the new round-off error. That this is indeed correct can be seen from the fact that the computation would have been identical, if those error terms would have been noise terms of the $\hat{x}$ and $\hat{y}$.

Multiplication From multiplying out (we leave out $\hat{u}$'s for simplicity)

$$\begin{aligned}\hat{x} * \hat{y} &= (x_\circ + \hat{e_x} + \hat{\rho_x}) * (y_\circ + \hat{e_y} + \hat{\rho_y}) \\ &= x_\circ * y_\circ + (x_\circ\hat{e_y} + y_\circ\hat{e_x} + \hat{e_x}\hat{e_y}) + (x_\circ\hat{\rho_y} + y_\circ\hat{\rho_x} + \hat{e_x}\hat{\rho_y} + \hat{e_y}\hat{\rho_x} + \hat{\rho_x}\hat{\rho_y})\end{aligned}$$

The last term is the contribution to the new error. The linear part $(x_\circ\hat{\rho_y} + y_\circ\hat{\rho_x})$ is computed as before for the propagation of ranges by multiplication by x_0 and y_0. The nonlinear part $(\hat{e_x}\hat{\rho_y} + \hat{e_y}\hat{\rho_x} + \hat{\rho_x}\hat{\rho_y})$ poses the difficulty that it involves cross terms between the noise and error terms. We compute the new η_e as follows:

$$\eta_e = \mathsf{radius}(\hat{x}) \cdot \mathsf{radius}(\hat{\rho_y}) + \mathsf{radius}(\hat{y}) \cdot \mathsf{radius}(\hat{\rho_x}) + \mathsf{radius}(\hat{\rho_x}) \cdot \mathsf{radius}(\hat{\rho_y})$$

Note that this produces an over-approximation, because some of the errors from the error terms are also included in the noise terms.

Other Non-affine Because the nonlinear function approximations compute α, ζ and δ, the propagation of errors reduces to the affine propagation with one modification. The factor used to propagate the round-off errors must be instead of α the maximum slope of the function over the given input range to ensure soundness. Because this value does not necessarily equal α, we compute this factor separately.

Additional Errors

Additional domain-specific errors can also be added to an affine form to be tracked for the rest of the computation. One source of these errors are measurement inaccuracies in systems that interact with the physical world through sensors. Another source are truncation errors, i.e. the differences between a (hypothetical) analytical solution and a result computed with an iterative method. Given $\hat{x} = (x_\circ + \sum x_i\epsilon_i + \sum x_i u_i, \sum r_i\rho_i)$ and the error to be added $\hat{y} =$

$(y_\circ + \sum y_i \epsilon_i + \sum y_i u_i, \sum s_i \rho_i)$, the resulting affine form is given by

$$\hat{z} = (x_\circ, \sum x_i \epsilon_i + \sum x_i u_i + (|y_\circ| + \text{radius}(\sum y_i \epsilon_i + \sum y_i u_i))\epsilon_{n+1}, \\ \sum r_i \rho_i, +(\text{radius}(\sum s_i \rho_i)\rho_{m+1})$$

That is, the maximum magnitude of the error is added as a new noise term, and the maximum magnitude of the round-off committed when computing this error is added as a new error term.

2.4.4 Achieving Accuracy

It turns out that even when choosing the min-range approximation with input ranges with small widths (order 10^{-10} and smaller), computing the result of a nonlinear function in interval arithmetic gives better (more accurate) results. The computation of α and ζ in our approximation cannot be changed for soundness reasons, but it is possible to optimize the computation of δ. Some of our applications use a rational data type in the implementation for this reason, but for a runtime application performance becomes an issue due to long integers. In order to avoid arbitrary precision libraries also for performance reasons, our library uses double-double precision (denoted as $\mathbb{DD}$) as a suitable compromise for tracking round-off errors for a computation performed in double floating-point precision. For other precisions, the internal precision needs to be adjusted accordingly. Each value is represented by two standard double precision values. Algorithms have been developed and implemented [131, 49] that allow the computation of standard arithmetic operations with only ordinary floating-points, making the performance trade-off acceptable. We found that using extended precision reduces the internal round-off errors sufficiently for most nonlinear function approximations. To ensure soundness we also apply directed rounding to double-double computations.

Precise Handling of Constants A single value a is represented in a real-valued interval semantics as the point interval $[a, a]$ or in affine arithmetic as $\hat{x} = a$ without noise terms. This no longer holds for finite-precision values that cannot be represented exactly in the underlying finite representation. Our library tests each value for whether it can be represented or not in a given precision and adds noise terms only when necessary. Adding round-off errors only when constants cannot be represented exactly limits the over-approximations committed and provides a more accurate analysis.

Managing Noise Symbols in Long Computations The runtime performance of our library depends on the number of noise terms in each affine form, because each operation is at least linear in the number of noise terms. Hence, an appropriate “compacting” strategy for noise symbols becomes crucial for performance in longer calculations. Compacting too little means that our approach becomes too slow, whereas compacting too much means the loss of too much correlation information.

The goal of compaction is to take as input a list of noise terms and output a new list with fewer terms, while preserving the soundness of the round-off error approximation and, ideally, keeping the most important correlation information. Our library performs compaction by adding up the absolute values of the smaller terms and introducing them as a fresh noise symbol along with the (unchanged) remaining terms. We propose the following strategy to compute the new noise terms, which is used when the number of noise terms exceeds a certain user-defined threshold and attempts to reduce the number below it. In our implementation, all the parameters can be adjusted for a specific application.

i) Compact all error terms smaller than a given threshold which identifies (likely) internal errors. For instance, for the analysis of double floating-point precision computations, we set this threshold as 10^{-33}.

ii) Compute the average (avrg) and the standard deviation (stdDev) of the remaining noise terms. Compact all terms smaller than $\mathsf{avrg} \cdot a + \mathsf{stdDev} \cdot b$ and keep the rest. The factors a and b are user- controllable positive parameters, and can be chosen separately for each computation. (The result is sound regardless of the particular values.)

iii) In some cases the above steps are still not enough to ensure that the number of symbols is below the threshold. This occurs, for example, if nearly all errors have the same magnitude. If our library detects this case, it repeats the above procedure one more time on the newly computed noise terms. In our examples, at most two iterations were sufficient. In pathological cases in which this does not suffice, the library compacts all noise symbols into a single one.

2.4.5 Correctness

One condition for the presented round-off error analysis to be sound is that the operations are made in *exactly* the order as given in source code and without optimizations or fused-multiply instructions. We have enforced this for our code by using the `strictfp` modifier for the calculations. We are able to avoid several pitfalls related to floating-point numbers [23, 118] by writing our library in Scala and not for example in C, as the JVM is not as permissive to optimizations that may alter the actual execution of code.

The correctness of each step of the interval or affine arithmetic computation implies the correctness of our overall approach: for each operation in interval or affine arithmetic the library computes a rigorous over-approximation, and thus the overall result is an over-approximation. This means, that for all computations, the resulting interval is guaranteed to contain the result that would have been computed in an ideal real semantic.

The use of assertions in our implementation certifying that certain invariants always hold support the correctness of our implementation. Example invariants for the case when we track only one computation include the statement that the computed double precision value has to be exactly the same as the central value of the affine form, a prerequisite for our round-off analysis.

In addition, we have tested our library extensively on several benchmarks (see section 3.2) and our implementation of nonlinear functions against the results from 30 digit precision results from Mathematica.

2.5 Comparison with Fluctuat

We are aware of one tool, Fluctuat [72], which is able to quantify soundly and automatically roundoff errors. It is an abstract interpretation based static analysis tool whose core analysis, although developed independently, relies on affine arithmetic in a very similar way to our design. The techniques described in this chapter for tracking a single computation are dynamic in nature since we are essentially running a computation with some additional information computed on the side. In contrast, the technique for computing worst-case error bounds for a whole range of inputs we view as static and it is this technique that we compare against Fluctuat. The `SmartFloat` data type implementing this technique (see chapter 3) is a kind of hybrid, in the sense that we execute the control flow dynamically, but bound the roundoff errors in a static way. Our techniques in later chapters (6 and 7) generalize this into a proper static analyzer.

As our technique, Fluctuat separates the range from the error estimation and keeps track of both with affine arithmetic, but also keeps a floating- point interval on the side. We did not find computing an additional interval side-by-side to provide more accuracy in general, so we keep only the two affine forms for efficiency reasons. Our design differs from Fluctuat's in a number of details and we review them briefly here.

Fluctuat uses an arbitrary precision library for internal computations. We use either double-double floating-point precision for our runtime libraries, or a rational data type for the implementation in static approaches. Our experiments in chapter 3 confirm that this precision is sufficient for a runtime approach and we believe that it is preferable over arbitrary precision from an efficiency standpoint. For the use in a static analysis tool (chapters 5-7), we chose a rational data type for accuracy and compatibility reasons, as we interface with an SMT solver which internally also uses rationals. Efficiency of the numerical data type did not appear as a bottleneck in our experiments (compared with the time taken by the SMT solver).

Our techniques are targeted towards scientific computing applications so that we support many transcendental functions. Fluctuat has been developed with industrial embedded code in mind and thus supports only standard arithmetic with square root. Where our nonlinear operations use the min-range approximation, Fluctuat approximates division and square root with a first-order Taylor approximation. Furthermore, we keep all higher-order terms by default and compact as needed, whereas Fluctuat collects all higher-order contributions in a single noise term. We have not performed a systematic and direct comparison of the individual features but this would certainly be an interesting project in the future. In our experiments, presented in chapter 3, we have not found big differences in that sometimes our implementation provides better results and sometimes it was Fluctuat's. Overall, we believe that more fundamental differences and extensions, such as those we present in chapter 6 have a bigger impact.

Conclusion

We have presented an adaptation of affine arithmetic for the estimation of round-off errors. In fact, we discuss two such adaptations, one for the case of tracking a single value through a computation, including it's round-off errors, which can serve as a replacement for the traditionally used interval arithmetic. The second adaptation shows how to use two affine forms to track the ranges for a computation where the inputs are interval-valued and their corresponding worst-case round-off errors. We further describe how to implement nonlinear operations, including transcendental functions, with floating-point arithmetic such that it computes sound and correct results. We are not aware of other work that can do this computation for very small ranges (order of machine epsilon) and with a performance that is acceptable for runtime applications.

3 Tracking Roundoff Errors at Runtime

Previously, we have described our roundoff error computation which is applicable to both floating-point and fixed-point arithmetic. This chapter (based on the paper [44]) presents an application of affine arithmetic to track round-off errors in double precision floating-point computations at runtime. We chose floating-points as the target, since `Doubles` have established themselves as the default data type for implementing software that approximates real-valued computations. Since the definition of the IEEE 754 standard [84], these computations have also become reliable and, to a certain extent, portable [118].

The general idea is to replace the floating-point data types with a smarter version which automatically keeps track of the accumulating errors. Our smarter data types come in two flavors: `AffineFloat` tracks a single computation and corresponds to the affine arithmetic interpretation of subsection 2.4.2, and `SmartFloat` which tracks a range of values and their worst-case round-off errors, corresponding to the interpretation in subsection 2.4.3. `AffineFloats` can often provide tighter error estimates than `SmartFloats`, at the expense of lack of generality. Our data types improve over a computation performed in interval arithmetic in terms of accuracy and generality.

We implemented our data types as a runtime library in Scala [125] for tracking round-off errors for double floating-point arithmetic. A seamless integration allows users to apply our library with very few changes to existing floating-point code. A runtime library is especially useful in the case of floating-point numbers, because the knowledge of exact values enables us to provide a tight analysis. Also, with our tight integration it is possible to use any Scala construct, thus not restricting the user to some subset that an analyzer front-end can handle. The library presented is specialized for double floating-point precision, as this is the common default choice in scientific computing, but the underlying techniques are general and can be applied to other floating-point or fixed-precisions equally well. Our code is open-source and available from `github.com/malyzajko/ceres`.

We evaluate our implementation on a number of benchmarks from scientific computing, and compare our results against those computed by interval arithmetic and the tool called

Fluctuat [72]. Fluctuat is an abstract interpretation bases static analyzer and is the only other tool we are aware of for soundly estimating round-off errors. However, while our code has been always open-source, Fluctuat's binary became available only recently, hence we compare our results against the newest version of Fluctuat.

3.1 Integration into Scala

Our library provides the wrapper types `AffineFloat` and `SmartFloat` that are meant to replace the `Double` type in the part of a program that the user wishes to check. For instance, the code in Figure 3.1 shows a short function computing the area of a triangle with sides `a, b` and `c` using `SmartFloats`. All that is needed to put our library into action are two import statements and the replacement of `Double` types by one of our `AffineFloat` or `SmartFloat` types. Any remaining conflicts are signaled by the compiler's strong type checker. The computed errors can be accessed in different ways, with the methods `absError`, `interval` and `toString`. `absError` simply returns the maximum absolute error, `interval` returns an interval of the variable which is sound with respect to all uncertainties and roundoffs, and finally `toString` provides a pretty-printed representation of the computed value and its associated error.

Our new data types handle definitions of variables and the standard arithmetic operations, as well as many `scala.math` library functions, including the most useful ones:

- log, expr, pow, cos, sin, tan acos, asin, atan
- abs, max, min
- constants Pi (π) and E (e)

The library also supports special values NaN and $\pm\infty$ with the same behavior as IEEE floating-point arithmetic. To accomplish such an integration, we needed to address the following aspects.

```
import ceres.smartfloat.SmartFloat
import SmartFloat._

...

def triangleTextbook(a: SmartFloat, b: SmartFloat, c: SmartFloat): SmartFloat = {
  val s = (a + b + c)/2.0
  sqrt( s*(s-a)*(s-b)*(s-c))
}
```

Figure 3.1 – Computing the area of a triangle with `SmartFloats`

Operator overloading Developers should be able to use the usual operators `+, -, *,/` without having to rewrite them as functions, e.g as `x.add(y)`. Scala supports operator overloading in the sense that `+, -, *, /` are regular method calls, and `x.m(y)` can equivalently be written as `x m y` [125], resulting in the familiar syntax for arithmetic.

Symmetric equals Comparisons between our data types and regular numeric Scala types should be symmetric, that is, the following should hold

```
val x: SmartFloat = 1.0
val y: Double = 1.0
assert(x == y && y == x)
```

In Scala, == delegates to the `equals` method, if one of the operands is not a primitive type. However, this is not enough to ensure a symmetric comparison, as `Double`, or any other built-in numeric type, cannot compare itself correctly to a `SmartFloat`. For this case, Scala provides the trait `ScalaNumber` which has a special semantics in comparisons with ==. If `y` is of type `ScalaNumber`, then both `x == y` and `y == x` delegate to `y.equals(x)` and thus the comparison can be correctly and symmetrically handled inside our data type classes.

Mixed arithmetic Developers should be able to freely combine our data types with Scala's built-in primitive types, as in the following example

```
val x: SmartFloat = 1.0
val y = 1.0 + x
if (5.0 < x) {...}
```

This is made possible with Scala's implicit conversions, strong type inference and companion objects [125]. In addition to the class `SmartFloat`, the library defines the (singleton) object `SmartFloat`, which contains an implicit conversion such as

```
implicit def double2SmartFloat(d : Double): SmartFloat = new SmartFloat(d)
```

As soon as the Scala compiler encounters an expression that does not type check, but a suitable conversion is present in scope, the compiler inserts an automatic conversion from the `Double` type in this case to a `SmartFloat`. Implicit conversions allow a `SmartFloat` to show a very similar behavior to the one exhibited by primitive types and their automatic conversions. The code for `AffineFloat` is analogous.

Library functions Having written code that utilizes the standard mathematical library functions, developers should be able to reuse their code without modification. Our library implements these functions in the companion `AffineFloat` or `SmartFloat` objects. It is thus possible to write code such as

```
val x: SmartFloat = 0.5
val y = sin(x) * Pi
```

Concise code For ease of use and general acceptance it is desirable to be able to skip the **new** keyword in definitions and to simply write SmartFloat(1.0). This is made possible in Scala where the special SmartFloat.apply(...) method (also placed in the companion object) is syntactic sugar for SmartFloat(...).

3.1.1 Conditionals

Path consistency is ensured by the compare method of the AffineFloat and SmartFloat data types which takes uncertainties into account. The user can choose between two different behaviors: if a comparison $x < y$ cannot be decided for sure due to uncertainties on the arguments, either a warning is printed or an exception is thrown. In the case where the user chooses to throw an exception, we provide two methods for handling such a failure gracefully but yet explicitly:

```
def certainly(b : ⇒ Boolean) : Boolean = {
  try b catch ComparisonUndeterminedException ⇒ false
}
def possibly(b : ⇒ Boolean) : Boolean = {
 try b catch ComparisonUndeterminedException ⇒ true
}
```

If we cannot be sure a boolean expression involving SmartFloats is true, we assume it is **false** in the case of **certainly**, and that it is **true** in the case of **possibly**. Hence, the following identity holds:

```
if (certainly(P)) T else E   ⇔   if (possibly(!P)) E else T
```

We show a possible application in subsection 3.2.3.

3.2 Experimental Results

We have selected several benchmarks for evaluating our library. Many of them were originally written in Java or C and we ported them to Scala as faithfully as possible. Once written in Scala, we found that changing the code to use our AffineFloat or SmartFloat types instead of Double is a straightforward process and needs only few manual edits. Scala compiler's type checker was particularly helpful in this process.

3.2.1 Accuracy of AffineFloat

We evaluate the accuracy of our AffineFloat data type against a round-off error estimation implementation based on interval arithmetic as described in section 2.2. For this purpose, our library also provides the IntervalFloat data type, with the same signatures as AffineFloat, but with an back-end implementation using intervals implemented with floating-point arithmetic and directed rounding. Our chosen benchmarks for this evaluation are the following:

Nbody simulation is a benchmark from [6] simulating the orbits of the planets Jupiter, Saturn, Uranus and Neptune around the Sun using a simple symplectic-integrator. Since the benchmark includes many variables that we could potentially measure, we choose the total energy of the system as our measured value. The energy changes both due to truncation errors in the integration method and due to round-off errors. Here, we measure the latter.

Spectral norm is a benchmark from [6] that calculates the spectral norm of the infinite matrix with entries $a_{11} = 1$, $a_{12} = \frac{1}{2}$, $a_{21} = \frac{1}{3}$, $a_{13} = \frac{1}{4}$, $a_{22} = \frac{1}{5}$, $a_{31} = \frac{1}{6}$, etc. The benchmark uses the power method, which is an iterative method, and we choose to study the error on the single output of this function for different numbers of iterations.

Jacobi Successive Over-relaxation (SOR) is taken from the SciMark 2.0 benchmark set [130] and is an iterative numerical method for solving a linear system of equations. We report errors for different numbers of iterations. The input is a random matrix of size $100x100$.

LU decomposition is also taken from SciMark, and performs a LU decomposition of matrix A with or without pivoting, which can then be used to solve a system of linear equations $Ax = b$. Pivoting attempts to select larger elements in the matrix during factorization to avoid numerical instability. We compare the errors on the solution x.

FFT performs a one-dimensional Fast Fourier Transform. This benchmark is also part of SciMark. We perform the forward and backward transform and report the maximum absolute error on the recovered vector.

For the last three benchmarks, where the input data is vector or matrix-valued, we report the maximum error over all entries. We use matrices with random entries with maximum value 10.0 as inputs. We ran the code with different initial seeds and can confirm that the results we present here are consistent across runs.

Table 3.1 shows a comparison between the absolute errors computed by the `IntervalFloat`, `AffineFloat` and `SmartFloat` data types (we compare performance in subsection 3.2.4). These results provide an idea on the order of magnitude of round-off error estimates, as well as the scalability of our approach. Note that none of these benchmarks is known to be particularly unstable for floating-point errors, so we cannot observe some especially bad behavior. We can see that our `AffineFloats` and `SmartFloats` give consistently better, that is, more accurate bounds on the absolute round-off errors. The numbers for the SOR and LU benchmark also suggest that our library generally scales better in terms of accuracy on longer computations than interval arithmetic. The type of computation has a strong influence on how fast the over-approximation of error bounds grows; linear computations can naturally be handled better with affine arithmetic than nonlinear ones. Furthermore, the numbers show that in most cases, when considering a single computation, `AffineFloat` computes tighter bounds by leveraging its different and more accurate semantics. On three benchmarks `SmartFloat` actually performs

benchmark	details	`IntervalFloat`	`AffineFloat`	`SmartFloat`
Nbody	1s, dt=0.01	4.20e-14	**1.94e-14**	1.95e-14
	1s, dt=0.015625	2.83e-14	**1.28e-14**	1.29e-14
	5s, dt=0.01	3.12e-12	**7.55e-13**	7.57e-13
	5s, dt=0.015625	2.06e-12	**4.89e-13**	4.98e-13
Spectral norm	2 iter.	1.69e-14	**1.33e-15**	1.73e-15
	5 iter.	6.24e-14	**3.33e-15**	1.07e-14
	10 iter.	1.43e-13	**9.77e-15**	1.12e-14
	15 iter.	2.24e-13	**1.38e-14**	1.27e-13
	20 iter.	3.03e-13	3.75e-14	**1.82e-14**
SOR	5 iter.	6.17e-13	**1.30e-13**	1.88e-13
	10 iter.	4.01e-11	**2.77e-12**	3.80e-12
	15 iter.	2.70e-9	**5.68e-11**	7.84e-11
	20 iter.	1.84e-7	**1.19e-9**	1.39e-9
LU with pivoting	dim 5	3.89e-14	**6.40e-15**	2.17e-14
	dim 10	2.14e-11	**2.34e-12**	NaN
	dim 15	4.07e-9	**1.53e-10**	NaN
	dim 5	6.53e-9	**6.01e-11**	NaN
	dim 10	1.51e-8	**1.44e-10**	NaN
LU w/o pivoting	dim 15	4.36e-1	**8.24e-5**	NaN
FFT	dim 256	4.11e-11	9.66e-12	**7.21e-12**
	dim 512	1.33e-10	3.77e-11	**2.44e-11**

Table 3.1 – Comparison of absolute errors computed by `IntervalFloat`, `AffineFloat` and `SmartFloat` with compacting threshold 42 (which seemed like a good compromise between accuracy and performance).

better, which is likely due to the slightly larger over-approximation `AffineFloat` may have to make for nonlinear computations.

3.2.2 Accuracy of SmartFloat

We evaluate the accuracy and usefulness of our SmartFloat data type on a number of selected benchmarks.

- The **triangle** example from Figure 3.1 is known to be exhibit cancellation, which happens when two quantities close in value are subtracted and thus many of the correct digits get subtracted away in the process. This happens in particular for flat triangles, where we observe a loss of accuracy. Figure 3.2 shows an improved version due to [93]. The code first sorts the sides of the triangle by their lengths and refactors the final formula such that computations are performed in an order that minimizes accuracy loss.

- Another famous example is the formula for computing the roots of a **quadratic** equation. The formulation usually taught in schools produces less accurate results (orders of magnitude), when one root is much smaller. Figure 3.3 shows both the classical formulation as well as an improved version from [71].

- The following equation computes the frequency change due to the **Doppler** effect:

$$z = \frac{dv}{du} = \frac{-(331.4 + 0.6T)v}{(331.4 + 0.6T + u)^2}$$

  This benchmark is challenging because it is nonlinear, includes a division, and contains a strong correlation between the nominator and denominator.

- **B-spline** basic functions are commonly used in image processing [91]

$$\begin{aligned} B_0(u) &= (1-u)^3/6 \\ B_1(u) &= (3u^3 - 6u^2 + 4)/6 \\ B_2(u) &= (-3u^3 + 3u^2 + 3u + 1)/6 \\ B_3(u) &= u^3/6 \end{aligned}$$

  With $u \in [0,1]$, this benchmark also features strong correlations.

Table 3.2 compares the round-off errors computed by our tool against those computed by Fluctuat [72]. At the time when we developed our library (2011), Fluctuat was not available for comparison, so the numbers we report here were obtained later with the spring 2014 version of Fluctuat, which may include later improvements. We note that overall the differences are, in general, rather small, but it would be interesting to determine which design choices together are optimal in the future. Both tools are clearly able to determine that Kahan's triangle area and the smarter quadratic root computation produce more accurate results than their textbook versions.

```
def triangleKahan(a: SmartFloat, b: SmartFloat, c: SmartFloat): SmartFloat = {
  if(b < a) {
    val t = a
    if (c < b) { a = c; c = t }
    else
     if (c < a) { a = b; b = c; c = t }
     else { a = b; b = t }

  } else if (c < b) {
    val t = c; c = b;
    if (c < a) { b = a; a = t }
    else { b = t }
  }
  sqrt((a+(b+c)) * (c-(a-b)) * (c+(a-b)) * (a+(b-c))) / 4.0
}
```

Figure 3.2 – Computing the area of a triangle with `SmartFloats` and Kahan's improved version of the formula.

```
val a: SmartFloat = ...
val b: SmartFloat = ...
val c: SmartFloat = ...

val discr = b*b - a * c * 4.0

//classical way
var r1 = (-b - sqrt(discr))/(a * 2.0)
var r2 = (-b + sqrt(discr))/(a * 2.0)

//smarter way
val (rk1: SmartFloat, rk2: SmartFloat) =
if(b*b - a*c > 10.0) {
  if(b > 0.0)
    ( (-b - sqrt(discr))/(a * 2.0), c * 2.0 /(-b - sqrt(discr)) )
  else if(b < 0.0)
    ( c * 2.0 /(-b + sqrt(discr)), (-b + sqrt(discr))/(a * 2.0) )
  else
    ( (-b - sqrt(discr))/(a * 2.0), (-b + sqrt(discr))/(a * 2.0) )
}
else
  ( (-b - sqrt(discr))/(a * 2.0), (-b + sqrt(discr))/(a * 2.0) )
```

Figure 3.3 – Quadratic formula in two versions

benchmark	inputs	SmartFloat	Fluctuat
triangle textbook	a = 9, b = c = [4.71, 4.89]	**5.46272459e-14**	5.49398942e-14
	a = 9, b = c =[4.61, 4.79]	**7.09227167e-14**	7.19504679e-14
	a = 9, b = c =[4.501, 4.581]	1.16065397e-12	**8.38212416e-13**
triangle Kahan	a = 9, b = c =[4.71, 4.89]	1.62947801e-14	**1.36572207e-14**
	a = 9, b = c =[4.61, 4.79]	1.93681987e-14	**1.52819080e-14**
	a = 9, b = c =[4.501, 4.581]	1.94849999e-13	**1.29513866e-13**
quadratic classic			
r1		1.17647194e-14	**1.02354065e-14**
r2	a = [2.499, 3.499],	**1.57388750e-15**	1.60866674e-15
quadratic smarter	b = [55.5001, 56.5001],		
rk1	c = [0.50074, 1.50074]	1.17647194e-14	**1.02354065e-14**
rk2		4.24657821e-17	**1.11700058e-17**
doppler	u = [-100, 100], v = [20, 20 000], T = [-30, 50]	3.69565423e-12	**3.90280004e-13**
bsplines			
b0		**1.57281595e-16**	1.61907525e-16
b1	u = [0, 1]	**8.93960831e-16**	9.25185854e-16
b2		8.37293198e-16	**8.32667269e-16**
b3		**9.13621031e-17**	1.06396374e-16

Table 3.2 – Comparison of absolute errors computed by SmartFloat and Fluctuat

3.2.3 Spring Simulation

So far, we have only measured round-off errors, however, errors can come from other sources as well. For instance, during the integration of an ordinary differential equation, the numerical algorithm accumulates truncation errors, i.e. errors due to the discretization of the integration algorithm. Unlike roundoff errors, truncation errors highly depend on the particular integration method used so that no simple unique formula for quantifying these errors exists. One example that illustrates these errors is the simulation of an (undamped and unforced) spring in Figure 3.4. For simplicity, we use Euler's method. Although this method is known to be too inaccurate for many applications, it provides a good application showcase for our library. Note the method `addError` in line 15. In this example, we compute a coarse approximation of the method error by computing the maximum error over the whole execution. What happens

```
def springSimulation(h: SmartFloat) = {
  val k: SmartFloat = 1.0
  val m: SmartFloat = 1.0
  val xmax: SmartFloat = 5.0
  var x: SmartFloat = xmax //curr. horiz. position
  var vx: SmartFloat = 0.0 //curr. velocity
  var t: SmartFloat = 0.0 //curr. 'time'

  var truncError = k*m*xmax * (h*h)/2.0

  // while(certainly(t < 1.0)) {
  while(t < 1.0) {
    val x_next = x + h * vx
    val vx_next = vx - h * k/m * x
    x = x_next.addError(truncError)
    vx = vx_next
    t = t + h
  }
}
```

Figure 3.4 – Simulation of a spring with Euler's method

behind the scenes is that our library adds an additional error to the affine form representing x, i.e. it adds a new noise term in addition to the errors already computed. At the end of the simulation, we obtain the following results. The numbers in parentheses are the maximum absolute round-off errors committed. We have rounded the output outwards for readability reasons.

step size	t	x
h = 0.1	[1.099,1.101] (8.55e-16)	[2.174, 2.651] (7.4158e-15)
h = 0.125	[1.0,1.0] (0.00e+0)	[2.618, 3.177] (4.04e-15)
h = 0.01	[0.999, 1.001] (5.57e-14)	[2.699, 2.706] (6.52e-13)

When running this simulation with step sizes 0.1 and 0.01, time t cannot be computed accurately, whereas using $h = 0.125$, which is representable in binary, the result is exact. Our library detects this, and will print the warning `comparison failed!` when evaluating the while condition in the last iteration. This warning is (correctly) not printed for step size $h = 0.125$. Now consider x. We can see that choosing smaller step sizes, the enclosure of the result becomes smaller and thus more accurate, as expected. But note also, that the use of a smaller step size also increases the overall round-off errors. This is also to be expected, because we have to execute more computations.

Note that this accurate analysis of round-off errors is only possible with the separation of round-off errors from other uncertainties. Our `SmartFloat` type can thus be used in a more

benchmark	double	IntervalFloat	AffineFloat	SmartFloat
Nbody 1000 steps	0.2	28.5	1446.1	7079.8
Spectral norm, 20 iter.	1.5	35.4	168.4	410.0
SOR 20 iter.	1.1	52.5	4404.7	9396.1
LU w/ pivoting, size 15	2.7	3.2	12.4	40.0
FFT size 512	4.5	7.2	247.3	1088.5

Table 3.3 – Comparison of running times of our different data types and plain double precision. Compacting threshold was set as 42.

general framework that guarantees soundness with respect to a floating-point implementation but that also includes other sources of errors.

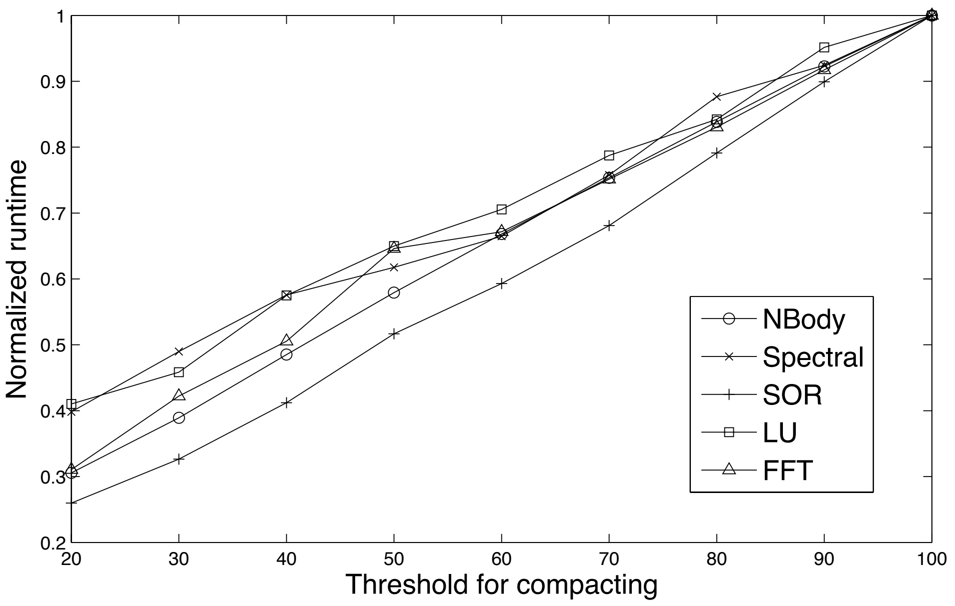
An alternative to printing the `comparison failed!` warning is to let the `SmartFloat` throw an exception when the comparison on line 12 cannot be decided with certainty. We can catch this exception for example with our **`certainly`** construct, as in line 11. In this case, the code will perform one iteration less. Had we chosen **`possibly`**, the comparison would be more permissive and the number of iterations would have been the same as when running this application with plain `Doubles`.

3.2.4 Performance

Our technique aims to provide much more information than ordinary floating point execution while using the same concrete execution. We therefore do not expect the performance to be comparable to that of an individual double precision computation on dedicated floating-point hardware. Nonetheless, our technique is effective for unit testing and for exploring smaller program fragments one at a time. `AffineFloat` and `SmartFloat` use the floating-point implementation of affine arithmetic as we found that the program fragments we consider in our benchmarks are already long enough to slow down a rational implementation unacceptably. The runtimes of `AffineFloat` and `SmartFloat` are summarized in Table 3.3. `SmartFloats` essentially use two affine forms, which accounts for the larger runtimes. We believe that given that the computations are long running, the runtimes remain acceptable. Similarly, total memory consumption in these benchmarks was not measurably large. We used the Scalameter framework [132] for benchmarking.

3.2.5 Compacting

We have also run experiments to determine the effect the maximal number of noise terms has both on the runtime and on the accuracy. Figure 3.5 shows the effect of different noise symbol count thresholds on the runtime and accuracy. The normalized running times of `AffineFloat` on our benchmarks show a more or less linear dependence and the table with error bounds



benchmark	20	40	60	80	100
Nbody 1000	6.27e-11	4.72e-11	3.64e-11	3.30e-11	3.32e-11
Spectral 20	3.66e-14	6.46e-14	2.66e-14	6.02e-14	2.33e-14
SOR 20	3.19e-8	1.96e-9	4.25e-10	1.65e-10	8.92e-11
LU 15	4.77e-10	2.63e-10	1.36e-10	5.35e-11	4.22e-11
FFT 512	6.79e-11	4.41e-11	3.41e-11	2.37e-11	1.80e-11

Figure 3.5 – Effect of the maximum number of noise symbols on the running time (graph) and error bounds (table).

gives an idea about the loss of correlation and accuracy compacting can cause. Based on this trade-off, the user of our library can set a different threshold for different applications and even dynamically.

3.2.6 Position of the Moon Case Study

Our final example in this chapter is an astronomical program that computes the moon's position as a function of the date. We obtained two implementations of the algorithm presented in [115] from the Geneva observatory, one in Java and one in Python. They observed that two otherwise identical implementations sometimes returned different results, and were thus worried about the accuracy. The implemented algorithm is about 120 lines of code, and performs 946 linear operations, 1390 multiplications, 74 divisions or square roots and calls 209 trigonometric functions. The algorithm returns two coordinates α and δ, for example for the input date 2012-2-10: (173.69903942141244, -2.6074387947918**73**) in Python, and (173.69903942141244, -2.6074387947918**56**) in Java, note the difference in the last two digits. This difference becomes even more visible, when these "raw" coordinates are converted into equatorial coordinates.

We used our `AffineFloat` data type on this code and our results are summarized in the following table.

	α	δ
Python	173.69903942141244	-2.607438794791873
`AffineFloat`	173.69903942141244	-2.607438794791856
	± 6.2172e-9	± 2.5341e-9
Python	11h 34m 47.76946113898774	-2d 36m 26.77966125074235
`AffineFloat`	11h 34m **47.7694**6113898774	-2h 36m **26.7796**61250681812
	± 1.4921e-6	± 9.1226e-6

The top part of the table shows the raw coordinates computed, and the bottom part the converted equatorial coordinates. Our `AffineFloat` running on the JVM returns the same result as plain Java code, so we only list one. The raw coordinate data only differs in the last two digits for the δ value, but this error grows when converted to coordinates. This is also reflected in the growth of the error reported by our `AffineFloat` (from order e-9 to order e-6). Finally, with our data type and analysis, we are able to certify that the digits in bold are definitely correct (which is sufficient for the particular application that used this code).

Conclusion

We have presented a runtime instantiation of our round-off error computation and demonstrated its usability and utility on a number of examples. We have extensively analyzed our implementation in terms of accuracy of results, scalability and performance and shown that even a 'straightforward' affine arithmetic implementation can provide decent results. In particular, we have thoroughly compared the affine arithmetic performance against interval arithmetic and can confirm that the theoretical advantages do indeed translate to the implementation, at least for mostly linear computations.

4 Synthesizing Accurate Fixed-Point Expressions

Many algorithms in control and signal processing are implemented on embedded devices which do not have a hardware floating-point unit due to their higher energy consumption and cost. Software emulation is slow and may thus not be an option either. Hence, these algorithms are often implemented in fixed-point arithmetic. Much research exists into bit width allocation [101, 110, 94], that is, the determination of the number of integer and fractional bits. The order of computation has been largely disregarded and often left as generated for example from MATLAB [113] or as written by hand. However, even with an optimal bit width allocation, the final round-off can depend on the exact order of computation due to the lack of associativity of fixed-point arithmetic. As we will show, this difference can be quite large.

One of the desired properties of control systems is asymptotic stability, i.e. the property that the state of the controlled plant will asymptotically converge to a given point. It has been shown [9] that when implementation errors are present, this convergence can only be shown for a set around the point, called the region of practical stability, whose size depends on the magnitude of the round-off errors. It is thus of practical importance to reduce errors as much as possible. One way to reduce errors is to increase the bit length available for fixed-point arithmetic. This approach, however, can be costly, especially when a gradual increase is not possible and a whole word needs to be added (e.g. increase 16 bits to 32 bits). An approach that can avoid this cost, such as ours, is thus preferable.

In this chapter we present our technique and tool called Xfp for synthesizing accurate fixed-point expressions from their real-valued specification. Our technique is based on a heuristic search with genetic programming [129] and uses our affine arithmetic based error computation as the fitness function. Since we are searching among mathematically equivalent expressions, we have adapted genetic programming to this setting by defining our own mutation and crossover operators. We focus mostly on linear controllers, which are very common.

The work presented in this chapter is based on [48]. The source code is available at `github.com/malyzajko/xfp`.

4.1 Exploiting Non-Associativity

Consider a controller for a batch reactor processor [134]. The computation of one state of the controller is given by the following expression:

$$\begin{aligned}(-0.0078) * \mathtt{st}_1 + 0.9052 * \mathtt{st}_2\ + (-0.0181) * \mathtt{st}_3 +\\ (-0.0392) * \mathtt{st}_4\ + (-0.0003) * \mathtt{y}_1 + 0.0020 * \mathtt{y}_2\end{aligned} \tag{4.1}$$

where $\mathtt{st_i}$ is an internal state of the controller and $\mathtt{y_i}$ are the inputs. Similar expressions compute the other states and outputs of the controller. Suppose we want to implement this controller in fixed-point arithmetic. If we assume an input range of $[-10, 10]$ for all input variables and a uniform bit length of 16, each input variable gets assigned a fixed-point format with 4 integer and 11 fractional bits. The constant -0.0078 gets 22 fractional bits: $0.0078 < 2^{16-1-22} = 2^{-7} = 0.0078125$. If we multiply $\mathtt{st}_1$ now by -0.0078, the result will have 33 bits, which we fit into 16 bits by performing a right shift. Continuing in this fashion and following the order of arithmetic operations in (4.1) we obtain the following fixed-point arithmetic program:

```
val tmp0 = ((-32716l * st1) >> 18)
val tmp1 = ((29662l * st2) >> 15)
val tmp2 = ((tmp0 + (tmp1 << 4)) >> 4)
val tmp3 = ((-18979l * st3) >> 16)
val tmp4 = (((tmp2 << 4) + tmp3) >> 4)
val tmp5 = ((-20552l * st4) >> 15)
val tmp6 = (((tmp4 << 4) + tmp5) >> 4)
val tmp7 = ((-20133l * y1) >> 22)
val tmp8 = (((tmp6 << 4) + tmp7) >> 4)
val tmp9 = ((16777l * y2) >> 19)
val tmp10 = (((tmp8 << 4) + tmp9) >> 4)
return tmp10
```

A fixed-point arithmetic implementation of the controller can have a large round-off error. For instance, with a bit length of 16, the input values can already have a round-off error as large as 0.00049. We can apply our affine arithmetic-based analysis presented in chapter 2 to compute an upper bound on the overall error of 3.9e-3. Since this technique computes worst-case errors, it is not necessarily clear whether such large errors can indeed occur. Running a simulation with a double floating-point and a fixed-point implementation side by side on a large number of random inputs, we obtain a *lower bound* on the error of 3.06e-3. So indeed, the error can be quite big.

One way to reduce the error is to increase the bit length. If we add one bit to each variable, we get a simulated maximum error of 1.51e-3, which is an improvement by about 50%. However, increasing bit widths may require implementing circuits with larger areas or using larger data

expression	simulated error
original	3.06e-3
worst rewrite	3.11e-3
additional bit	1.52e-3
best rewrite	1.39e-3
best found by GP	1.39e-3

Table 4.1 – Summary of absolute errors for different implementations

types, and may not be feasible. A different possibility is to use a different order of evaluation for the expression. As fixed-point arithmetic operations are not associative, two different evaluation orders for the same implementation can have significantly different round-off errors. If we reorder Equation (4.1) as

$$\begin{split}((0.9052 * \mathtt{st}_2) + (((\mathtt{st}_3 * -0.0181) + (-0.0078 * \mathtt{st}_1)) + \\ (((-0.0392 * \mathtt{st}_4) + (-0.0003 * \mathtt{y}_1)) + (0.002 * \mathtt{y}_2))))\end{split} \tag{4.2}$$

and again implement it using 16-bit fixed-point arithmetic, we find by simulation an (under-approximated) error bound of 1.39e-03. This is an even larger improvement of 55% than by increasing the bit length, without requiring any extra resources. Figure 4.1 summarizes the worst-case error bounds for the different formulations of the expression. By exhaustively enumerating all possible rewrites, we see that the maximum error bounds can vary between approximately 1.39e-3 and 3.11e-3. That is, even for a relatively short example, the worst error bound can be over a factor of 2 larger than the best possible one. The main reason for these differences comes the fact that changing the order of execution may change the intermediate ranges of variables, which in turn influence the fixed-point formats and round-off errors, as well as the propagation in the case of nonlinear operations.

Since exhaustive enumeration becomes infeasible very quickly, we want to search the space of possible implementations of an arithmetic expression in more a effective fashion, with the goal to find one that has the minimum fixed-point implementation error bound. We have chosen *genetic programming* (GP) as our search procedure. On our example, GP can find the optimal expression without an exhaustive enumeration and can do this at analysis time. This does not cost any additional resources, thus we get the additional accuracy “for free”.

4.2 Search with Genetic Programming

We now describe our algorithm to find a fixed-point implementation of a mathematical expression which reduces the overall round-off error over a straight-forward implementation. That is, given a real-valued expression t we aim to find an expression t' that is mathematically equivalent to t and whose implementation in fixed-point arithmetic $\tilde{t}'$ minimizes, among all equivalent expressions, the worst-case absolute error over all inputs in given ranges I:

$$\min_{\text{equivalent } t'} \max_{x \in I} \left| t(x) - \tilde{t}'(x) \right|.$$

Our tool accepts as input an arithmetic expression generated by the following grammar:

$$t ::= c \mid x \mid t_1 + t_2 \mid t_1 - t_2 \mid t_1 * t_2 \mid t_1 / t_2$$

where c and x are rational constants and variables, respectively. The output is an alternative expression with a different order of computation whose error according to our static analysis is smaller than from the original one. If our tool cannot find a better rewrite, it returns the original expression. We also generate the corresponding integer-valued fixed-point implementation.

4.2.1 Genetic Programming

Genetic programming is part of the broad class of genetic algorithms. These heuristic search algorithms, inspired by natural evolution, are parameterized by a fitness function to evaluate a candidate solution as well as operators called mutation and crossover to generate new candidate solutions from previous ones. The general algorithm maintains a population of candidate solutions, and evolves the current population in phases called *generations*. An evolution step picks two candidates from the current generation, and computes new candidate solutions by applying the mutation and crossover operations. The quality of the new solution is evaluated with the fitness function, and the new candidate is added to the next generation if the fitness function assigns a sufficiently high score to it. In genetic algorithms, candidate solutions are usually represented by strings, for which mutation and crossover can be defined easily. Selection of candidates from the population for mutation and crossover can be performed for example by tournament selection where a fixed number of candidates is chosen at random and the one with the highest fitness is selected as the final candidate. This allows, with some probability, “less fit” candidates to participate in the evolution process. Together with mutation and crossover, this allows for a search which can explore different parts of the search space quickly and has a smaller possibility to be caught in a local optimum. Genetic programming [129] is a variant of a genetic algorithm that performs the search over computer programs, i.e. their abstract syntax trees (AST), instead of strings, with mutation and crossover adapted accordingly.

4.2.2 Instantiating Genetic Programming

Algorithm 4.1 gives an overview of our search procedure, whose steps we explain in the following paragraphs. The input is a real-valued expression and ranges for its input variables. Our tool initializes the initial population with 30 copies of this expression. Our mutation and crossover operators need to generate expressions that are mathematically equivalent to the initial expression. Since standard genetic programming poses no constraints on generated

```
Input: expression, input ranges
initialize population of 30 expressions

repeat for 30 generations
  generate 30 new expressions:
    select 2 expressions with tournament selection
    do equivalence-preserving crossover
    do equivalence-preserving mutation
    evaluate fitness (roundoff error)

Output: best expression found during entire run
```

Figure 4.1 – Best expression search procedure

expressions and can also generate invalid ones, we define special purpose mutation and crossover operators ourselves. Our fitness function quantifies the worst-case numerical error between the fixed-point implementation and the mathematical expression.

Mutation Standard mutation in genetic programming first selects a random node in the expression AST and mutates it by either replacing it entirely with a different AST node, or by modifying its information. In our instantiation, the mutation operator applies one of the applicable rewrite rules from Figure 4.2 to the randomly selected node. The rules capture the usual commutativity, distributivity and associativity of real arithmetic and preserve mathematical equivalence. Some of these rules do not have an effect on the numerical accuracy by themselves, but are necessary to generate other rewrites of an expression. To keep the operations simple, we rewrite subtractions ($x - y \rightarrow x + (-y)$) and divisions ($x/y \rightarrow x * (1/y)$) before the GP run.

Crossover Usual genetic programming crossover picks a random subtree each from two expression candidates and exchanges them, creating two new expressions ASTs. As our constraint is to produce mathematically equivalent expressions, we need to find two subtrees that are mathematically equivalent. The following algorithm is incomplete in the sense that it may fail to find such a pair. It is, however, efficient as it does not perform a general equivalence check, and we have observed its effectiveness in practice.

(x + y) + z = x + (y + z)	(-x) * y = - (x * y)	(x * y) + (x * z) = x * (y + z)
x + y = y + x	x * (-y) = - (x * y)	(x * z) + (y * z) = (x + y) * z
(-x) + (-y) = -(x + y)	1/x * 1/y = 1/(xy)	(x * y) + (z * x) = x * (y + z)
(x * y) * z = x * (y * z)	- (1/x) = 1/(-x)	(y * x) + (x * z) = (y + z) * x
x * y = y *x		

Table 4.2 – Rewrite rules

Given two trees t_1 and t_2 as candidates, the crossover algorithm picks a random node in t_1, which is the root of the subtree we call s_1. To ensure that crossover produces a mathematically equivalent expression, we have to find a subtree s_2 in t_2 that is mathematically equivalent to s_1 in an efficient way. Instead of implementing a general decision procedure, each subtree is annotated at initialization time (line 2 in Figure 4.1) with a label that is the string representation of the expression at that subtree. During mutation, labels are preserved in the new generation as much as possible. For example, suppose we have the node $(x+y)+z$, with label `(x + y) + z`. We can apply mutation rule 1 to obtain $x+(y+z)$ but the label will remain `(x + y) + z`. Note that some of the mutation rules break equivalences (e.g. the rules in the third column), hence not all labels can be preserved. In that case we add a new label. During crossover, we then only need to check for identical labels. If labels match, it means that the subtrees come from the same initial subtree and hence are mathematically equivalent and we can exchange them in a crossover operation.

Parameters Our genetic programming pipeline has several parameters that can influence the results: the population size (we choose 30 as we observed no benefit beyond this value), the number of best individuals passed on to the next generation unchanged (elitism) (0, 2 or 6), the number of individuals considered during tournament selection (2, 4 or 6), and the probability of crossover (0.0, 0.5, 0.75 or 1.0). The most successful setting we found is with a tournament selection among 4 and an elitism of 2 while performing crossover every time, i.e. with probability of 1.0. Note, however, that even in the case of other settings, the improvements are still significant (on the order of 50%).

4.2.3 Fitness Evaluation

We use the round-off error analysis from chapter 2 as our fitness function, now applied to fixed-point arithmetic. This analysis is similar to [60], but we treat constants like normal variables and we do not discard higher order terms to ensure that the error bounds we compute are sound with respect to real arithmetic.

If we are interested in proving that the round-off errors stay within certain bounds, the computed absolute bounds on these errors need to be as tight as possible. The main requirement on the analysis in our current problem is slightly different, however. While tight bounds on errors are an advantage, what we need to know is the *relative precision* of our analysis tool. That is, we need to know whether the analysis tool is able to distinguish a better implementation from a less accurate one. This is not necessarily given, because the analysis assumes worst-case errors at each computation step and these will, in general, not be attained every time. Hence, our tool may report a difference between two expressions that is due to the over-approximations of the algorithm and not a true difference in accuracy.

Thus, before using our analysis tool in a GP framework, we evaluated this property experimentally. We generated a number of random rewrites for an expression, for which we then obtain

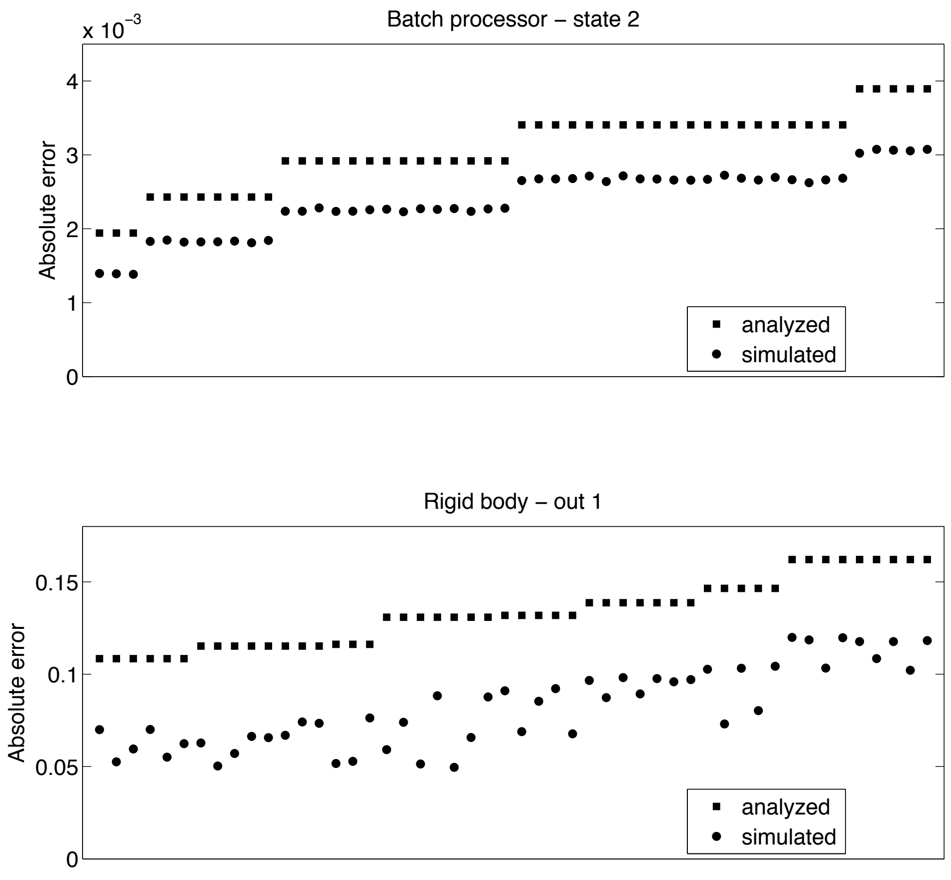


Figure 4.2 – Comparison of analyzed upper bound and simulated lower bound on maximum errors for a linear and a nonlinear benchmark

the (approximate) actual errors by simulation. We present here the results for one linear and one nonlinear benchmark (batch controller, state 2 and rigid body, out 1 respectively). For 100 different random expression formulations, the ratio between the analyzed upper bound on the error and the simulated lower bound on the error has a mean of 1.29387 and a variance of 0.00082 for the batch controller, state 2 benchmark and a mean of 1.66697 and a variance of 0.08315 for the rigid body, out 1 benchmark. Figure 4.2 shows a direct comparison between the analyzed and simulated errors. In the linear case, the computed bounds on the errors are proportional to the actual errors, thus indicating a good relative precision. In the nonlinear case the correspondence is not as accurate, however we expect it to be still sufficient for our purpose. The "more nonlinear" a computation becomes, the less accurate we expect affine arithmetic to be in discriminating expressions by accuracy.

Effectively, our search algorithm with this fitness function returns the expression for which *it is able to prove* the smallest error bound. Even if this does not happen to be the truly optimal one, for many applications or certification purposes it is already useful to obtain (an otherwise equivalent) expression but with a smaller *proven* error bound.

4.2.4 Why Genetic Programming?

It is in general not evident from an expression whether its order of computation is good with respect to accuracy and exhaustively enumerating all possible formulations of expressions becomes impossible very quickly. For only linear expressions the number of possible orders of adding n terms modulo commutativity, which does not affect accuracy, is $(2n-3)!!$[1]. For our example from Section 4.1 with 6 terms there are already 945 expressions. For our largest benchmark with 15 terms there are too many possibilities to enumerate.

We thus need a suitable search strategy to find a good formulation of an expression among all the possibilities. We want to minimize intermediate ranges of variables, but because the inputs for the expressions can, in general, be positive and negative, optimizing one subcomputation may lead to a very large intermediate sum in a different part of the expression. An algorithm that tries to find the optimal solution in a systematic way (e.g. dynamic programming) is thus unlikely to succeed. Our problem also does not have a notion of a gradient, so gradient descent approaches are not readily applicable. Furthermore, the problem cannot be easily formulated in terms of inputs and outputs or constraints, so constraint solving approaches are not applicable either. Genetic programming does not rely on any of these features, and its formulation as a search over program ASTs fits our problem well.

4.3 Experimental Results

We have implemented our genetic programming algorithm inside the Java-based Evolutionary Computation Research System (ECJ) [105] using the round-off error analysis presented in chapter 2. For this application, we used the rational implementation, as the expressions tend to be evaluated are relatively short.

Benchmarks come from the controller domain: a bicycle model [12], a DC motor position control [2], a pitch angle control [2], an inverted pendulum [2], and a batch reactor process [75]. The controllers for these systems are taken from [108], which attempted to minimize the size of the region of practical stability by choosing a controller whose fixed-point implementation has the best possible bound on the error among all controllers that stabilize the plant. To show the scalability of our tool we choose the example of a locomotive pulling a train car where the connection between the locomotive and the car is modeled by a spring in parallel with a damper [114]. By increasing the number of cars, we can increase the dimension of the system. We also consider a nonlinear controller for a rigid body [10] and the nonlinear B-splines functions [101]. Though most of our benchmarks are from the controller domain, nothing in our approach is actually domain specific.

[1]The number of full binary trees with n leaves is C_{n-1}, where C_n are the Catalan numbers. We can label each of the trees in n! ways. Taking into account commutativity gives: $\frac{C_{n-1} \cdot n!}{2^{n-1}}$.

Each benchmark consists of one expression and ranges for its input parameters. We wish to minimize the error on the one output value it computes over all possible inputs. Some of the benchmarks compute internal states of a controller (denoted with e.g. “state 1”). Since each state is computed with a different expression, we treat them here as separate benchmarks. For all benchmarks we consider a fixed bit length, signed fixed-point format, and truncation as the rounding mode.

4.3.1 End-to-End Results

Tables 4.3 and 4.4 list the maximum absolute errors (as computed by our analysis tool) for the best expressions found by GP for all our benchmarks. The results were obtained with the best parameter settings we found: elitism: 2, tournament selection: 4, with and without crossover (seed used in ECJ: 4357). The best found expression is the same for different bit lengths, so the computed results are applicable in several different hardware settings. The benchmarks are ordered approximately by complexity with the smaller linear benchmarks in Table 4.3 and the nonlinear benchmarks in Table 4.4. From the third column we can see that we get substantial improvements in accuracy of up to 70%.

Benchmark	err	$\frac{\text{orig.- no-cross}}{\text{orig.}}$	$\frac{\text{orig.- best}}{\text{orig.}}$	g
bicycle (out1)	2.66e-3	0.00	0.00	-
bicycle (state1)	2.53e-4	0.19	0.19	1
bicycle (state2)	1.82e-4	0.00	0.00	-
dc motor (out1)	1.06e-4	0.00	0.00	-
dc motor (state1)	2.77e-4	0.00	0.00	-
dc motor (state2)	3.75e-4	0.25	0.25	4
dc motor (state3)	1.27e-4	0.00	0.00	-
pendulum (out1)	8.09e-8	0.03	0.03	5
pendulum (state1)	5.13e-9	0.17	0.17	1
pendulum (state2)	6.11e-9	0.38	0.38	16
pendulum (state3)	5.14e-9	0.00	0.00	-
pendulum (state4)	4.97e-9	0.27	0.27	7
pitch angle (out1)	1.33e-7	0.18	0.18	4
pitch angle (state1)	4.26e-9	0.30	0.30	2
pitch angle (state2)	2.79e-9	0.00	0.00	-
pitch angle (state3)	3.81e-9	0.20	0.20	2

Table 4.3 – Maximum absolute errors for the best expression found by GP (first part). g denotes the generation in which the solution is found.

Benchmark	err	$\frac{\text{orig.- no-cross}}{\text{orig.}}$	$\frac{\text{orig.- best}}{\text{orig.}}$	g
batch reactor (out1)	5.15e-4	0.00	0.00	-
batch reactor (out2)	1.28e-3	0.12	0.12	2
batch reactor (state1)	3.46e-4	0.15	0.15	1
batch reactor (state2)	2.77e-4	0.00	0.00	-
batch reactor (state3)	3.55e-4	0.26	0.26	2
batch reactor (state4)	4.11e-4	0.23	0.23	7
traincar 1 (out)	1.11e-4	0.09	0.09	2
traincar 1 (state 1)	1.98e-6	0.03	0.03	6
traincar 1 (state 2)	3.57e-7	0.25	0.25	16
traincar 1 (state 3)	2.79e-7	0.24	0.24	7
traincar 2 (out)	7.40e-5	0.09	0.09	19
traincar 2 (state 3)	1.23e-7	0.49	0.59	21
traincar 3 (out)	1.26e-3	0.13	0.13	7
traincar 3 (state 6)	1.32e-7	0.48	0.58	21
traincar 3 (state 7)	1.31e-7	0.43	0.53	17
traincar 4 (out)	9.34e-3	0.26	0.29	27
traincar 4 (state 1)	7.29e-8	0.73	0.73	19
traincar 4 (state 2)	7.34e-8	0.67	0.73	25
traincar 4 (state 3)	1.01e-7	0.66	0.60	14
traincar 4 (state 4)	6.96e-8	0.64	0.70	26
traincar 4 (state 5)	1.42e-7	0.61	0.68	26
traincar 4 (state 6)	1.67e-7	0.59	0.59	16
traincar 4 (state 7)	1.67e-7	0.56	0.56	13
traincar 4 (state 8)	1.38e-7	0.60	0.60	19
traincar 4 (state 9)	1.67e-7	0.47	0.47	7
bspline 1	2.29e-4	0.36	0.36	6
bspline 2	1.66e-4	0.52	0.52	4
rigid-body (out1)	1.08e-1	0.33	0.33	5
rigid-body (out2)	9.92e-1	0.20	0.20	15

Table 4.4 – Maximum absolute errors for the best expression found by GP (second part). g denotes the generation in which the solution is found.

Benchmark	Original	Best (% of original)	Worst (% of original)	Added bit (% of original)
batch processor (out 1)	2.89e-3	0.91	1.37	0.51
batch processor (out 2)	7.20e-3	0.81	1.13	0.49
batch processor (state 1)	2.66e-3	0.52	1.02	0.50
batch processor (state 2)	3.06e-3	0.45	1.03	0.50
batch processor (state 3)	2.66e-3	0.50	1.16	0.49
batch processor (state 4)	2.24e-3	0.61	1.40	0.51
traincar 1 (out)	5.29e-5	0.78	1.02	0.60
traincar 1 (state 1)	1.02e-6	0.97	1.01	0.50
traincar 1 (state 2)	2.66e-7	0.71	1.02	0.52
traincar 1 (state 3)	2.04e-7	0.74	1.14	0.48

Table 4.5 – Best and worst absolute errors among exhaustively enumerated rewritings, determined by simulation

It can be shown [48] that our rewriting procedure, when implemented inside a tool that performs a search for a best controller [108], can provide significant improvements in the size of the region of practical stability. This result also shows that our technique is orthogonal to other optimization strategies and can thus be used in addition to provide further improvements.

4.3.2 Exhaustive Rewriting

For our smaller benchmarks (linear with up to 6 terms) we also generate all possible rewrites up to commutativity and determine tight bounds on the maximum errors by simulation. For this, we first automatically generate a fixed-point implementation, which we then evaluate on a number of random inputs. We use double precision floating-point code as reference, thus obtaining a lower bound on the error. Using a large enough number of random inputs (10^7), we obtain reasonably tight error bounds. Table 4.5 shows the simulation results for the original formulation of the expression and the best and worst among all the possible rewrites. From the 10 benchmarks, 6 have an error in the original formulation that is about as bad as it gets. But we can also see that the best possible rewrite can improve the accuracy substantially. On the other hand, the expression may also be such that no matter how we rewrite it, the accuracy does not vary much, as is the case with the traincar 1, state 1 benchmark. This case, however, seems to be rather rare.

Anther possibility of improving the accuracy is to increase the bit lengths (possibly selectively). We do so in a minimal way by allowing one more bit for each intermediate variable, i.e., we increase the bit length by one and evaluate the accuracy with simulation. Note that in many cases, such a gradual increase may not be possible and one would have to add a whole word, e.g. go from 16 to 32 bits, with the associated increase in hardware cost. Compile-time transformations, on the other hand, come “for free” and, as Table 4.5 shows, can have an effect on the same order of magnitude as the addition of a bit. In the case of longer benchmarks, the effect of rewriting can be even larger (see Table 4.4), while the added bit always gains only about 50% of accuracy as compared to the original error.

4.3.3 Genetic Programming

Tables 4.3 and 4.4 shows the results obtained for the most successful setting we have found. It also shows the results with crossover turned off. The comparison of these two columns suggests that crossover is helpful. We therefore expect that randomized local search techniques are not as effective as genetic programming, but they still produce useful reductions in the errors.

Optimality For the smaller benchmarks, GP always finds the same expressions with respect to the error bounds. Exhaustive enumeration has confirmed that the found expressions are indeed the optimal ones. For larger benchmarks, we do get improvements and we know of no technique to obtain better results.

Performance The runtimes of our GP algorithm depend in general mostly on the number of generations considered and the population size. Crossover only has a small effect on the overall runtime, but we have found that it provides the best results in the setting given above. In Table 4.6 we report the running times of our benchmarks with the default setting of 30 generations with a population size of 30 and the best GP settings we found.

Examples	Runtime (s)
batch (out1)	1.964
batch (state2)	2.735
traincar 1 (state 3)	6.358
traincar 2 (state 5)	9.794
traincar 3 (state 7)	15.388
traincar 4 (state 9)	17.228
rigid-body (out1)	1.394
rigid-body (out2)	2.698
bspline 1	2.234

Table 4.6 – Average runtimes of GP on selected benchmarks in seconds. Experiments were performed on a 3.5GHz Linux desktop with 16GB RAM.

Example	GP	Random	(Random-GP)/Random
traincar 4 (out)	0.00934	0.0103	0.09
traincar 4 (1)	7.29e-8	2.07e-7	0.65
traincar 4 (2)	7.34e-8	2.08e-7	0.65
traincar 4 (3)	1.01e-7	1.90e-7	0.47
traincar 4 (4)	6.96e-8	1.74e-7	0.60
traincar 4 (5)	1.42e-7	3.21e-7	0.56
traincar 4 (6)	1.67e-7	2.86e-7	0.42
traincar 4 (7)	1.67e-7	2.57e-7	0.35
traincar 4 (8)	1.38e-7	2.28e-7	0.39
traincar 4 (9)	1.67e-7	1.97e-7	0.15

Table 4.7 – Comparison of maximal errors between best expressions from GP and random search

Efficiency Improvement over Random Search For the expressions of the traincar 4 controller (15 terms) we also compare the results from the GP algorithm against a random search. This experiment is performed by generating 900 random and unique rewrites of the original expression, and comparing the best seen expressions against each other. Since we run the GP algorithm for 30 generations with a population of 30, we can see at most 900 unique expressions. As the results in Table 4.7 confirm, the GP based search is more effective than a random one. The third column shows the relative difference between the errors. Thus, many times the GP found expression is by over 50% more accurate than a randomly found one. That GP does not perform a random search can also be seen on the evolution of the population in Figure 4.3 for one benchmark (traincar 4, state 1). The plot shows the best, worst and the average errors of the expressions in each generation. The convergence to a low-error expression is clear.

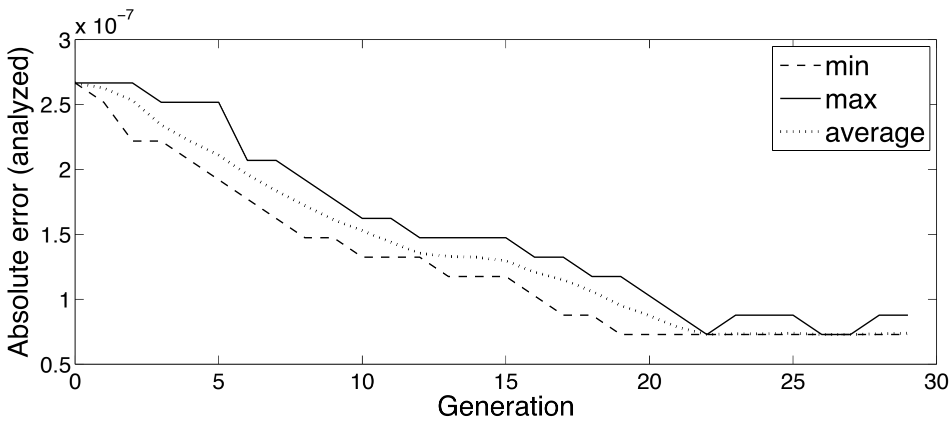


Figure 4.3 – Evolution of errors across generations for the traincar 4 - state 1 example

4.4 Conclusion

We have presented a genetic programming based search for synthesizing fixed-point arithmetic expressions with improved errors bounds. We have systematically investigated the effect of rewriting on the round-off errors, the suitability of an affine-arithmetic based static analysis technique as the fitness function, as well as the effects of mutation and crossover. A light-weight static analysis such as our chosen one is particularly practical for a search like genetic programming, because it is fast and can thus be applied repeatedly. In addition, it has the advantage that returned results come with sound error bounds.

5 Writing and Compiling Real Programs

So far, we have presented an analysis technique for estimating round-off errors in finite-precision code, as well as an application in a runtime library for floating-points and inside a genetic programming algorithm to find more accurate fixed-point implementations. Both of these applications have in common that they require the programmer to choose the data type up front and then to remember verifying the accuracy of results, essentially leaving accuracy considerations an afterthought.

In this chapter, we propose an alternative way of programming numerical code by introducing a specification language with a real semantics together with a high-level compilation and verification algorithm. Our language includes numerical errors explicitly and uses these specifications to determine a suitable finite-precision data type automatically.

This chapter is based on the paper [47]. The presented framework has been implemented inside the tool called *Rosa*, and is part of the Leon verification framework [21] and relies on the Z3 solver [53] in the back-end. Its source code is available at `github.com/malyzajko/rosa`.

5.1 Programming in Reals

Many current approaches for verifying numerical programs start with the finite-precision implementation and then try to verify the absence of (runtime) errors. Not only are such verification results specific to a given representation of numbers, but the absence of run-time errors does not guarantee that program behavior matches the desired specification expressed using real numbers. Fundamentally, the source code semantics is mostly expressed in terms of low-level data types such as floating-points. This is problematic not only for developers but also for compiler optimizations, because, e.g. code transformations based on associativity such as those presented in chapter 4 are unsound with respect to such source code semantics.

We propose the following alternative: source code programs should be expressed in terms of mathematical real numbers and determining which data type is suitable for the low-level implementation should be done, or at least supported by automated tools. In our system, the programmer writes a program using a `Real` data type, including pre- and postconditions which specify explicitly the uncertainties on inputs as well as the desired accuracy of the result.

It is then up to our trustworthy compiler to check, taking into account all uncertainties and their propagation, that the desired accuracy can be soundly realized in a finite-precision implementation. If so, the compiler chooses and emits one such implementation, selecting from a range of (software or hardware) floating-point or fixed-point arithmetic representations. Viewing source code as operating on real numbers has many advantages:

Separation of concerns A program written with reals separates the mathematical algorithm from its low-level implementation details. Programmers can reason about correctness using real arithmetic instead of finite-precision arithmetic. We achieve separation of the design of algorithms (which may still be approximate for other reasons) from their realization using finite- precision computations.

Verification over reals We can verify the ideal meaning of programs using techniques developed to reason over real numbers, which are more scalable and better understood than techniques that directly deal with finite-precision arithmetic. If we then also establish bounds on how far the results computed with finite precision are from the real-valued ones, we can obtain overall correctness guarantees.

Compiler optimizations The compiler for reals is free to do optimizations as long as they preserve the accuracy requirements. This allows the compiler to apply, for example, associativity of arithmetic (for example as described in chapter 4), or even select different approximation schemes for transcendental functions.

Specification of ideal behavior A real specification language provides us with the ideal behavior of the program. We can use this as the baseline against which to compare approximate implementations. This enables us to quantify the deviation of implementation outputs from ideal ones, instead of merely proving e.g. range bounds of floating-point variables which is used in simpler analyses that check for runtime error freedom.

5.1.1 A Real Specification Language

In our framework each program to be compiled consists of one top-level object with methods written in a functional subset of the Scala programming language [125]. Choosing only the functional subset makes, on the one hand, the verification task easier, but on the other hand, we believe that this subset is also closer to mathematical semantics.

All methods are functions over the `Real` data type and the user annotates them with pre- and postconditions that explicitly talk about uncertainties. `Real` represents ideal real numbers without any uncertainty. We allow arithmetic expressions over `Real`s with the standard arithmetic operators $\{+, -, *, /, \sqrt{}\}$, and together with conditionals and function calls they form the body of methods. Our tool also supports immutable variable declarations such as `val x = ...`.

```
def triangle(a: Real, b: Real, c: Real): Real = {
  require(1.0 < a && a < 9.0 && 1.0 < b && b < 9.0 && 1.0 < c && c < 9.0 &&
        a + b > c + 0.1 && a + c > b + 0.1 && b + c > a + 0.1)

  val s = (a + b + c)/2.0
  sqrt(s * (s - a) * (s - b) * (s - c))

} ensuring (res => 0.29 <= res && res <= 35.1 && res +/- 2.7e-11 &&
              0.29 <= ~res && ~res <= 35.1)
```

Figure 5.1 – Example function written in our specification language

Loops can be expressed with recursive functions. This language allows the user to define the *ideal* (or baseline) computation over real numbers. Note that this specification language is not executable.

For example, recall the function computing the area of a triangle from section 3.1. The code in Figure 5.1 shows a possible specification of this function using our proposed real-valued semantics. The precondition allows the user to provide a specification of the environment consisting of lower and upper bounds for all method parameters and an upper bound on the uncertainty or noise. Range bounds are expressed with regular comparison operators, e.g. `x < 9.0`. Uncertainty is expressed with a predicate such as `x +/- 2.7e-11`, which denotes that the variable x is only known up to an uncertainty of $2.7e-11$. If the programmer does not specify the noise explicitly, round-off errors are assumed by default. The postcondition can specify constraints on the output, and in particular the range and the maximum accepted uncertainty.

In addition to this specification, the user may also provide additional constraints on the inputs, such as `a + b > c + 0.1` in the example. These constraints go beyond simple interval constraints and are important since they allow us to analyze more general functions. Indeed, in the triangle area example we do not have to restrict the input ranges such that a negative radicand does not occur.

Writing x in our language references the ideal real-valued variable. The user may also want to express that the finite-precision computation does not violate certain bounds. For this case, our language provides the notation `~x` which references the *actual* value of x when executed in finite precision, which we denote by $\tilde{x}$.

5.2 Compiling Reals to Finite Precision

Now that we have a real-valued specification, a "verifying" compiler needs to instantiate the program with a concrete finite-precision data type that satisfies the specification. In the case where the choice falls on a fixed-point data type, the tool also needs to generate the fixed-point code (over integers), for the case of floating-points code generation essentially reduces to replacing the `Real` type with the chosen floating-point type. The main task thus consists in verifying that a given data type satisfies the specification.

To this end, we generate verification conditions which are based on the high-level idea to explicitly model the *ideal* program without external uncertainties and round-offs, the *actual* program, which is executed in finite precision with possibly noisy inputs, and the relationship between the two. This is possible exactly because we have the real-valued semantics available through our real-valued specification. Furthermore, we can encode reasoning about finite-precision round-off errors into reasoning about real numbers. This allows us on one hand, to leverage the superior scalability of real-valued solvers over floating-point or bit vector ones. On the hand, a combination of theories is problematic, and hence encoding everything into reals lets us consider the relationship between both the real-valued and the finite-precision computation.

5.2.1 Verification Conditions for Loop-Free Programs

In this and the following chapter we consider loop-free programs. While the presented techniques also work, in principle, for recursive functions, in practice a successful verification requires the specification to be inductive, including its error part. However, except in very special cases, round-off errors grow with every iteration, hence inductive specification written in the language we present here (with constant error bounds) are unlikely to exist. We present an extension that can handle programs with loops in chapter 7 and the remaining concepts presented in this chapter carry over to those programs as well.

Our approach constructs the following verification condition for each method with a precondition P and a postcondition Q:

$$\forall x, \mathsf{res}, y.\ P(x) \land body(x, y, \mathsf{res}) \rightarrow Q(x, \mathsf{res}) \tag{5.1}$$

where x, res, y, possibly vector valued, denote the input, output and local variables respectively.

Table 5.1 summarizes how verification constraints are generated from our specification language for floating-point arithmetic. Each variable x in the specification corresponds to two real-valued variables $x, \tilde{x}$, the ideal one in the absence of uncertainties and round-off errors and the actual one, computed by the compiled program. The ideal and actual variables are related only through the error bounds in the pre- and postconditions, which allows for the ideal and actual executions to take different paths through the program. In the method body we have to take into account round-off errors from arithmetic operations and the propagation of existing errors. Note that the resulting verification conditions are parametric in the machine epsilon.

Fixed-point arithmetic can, in principle, also be encoded in a similar fashion, albeit with more complex constraints as it does not admit a dynamic format allocation as floating-point

`a <= x && x <= b`	$x \in [a, b]$
`x +/- k`	$\tilde{x} = x + err_x \wedge err_x \in [-k, k]$
`~x`	$\tilde{x}$
`x ◇ y` (ideal part)	$(x \diamond y)$
`x ◇ y` (actual part)	$(\tilde{x} \diamond \tilde{y})(1 + \delta_1)$
`sqrt(x)` (ideal part)	$sqrt(x)$
`sqrt(x)` (actual part)	$sqrt(\tilde{x})(1 + \delta_2)$
`val` `z = x`	$z = x \wedge \tilde{z} = \tilde{x}$
`if` `(c(x)) e1(x)`	$((c(x) \wedge e_1(x)) \vee (\neg c(x) \wedge e_2(x))) \wedge$
`else` `e2(x)`	$((\tilde{c}(\tilde{x}) \wedge \tilde{e}_1(\tilde{x})) \vee (\neg \tilde{c}(\tilde{x}) \wedge \tilde{e}_2(\tilde{x})))$
`g(x)`	$g(x) \wedge \tilde{g}(\tilde{x})$

$\diamond \in \{+, -, *, /\}$

$-\epsilon_m \leq \delta_i \wedge \delta_i \leq \epsilon_m$, all δ are fresh

$\tilde{c}$ and $\tilde{e}$ denote functions with roundoff errors at each step

Table 5.1 – Specification language semantics

arithmetic. Current approaches for encoding fixed-point arithmetic rely on bit vectors, whose use in our case is problematic as we would need to mix bit vector and nonlinear real theories in the SMT solver. As we will explain shortly (subsection 5.2.3), such a direct encoding does not scale well for floating-points already, hence we switch to using approximations immediately.

Our system currently supports the operations $\{+, -, *, /, \sqrt{}\}$, which are the supported operations of the nonlinear back-end solver Z3. The techniques in the following chapters can be extended to elementary functions, provided the solver can handle them. Another possible avenue would be to encode them via Taylor expansions [109].

5.2.2 Specification Generation

In order to give feedback to developers and to facilitate automatic modular analysis, Rosa also provides automatic specification generation. By this we mean that the programmer still needs to provide the environment specification in form of preconditions, but our tool automatically computes an accurate postcondition. Formally, we can rewrite the constraint (5.1) as

$$\forall x, res.\ (\exists y.\ P(x) \wedge \mathsf{body}(x, y, res)) \rightarrow Q(x, res)$$

where Q is now unknown. We obtain the most accurate postcondition Q by applying quantifier elimination (QE) to $P(x) \wedge \mathsf{body}(x, y, res)$ and eliminate y. The theory of arithmetic over reals admits QE so it is theoretically possible to use this approach.
We do not currently use a full QE procedure for specification generation, as it is expensive and it

is not clear whether the returned expressions would be of a suitable format. Instead, we use our approximation approach which computes ranges and maximum errors in a forward fashion and computes an (over) approximation of a postcondition of the form $res \in [a, b] \wedge res \pm u$. When proving a postcondition, our tool automatically generates these specifications and provides them as feedback to the user.

5.2.3 Difficulty of Simple Encoding into SMT solvers

For small functions we can already prove interesting properties by using the exact encoding of the problem just described and discharging the verification constraints with Z3. Consider the following code a programmer may write to implement the third B-spline basic function from subsection 3.2.2.

```
def bspline3(x: Real): Real = {
  require(0 ≤ x && x ≤ 1 && x +/- 1e-13)

  -x*x*x / 6.0

} ensuring (res => -0.17 ≤ res && res ≤ 0.05 && res +/- 1e-11)
```

Functions and the corresponding verification conditions of this complexity are already within the possibilities of the nonlinear solver within Z3 [92]. For more complex functions however, Z3 does not (yet) provide an answer in a reasonable time, or returns unknown. Whether alternative techniques in SMT solvers can help in such cases remains to be seen [27].

5.2.4 Compilation Algorithm

Since a direct attempt to solve verification conditions using an off-the-shelf solver alone is not satisfactory, we use approximations to tackle the verification. Before we address the individual challenges, we present here an overview of our compilation algorithm.

Given a specification or program over reals and possible target data type(s), Rosa generates code over floating-point or fixed-point numbers that satisfy the given pre- and postconditions (and thus meet the target accuracy). Figure 5.2 presents a high-level view of our compilation algorithm. Rosa first analyses the entire specification and generates one verification condition for each postcondition to be proven. To obtain a modular algorithm, Rosa also generates verification conditions that check that at each function call the precondition of the called function is satisfied. The methods are then sorted by occurring function calls. This allows us to re-use already computed postconditions of function calls in a modular analysis. If the user specifies one target data type, the remaining part of the compilation process is performed with respect to this data type's precision. If not or in the case the user specified several possible types, Rosa will perform a binary search over the possible types to find the least in the sorted

```
Input: spec: specification over Reals, prec: candidate precisions
for fnc ← spec.fncs
 fnc.vcs = generateVCs(fnc)

spec.fncs.sortBy((f1, f2) => f1 ⊆ f2.fncCalls)

while prec ≠ ∅ and notProven(spec.fncs)
 precision = prec.nextPrecise
 for fnc ← spec.fncs
  for vc ← fnc.vcs
   while vc.hasNextApproximation ∧ notProven(vc)
    approx = getNextApproximation(vc, precision)
    vc.status = checkWithZ3(approx)
 generateSpec(fnc)
generateCode(spec)

Output: floating-point or fixed-point code
```

Figure 5.2 – Compilation algorithm

list that satisfies all specifications. The user can provide the list of possible data types manually and sort them by her individual preference. Currently, the analysis is performed separately for each data type, which is not a big issue performance wise due to the relatively small number of alternatives. We did identify certain shared computations between iterations which can be exploited in the future for more efficient compilation. In order for the compilation process to succeed, the specification has to be met with respect to some given finite-precision arithmetic. We envision that in the future the compilation task will also include automatic accuracy-improving code optimizations.

Rosa can currently generate Scala code over

- fixed-point arithmetic with a 16 or 32 bit width, or
- floating-point arithmetic in single (32 bit), double (64 bit), double-double (128 bit) and quad-double (256 bit) precision.

Our approach is parametric in the bit width of the representations and can thus be used on different platforms, from embedded controllers without floating-point units (where fixed-point implementations are needed), to platforms that expose high-precision floating-point arithmetic in their instruction set architecture. Using the QuadDouble library [82, 16] we can also emit code that uses precision twice or four times that of the ubiquitous `Double` data type. When multiple representations are available (as specified by the user), the compiler can select, e.g., the smallest representation needed to deliver the desired number of trusted significant digits. We currently do not support mixing of data types in one program, but this is possible avenue for future work.

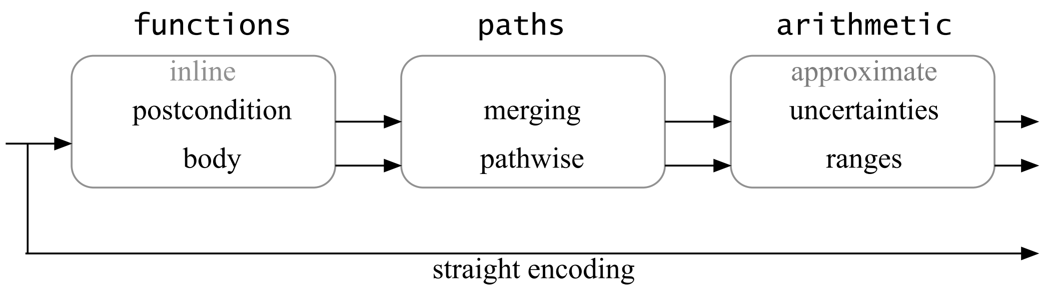


Figure 5.3 – Approximation pipeline

In case the verification part of compilation fails, our tool prints a failure report with the best postconditions Rosa was able to compute. The user can then use the generated specifications to gain insight why and where his or her program does not satisfy the requirements. This computed postcondition is also printed with the successfully generated program.

While we have implemented our tool to accept specifications in a domain specific language embedded in Scala and generate code in Scala, all our techniques apply equally to all programming languages and hardware that support the floating-point features we assume (subsection 2.1.1).

5.2.5 Verification with Approximations

We now describe the approximations Rosa uses to soundly compile more interesting programs. We use an approximation pipeline, which exploits different combinations of approximations of nonlinear arithmetic, function calls, and paths due to conditionals. Each can be approximated at different levels and we have observed in our experiments, that one size does not fit all and a combination of different approximations is most successful in proving the verification conditions we encountered in our examples. The computed approximations can also be directly used to generate the postconditions.

For each verification condition we thus construct approximations until we are able to prove one, or until we run out of approximations where we report the verification as failed. We can thus view verification as a stream of approximations to be proven. We illustrate the pipeline that computes the different approximations in Figure 5.3. The first approximation (indicated by the long arrow) is to use Z3 alone on the entire constraint constructed by the rules in Table 5.1. This is indeed an approximation, as all function calls are treated as uninterpreted functions in this case. This approximation only works for floating-point precisions in very simple cases without function calls. Then, taking all possible combinations of subcomponents in our pipeline we obtain the other approximations, which are filtered according to the presence of function calls or conditional branches.

Function calls If the verification constraint contains function calls, Rosa will attempt to first inline postconditions. We support inlining of both user-provided postconditions and postconditions computed by our own specification generation procedure. If this is not accurate enough (and the function is not recursive), we inline the entire function body.

Postcondition inlining is implemented by replacing the function call with a fresh variable and constraining it with the postcondition. Thus, if verification succeeds with inlining the postcondition, we avoid having to consider each path of the inlined function separately and can perform modular verification avoiding a potential path explosion problem. Obviously, such modular verification is only feasible when postconditions are accurate enough. Section 6.2 presents an improvement over the error bound computation from chapter 2.

Paths In the case of several paths through the program, we have the option to consider each path separately or to merge results at each join in the control flow graph. This introduces a trade-off between efficiency and accuracy, since on one hand, considering each path separately leads to an exponential number of paths to consider. On the other hand, merging at each join looses correlation information between variables which may be necessary to prove certain properties. Our approximation pipeline chooses merging first, before resorting to a path-by-path verification in case of failure. We believe that other techniques for exploring the path space could also be integrated into our tool [97, 34]. Another possible improvement are heuristics that select a different order of approximations depending on particular characteristics of the verification condition.

Another aspect of conditional branches are *discontinuity* errors. These capture the difference in the result when the finite-precision computation takes a different path through the program than the ideal real-valued one. Whether this divergence happens and how large it becomes depends on the errors of the variables in the branch condition. Section 7.1 presents two possible approaches we have developed to tackle this challenging problem. Since this computation may be costly, Rosa includes a `@robust` annotation, which signals that the discontinuity errors have been treated already and need not be re-checked.

Arithmetic The arithmetic part of the verification constraints generated by Table 5.1 can be essentially divided into the ideal part and the actual part, which includes round-off errors at each computation step. The ideal part determines whether the ideal range constraints in the postcondition are satisfied and the actual part determines whether the uncertainty part of the postcondition is satisfied.

Based on this, our tool first constructs an approximation which leaves the ideal part unchanged, but replaces the actual part of the constraint by the computed uncertainty bound. This effectively removes a large number of variables and is many times a sufficient simplification for Z3 to succeed in verifying the entire constraint. If Z3 is still not able to prove the constraint, our tool constructs the next approximation by also replacing the ideal part, this time with a constraint of the result's range which has been computed by our approximation procedure previously. Note that this second approximation may not have enough information to prove a more complex postcondition, as correlation information is lost. We note that the computation of ranges and errors is the same for both approximations and thus trying both does not affect efficiency significantly. In our experiments, Z3 is able to prove the ideal, real-valued part in most cases, so this second approximation is rarely used.

Our approximation for arithmetic satisfies the following:

$$\begin{aligned}
&([a,b], err) = \text{evalWithError}(P, expr) \Rightarrow \\
&\forall x, \tilde{x}, \text{res}, \tilde{\text{res}}. P(x,\tilde{x}) \land \text{res} = \text{expr}(x) \land \tilde{\text{res}} = \tilde{\text{expr}}(\tilde{x}) \\
&\qquad\qquad \rightarrow a \leq \text{res} \land \text{res} \leq b \land |\text{res} - \tilde{\text{res}}| < \text{err}
\end{aligned}$$

That is, the procedure `evalWithError` computes a sound over-approximation of the range of an expression and of the uncertainty on the output. In particular, the affine arithmetic-based procedure from chapter 2 for ranges of inputs satisfies the constraint and can be used here. It turns out, however, that for many even short computations this procedure is very inaccurate and we present significant improvements in chapter 6 using the power of a nonlinear SMT solver both for the estimation of the ranges and the errors.

Counter-examples Our procedure is sound because our constraints over-approximate the actual errors. Furthermore, even in the full constraint as generated from Table 5.1, round-off errors are over-approximated since we assume the worst-case error bound at each step. While this ensures soundness, it also introduces incompleteness, as we may fail to validate a specification because our over-approximation is too large. This implies that counterexamples reported by Z3 are in general only valid if they disprove the ideal real-valued part of the verification constraint. Our tool checks whether this is the case by constructing a constraint with only the real-valued part, and reports the counterexamples, if such are returned from Z3.

5.3 Experience

Before we explain our employed techniques in detail, we illustrate the capabilities of our system on a number of examples.

5.3.1 Polynomial Approximations of Sine

We illustrate the verification algorithm on the example in Figure 5.4, using double floating-point precision as the target. The functions `sineTaylor` and `sineOrder3` are verified first since they do not contain function calls. Verification with the straight encoding fails. Next, Rosa computes the round-off errors on the output and Z3 succeeds to prove the resulting

```
def comparisonValid(x: Real): Real = {
  require(-2.0 < x && x < 2.0)
  val z1 = sineTaylor(x)
  val z2 = sineOrder3(x)
  z1 - z2
} ensuring(res => res <= 0.1 && res +/- 5e-14)

def comparisonInvalid(x: Real): Real = {
  require(-2.0 < x && x < 2.0)
  val z1 = sineTaylor(x)
  val z2 = sineOrder3(x)
  z1 - z2
} ensuring(res => res <= 0.01 && res +/- 5e-14)

def sineTaylor(x: Real): Real = {
  require(-2.0 < x && x < 2.0)
  x - (x*x*x)/6.0 + (x*x*x*x*x)/120.0 - (x*x*x*x*x*x*x)/5040.0
} ensuring(res => -1.0 < res && res < 1.0 && res +/- 1e-14)

def sineOrder3(x: Real): Real = {
  require(-2.0 < x && x < 2.0)
  0.954929658551372 * x - 0.12900613773279798*(x*x*x)
} ensuring(res => -1.0 < res && res < 1.0 && res +/- 1e-14)
```

Figure 5.4 – Different polynomial approximations of sine

constraint with the ideal part untouched. From this approximation Rosa directly computes a new, more accurate postcondition, in particular it can narrow the resulting errors to 1.48e-15 and 1.23e-15 respectively. Next, our tool considers the `comparisonValid` function. Inlining only the postcondition is not enough in this case, but computing the error approximation with the functions inlined succeeds in verifying the postcondition. Note that our tool does not approximate the real-valued portion of the constraint, i.e. Z3 is used directly to verify the constraint $z1 - z2 \leq 0.1$. This illustrates our separation of the real reasoning from the finite-precision implementation: with our separation we can use a real arithmetic solver to deal with algorithmic reasoning and verify with our error computation that the results are still valid within the error bounds in the implementation. Finally, the tool verifies that the preconditions of the function calls are satisfied by using Z3 alone. Verification of the function `comparisonInvalid` fails with all approximations. Our tool is able to determine that the ideal real-valued constraint alone ($z1 - z2 \leq 0.01$) is not valid, reports a counterexample ($x = 1.875$) and returns `invalid` as the verification result.

5.3.2 Trade-off between Accuracy and Efficiency

Which data type is suitable for a given program depends on the parameter ranges, the code itself but also the accuracy required by the application using the code. For example, take the functions in Figure 5.5. Depending on which accuracy on the output the user needs, our tool will select different data types. For the requirement `res +/- 1e-12`, as specified, `Double` is a suitable choice for `doppler` and `turbine` [151], however for the `jetEngine` example [10] this is not sufficient, and thus `DoubleDouble` would be selected by our tool. The user can influence which data types are preferred by supplying a list to our tool which is ordered by her preference.

Figure 5.6 illustrates the trade-off between the accuracy achieved by different data types against the runtime of the compiled code generated by our tool. We used the Caliper [1] framework for benchmarking the running times.

```
def doppler(u: Real, v: Real, T: Real): Real = {
  require(-100 < u && u < 100 && 20 < v && v < 20000 && -30 < T && T < 50)

  val t1 = 331.4 + 0.6 * T
  (- (t1) *v) / ((t1 + u)*(t1 + u))

} ensuring(res => res +/- 1e-12)

def jetEngine(x1: Real, x2: Real): Real = {
  require(-5 < x1 && x1 < 5 && -20 < x2 && x2 < 5)

  val t = (3*x1*x1 + 2*x2 - x1)
  x1 + ((2*x1*(t/(x1*x1 + 1))*
  (t/(x1*x1 + 1) - 3) + x1*x1*(4*(t/(x1*x1 + 1))-6))*
  (x1*x1 + 1) + 3*x1*x1*(t/(x1*x1 + 1)) + x1*x1*x1 + x1 +
  3*((3*x1*x1 + 2*x2 -x1)/(x1*x1 + 1)))

} ensuring(res => res +/- 1e-12)

def turbine(v: Real, w: Real, r: Real): (Real, Real, Real) = {
  require(-4.5 < v && v < -0.3 && 0.4 < w && w < 0.9 && 3.8 < r && r < 7.8)

  val t1 = 3 + 2/(r*r) - 0.125*(3-2*v)*(w*w*r*r)/(1-v) - 4.5
  val t2 = 6*v - 0.5 * v * (w*w*r*r) / (1-v) - 2.5
  val t3 = 3 - 2/(r*r) - 0.125 * (1+2*v) * (w*w*r*r) / (1-v) - 0.5
  (t1, t2, t3)

} ensuring (_ match {
  case (r1, r2, r3) => r1 +/- 1e-12 && r2 +/- 1e-12 && r3 +/- 1e-12
})
```

Figure 5.5 – Benchmark functions from physics and control systems

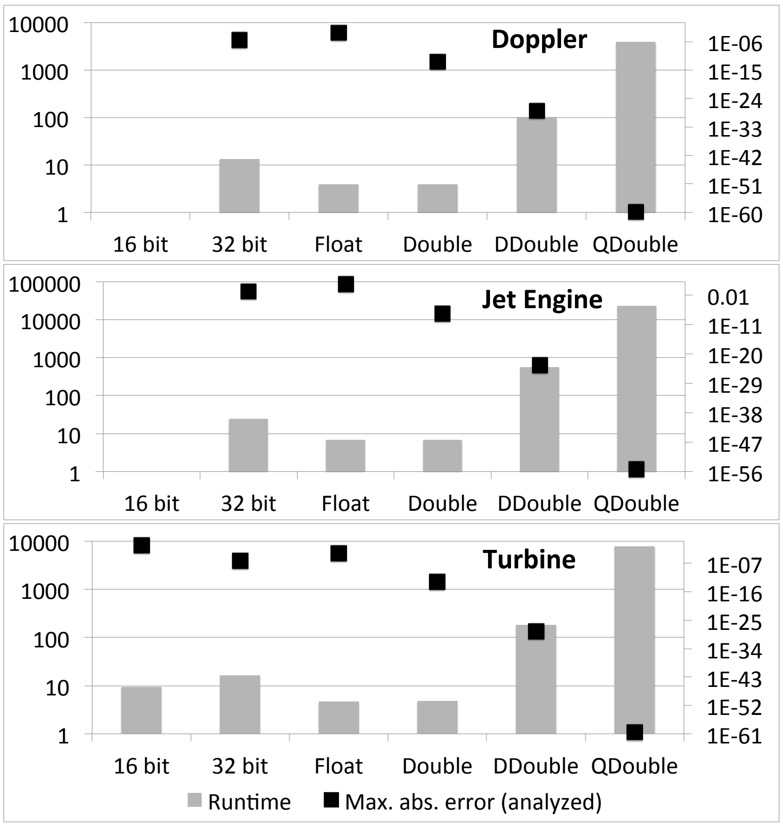


Figure 5.6 – Runtimes of benchmarks compiled for different precisions in nanoseconds (left) vs. abs. error bounds computed for that precision by our tool (right).

For our benchmarks with their limited input ranges, 32 bit fixed-point implementations provide better accuracy than single floating-point precision because single precision has to accommodate a larger dynamic range which reduces the number of bits available for the mantissa. That said, fixed-point implementations run slower, at least on the JVM, than the more accurate double floating-point arithmetic with its dedicated hardware support. However, the choice for fixed-point rather than floating-point may be also due to this hardware being unavailable. Our tool can thus support a wide variety of applications with different requirements. We also note that across the three (not specially selected) benchmarks, the results are very consistent and we expect similar behavior for other applications as well.

6 Computing Precise Range and Error Bounds

This chapter presents our techniques that tackle one of the main challenges when analyzing and verifying numerical code: nonlinear arithmetic. Since a direct translation of finite precision arithmetic into a real-valued constraint is not tractable at this point (see chapter 5), we need to approximate both the ranges of variables and the round-off errors committed.

Being able to accurately compute a sound range of an expression is important by itself for many applications, as many functions and algorithms work only within limited domains. Range computations are also often building blocks for many (sound) algorithms. For example, round-off errors depend directly on variable ranges, hence we need to be able to compute them accurately. We will also use range computations extensively for the accurate computation of propagation and discontinuity errors in this and the next chapter. We observe that affine arithmetic alone often cannot provide satisfying results when faced with nonlinear arithmetic. We show in the first part of this chapter how we combine nonlinear SMT solving with affine arithmetic to obtain the needed tight bounds. This work is based on [47].

In the second part of this chapter, we propose a new error computation based on separating the propagation of *existing* errors from the roundoff or truncation errors committed *during* the computation. This separation allows us to distinguish the implementation aspects from the mathematical properties of the underlying function and handle them individually with appropriate techniques. In particular, this separation allows us to directly use the properties of the real-valued functions underlying the finite-precision implementation. Such a separation of errors applies fairly generally: it enabled us to i) improve computed error bounds on straight-line nonlinear code (this chapter), ii) characterize errors in loops as functions of the number of iterations (chapter 7), and iii) scale discontinuity error computations to multivariate functions (chapter 7).

Using the separation of errors, our new propagation procedure improves the computed error bounds compared to affine arithmetic. Our approach considers an entire arithmetic expression at a time, computing an approximation of the *global* effect of the function on the input errors, in contrast to the local linear approximation of affine arithmetic. Our procedure is backed by our new range computation, which allows us to capture nonlinear correlations precisely. Moreover, it provides information about the sensitivity of the function with respect to the errors, which is useful for understanding the behavior of the function and for modular inter-procedural analysis. This work is currently under submission.

6.1 Range Bounds

The goal is to compute an accurate bound on the real-valued range of a nonlinear expression, given ranges for its inputs. So far, we have performed range computation with interval or affine arithmetic, and argued that most of the time the latter gives more accurate results since it takes into account linear correlations. However, both arithmetics perform over- approximations which become significant for nonlinear expressions, especially when the input intervals are not small (radius > 1). In the case of nonlinear expressions the results computed by affine arithmetic can actually become worse than for interval arithmetic. For example, $x * y$ with $x = [-5,3], y = [-3,1]$ gives $[-13,15]$ in affine arithmetic and $[-9,15]$ in interval arithmetic. This over-approximation leads to less accurate bounds and round-off error estimates, but becomes critical when division or square root operations are involved. For example, when we try to analyze the jet engine controller [10] from Figure 5.5, interval and affine arithmetic provide the bound $[-\infty,\infty]$ on the output, since neither can bound the denominators in the divisions away from zero. They fail to do this because of the nonlinearity and the high correlation of the (only) two inputs `x1` and `x2`.

Furthermore, we have seen in the triangle area example (Figure 5.1) that additional constraints beyond simple range bounds can be very useful. The precondition bounds the inputs as $a,b,c \in [1,9]$, but the formula is useful only for valid triangles, i.e. when every two sides together are longer than the third. If not, we will get an error at the latest when we try to take the square root of a negative number. Interval and affine arithmetic again fail to analyze this example, since they cannot, without the valid-triangle constraints, bound the radicand away from the negative range.

6.1.1 Range Computation

The input to our algorithm is a nonlinear real-valued expression expr and a precondition P on its inputs, which specifies, among possibly other constraints, ranges on all input variables $x \in \mathbb{R}^n$. The output is an interval $[a,b]$ which satisfies the following:

$$
\begin{aligned}
&[a,b] = \text{getRange}(P, \mathsf{expr}) \Rightarrow \\
&\qquad \forall x, \mathsf{res}. P(x) \wedge \mathsf{res} = \mathsf{expr}(x) \rightarrow (a \leq \mathsf{res} \wedge \mathsf{res} \leq b)
\end{aligned}
$$

Observation: A nonlinear theorem prover such as the one that comes with Z3 can decide with fairly good accuracy whether a given bound is sound or not. That is, we can check with a prover whether for an expression e the range $[a, b]$ is a sound interval enclosure. This observation is the basis of our range computation.

The algorithm for computing the lower bound of a range is given in Figure 6.1. The computation for the upper bound is symmetric. For each range to be computed, our tool first computes an initial sound estimate of the range with interval arithmetic. It then performs an initial quick check to test whether the computed first approximation bounds are already tight. If not, it uses the first approximation as the starting point and then narrows down the lower and upper bounds using a binary search. At each step of the binary search our tool uses Z3 to confirm or reject the newly proposed bound.

The search stops when either Z3 fails, i.e. returns unknown for a query or cannot answer within a given timeout, the difference between subsequent bounds is smaller than an accuracy threshold, or the maximum number of iterations is reached. This stopping criterion can be set dynamically.

Additional constraints In addition to the input ranges, the precondition may also contain further constraints on the variables, such as the valid-triangle constraints from Figure 5.1. In interval-based approaches we can only consider input intervals that satisfy this constraint for

```
def getRange(expr, precondition, precision, maxIterations):
  z3.assertConstraint(precondition)
  [aInit, bInit] = evalInterval(expr, precondition.ranges);

  //lower bound
  if z3.checkSat(expr < a + precision) == UNSAT
    a = aInit
    b = bInit
    numIterations = 0
    while (b-a) < precision ∧ numIterations < maxIterations
      mid = a + (b - a) / 2
      numIterations++
      z3.checkSat(expr < mid) match
        case SAT ⇒ b = mid
        case UNSAT ⇒ a = mid
        case Unknown ⇒ break
    aNew = a
  else
    aNew = aInit

  //upper bound symmetrically
  bNew = ...
  return: [aNew, bNew]
```

Figure 6.1 – Algorithm for computing the range of an expression

all values, and thus have to check several (and possibly many) cases. In our approach, since we are using Z3 to check the soundness of bounds, we can assert the additional constraints up-front and then all subsequent checks are performed with respect to all additional and initial constraints. This allows us to avoid interval subdivisions due to inaccuracies or problem specific constraints such as those in the triangle example. This becomes especially valuable in the presence of multiple variables, where we may otherwise need an exponential number of subdivisions.

A very similar approach has been developed independently in [94] in the context of bit-width allocation for fixed-point arithmetic, using HySAT [63] as their back-end solver. We also identify the potential and make use of additional constraints extensively, in fact, most following techniques rely on them.

Choice of Solver We use the nlsat solver inside Z3 [92], which is based on conflict driven clause learning (CDCL) together with a projection operator adapted from cylindrical algebraic decomposition (CAD) [40]. It supports real closed fields together with division and is thus suitable for our purpose, especially, since Z3 is available through the Leon framework inside which Rosa is implemented. While Z3 supports 'only' real arithmetic and thus our approach is limited to such code, it nonetheless covers a wide range of applications. Our algorithms can be straight-forwardly extended to code including exponential or trigonometric functions, provided a suitable solver is available.

iSAT3 [141] is a possible alternative solver, whose algorithm [64, 142] relies on a tight coupling of interval constraint propagation (ICT) with CDCL. dReal [68] implements a decision procedure based on delta-satisfiability [66] and also uses ICT, but inside a DPLL framework. MetiTarski [7] applies simplifications and approximations to reduce more complex operations to real closed fields and applies CAD on the resulting constraint. This list of possible solvers is certainly not exhaustive and while they support many transcendental functions, they are also necessarily incomplete. Any solver that can determine unsatisfiability of real-valued formulas is suitable and it would be an interesting future project to see how well different solvers can work in our algorithms.

6.1.2 Error Propagation

Since the error propagation as described in subsection 2.4.3 assumed essentially two affine forms and we are replacing one of these by "Z3-powered" intervals, we need to adapt the propagation as well. That is, we want to define the `evalWithError` procedure:

$$\begin{aligned}
&([a,b], err) = \text{evalWithError}(P, expr) \rightarrow \\
&\forall x, \tilde{x}, \mathsf{res}, \tilde{\mathsf{res}}. P(x,\tilde{x}) \wedge \mathsf{res} = \mathsf{expr}(x) \wedge \tilde{\mathsf{res}} = \tilde{\mathsf{expr}}(\tilde{x}) \\
&\qquad\qquad \rightarrow a \le \mathsf{res} \wedge \mathsf{res} \le b \wedge |\mathsf{res} - \tilde{\mathsf{res}}| < \mathsf{err}
\end{aligned}$$

where $\tilde{\mathsf{expr}}$ represents the expression evaluated in finite-precision arithmetic and $x, \tilde{x}$ are the ideal and actual variables. The precondition includes a specification of the ranges and uncertainties of initial variables and other additional constraints on the ideal variables. Recall that the uncertainty specification relates the ideal and actual variables.

The general idea is a before: we "execute" a computation while keeping track of the range of the current intermediate expression and its associated errors. From this range, we can then compute the round-off error committed at each computation step. Instead of using affine arithmetic to compute the intermediate ranges, we use the Z3-backed range computation and adapt the error propagation. In our adaptation, we represent every variable and intermediate computation result as a data type with the following components:

$$x : (\mathsf{range} : \mathsf{Interval}, \hat{\mathsf{err}} : \mathsf{AffineForm})$$

where range is the real-valued range of this variable, computed with Z3 and $\hat{\mathsf{err}}$ is the affine form representing the errors. The (over-approximation) of the actual range including all uncertainties is then given by totalRange = range + $[\hat{\mathsf{err}}]$, where $[\hat{\mathsf{err}}]$ denotes the interval represented by the affine form. In this application, we use the rational implementation of affine arithmetic. The modular nature of our static analyzer keeps the expressions to be analyzed relatively short, making it possible to use rationals instead of double double precision floating-points. We found the rational implementation both clearer and more accurate than a floating-point one.

For the affine operations addition, subtraction, and multiplication by a constant factor the propagated errors are computed term-wise and thus in the same way as for standard affine arithmetic. For multiplication, division and square root, propagation has to be adjusted, since our ranges are not affine terms themselves. In the following, we denote the range of a variable x by $[x]$ and its associated error by the affine form $\hat{\mathsf{err}}_\mathsf{x}$. When we write $[x] * \hat{\mathsf{err}}_y$ we mean that the interval $[x]$ is converted into an affine form and the multiplication is performed in affine arithmetic. Multiplication is computed as

$$\begin{aligned} x * y &= ([x] + \hat{\mathsf{err}}_\mathsf{x})([y] + \hat{\mathsf{err}}_\mathsf{y}) \\ &= [x] * [y] + [x] * \hat{\mathsf{err}}_\mathsf{y} + [y] * \hat{\mathsf{err}}_\mathsf{x} + \hat{\mathsf{err}}_\mathsf{x} * \hat{\mathsf{err}}_\mathsf{y} + \rho \end{aligned}$$

where ρ is the new round-off error, computed as usual. Thus the first term contributes to the ideal range and the remaining three to the error affine form. The larger the factors $[x]$ and $[y]$ are, the larger the finally computed errors will be. In order to keep the over-approximation as small as possible, we evaluate $[x]$ and $[y]$ with our new range computation. Division is computed as

$$\begin{aligned} \frac{x}{y} &= x * \frac{1}{y} = ([x] + \hat{\mathsf{err}}_\mathsf{x})([1/y] + \hat{\mathsf{err}}_{1/y}) \\ &= [x] * [\frac{1}{y}] + [x] * \hat{\mathsf{err}}_{\frac{1}{y}} + [\frac{1}{y}] * \hat{\mathsf{err}}_\mathsf{x} + \hat{\mathsf{err}}_\mathsf{x} * \hat{\mathsf{err}}_{\frac{1}{y}} + \rho \end{aligned}$$

For square root, we first compute an error multiplier based on the affine approximation of square root: $(\sqrt{z})' = \frac{1}{2\sqrt{z}\downarrow}$ where $z = min(|a|, |b|)$ with $[a, b]$ being the input interval and $\downarrow$ denotes rounding towards $-\infty$. The propagated error is obtained by multiplying the error $\hat{err}_x$ by this factor term-wise.

Overflows and NaN Our procedure allows us to detect potential overflows, division by zero and square root of a negative value errors, as our tool computes ranges of all intermediate values. We currently report these issues as compilation errors to the user.

6.1.3 Limitations

The limitation of this approach is clearly the ability of Z3 to check our constraints. We found its capabilities satisfactory, although we expect the performance to still significantly improve. To emphasize the difference to the constraints that are defined by Table 5.1, the real-valued constraints we use here for narrowing down ranges do not include roundoff errors. Since each roundoff introduced one variable for each arithmetic operation, we reduce the number of variables significantly when only considering the real-valued computation. We also found several transformations helpful, such as rewriting powers (e.g. $x * x * x$ to x^3), multiplying out products and avoiding non-strict comparisons in the precondition, although the benefits are not entirely consistent. Note that at each step of our error computation, our tool computes the current range. Thus, even if Z3 fails to tighten the bound for some expressions, we still compute more accurate bounds than interval arithmetic overall in most cases, as the ranges of the remaining subexpressions have already been computed more accurately.

6.1.4 Experimental Results

We have chosen several nonlinear expressions commonly used in physics, biology and chemistry [151, 133, 122] as benchmark functions, as well as benchmarks used in control systems [9] and suitable benchmarks from [56]. Experiments were performed on a desktop computer running Ubuntu 12.04.1 with a 3.5GHz i7 processor and 16GB of RAM. Table 6.1 compares results of our range computation procedure against ranges obtained with standard interval arithmetic. We found affine arithmetic to give more pessimistic results on these benchmarks in our experiments. We believe that this is due to inaccuracies in computing nonlinear operations. Note, however, that we still use affine arithmetic to estimate errors given the computed ranges.

For our range computation, we set the default accuracy threshold to `1e-10` and maximum number of iterations for the binary search to 50. To obtain an idea about the true ranges of our functions, we have also computed a lower bound on the range using simulations with 10^7 random inputs and with exact rational arithmetic evaluation of expressions. We observe that our range computation can significantly improve over standard interval bounds.

Benchmark	Our range	Interval arithmetic	Simulated range
doppler1	[-137.639, -0.033951]	[-158.720, -0.029442]	[-136.346, -0.035273]
doppler2	[-230.991, -0.022729]	[-276.077, -0.019017]	[-227.841,-0.023235]
doppler3	[-83.066, -0.50744]	[-96.295, -0.43773]	[-82.624, -0.51570]
rigidBody1	[-705.0, 705.0]	[-705.0, 705.0]	[-697.132, 694.508]
rigidBody2	[-56010.1, 58740.0]	[-58740.0, 58740.0]	[-54997.635, 57938.052]
jetEngine	[-1997.037, 5109.338]	$[-\infty,\infty]$	[-1779.551, 4813.564]
turbine1	[-18.526, -1.9916]	[-58.330, -1.5505]	[-18.284, -1.9946]
turbine2	[-28.555, 3.8223]	[-29.437, 80.993]	[-28.528, 3.8107]
turbine3	[0.57172, 11.428]	[0.46610, 40.376]	[0.61170, 11.380]
verhulst	[0.31489, 1.1009]	[0.31489, 1.1009]	[0.36685,0.94492]
predatorPrey	[0.039677, 0.33550]	[0.037277, 0.35711]	[0.039669,0.33558]
carbonGas	[4.3032 e6, 1.6740 e7]	[2.0974 e6, 3.4344 e7]	[4.1508 e6, 1.69074 e7]
Sine	[-0.9999, 0.9999]	[-2.3012, 2.3012]	[-0.9999, 0.9999]
Sqrt	[1.0, 1.3985]	[0.83593, 1.5625]	[1.0, 1.3985]
Sine (order 3)	[-1.0001, 1.0001]	[-2.9420, 2.9420]	[-1.0, 1.0]

Table 6.1 – Comparison of ranges computed with out procedure against interval arithmetic and simulation. Simulations were performed with 10^7 random inputs. Ranges are rounded outwards. Affine arithmetic does not provide better results than interval arithmetic.

The `jetEngine` benchmark is a notable example, where interval arithmetic yields the bound $[-\infty,\infty]$, but our procedure can still provide bounds that are quite close to the simulated range.

Triangle Progression Table 6.2 presents another relevant experiment, evaluating the ability to use additional constraints during our range computation. For this experiment, we use double precision and the triangle example from Figure 5.1 with additional constraints allowing increasingly flat triangles by setting the threshold (e.g. `a + b > c + 1e-6`) to the different values given in the first column. As the triangles become flatter, we observe an expected increase in uncertainty on the output since the formula becomes more prone to round-off errors. At threshold `1e-10` our range computation fails to provide the necessary accuracy and the radicand becomes possibly negative. Using our tool, the developer can go beyond rules of thumb and informal estimates and be confident that the computed area is accurate up to seven decimal digits even for triangles that whose difference $a+b-c$ is as small as 10^{-9}.

Benchmark	Area	Max. abs. error
triangle1 (0.1)	[0.29432, 35.0741]	2.019e-11
triangle2 (1e-2)	[0.099375, 35.0741]	6.025e-11
triangle3 (1e-3)	[3.16031e-2, 35.0741]	1.895e-10
triangle4 (1e-4)	[9.9993e-3, 35.0741]	5.987e-10
triangle5 (1e-5)	[3.1622e-3, 35.0741]	1.894e-9
triangle6 (1e-6)	[9.9988e-4, 35.0741]	5.988e-9
triangle7 (1e-7)	[3.1567e-4, 35.0741]	1.897e-8
triangle8 (1e-8)	[9.8888e-5, 35.0741]	6.054e-8
triangle9 (1e-9)	[3.0517e-5, 35.0741]	1.962e-7
triangle10 (1e-10)	-	-

Table 6.2 – Area computed and error on the result for increasingly flat triangles. All values are rounded outwards. Interval or affine arithmetic alone fails to provide any result.

6.2 Error Bounds

Now that we have reasonably accurate ranges, we address error propagation in straight-line nonlinear functions, without loops and branches. The input is a real-valued arithmetic expression representing a function $f : \mathbb{R}^n \to \mathbb{R}$ over some inputs $x_i \in \mathbb{R}$, absolute error bounds on the inputs λ_i and a target precision. The arithmetic operators our tool accepts are $\{+, -, *, /, \sqrt{}\}$. We denote by f and x the exact *ideal* real-valued function and variables and by $\tilde{f} : \mathbb{R}^n \to \mathbb{R}, \tilde{x} \in \mathbb{R}^n$ their *actual* finite-precision counter-parts. Note that for our analysis all variables are real-valued; the finite-precision variable $\tilde{x}$ is considered as a noisy versions of x. $|x - \tilde{x}| \le \lambda$ then defines the input errors, where the absolute value is taken component-wise. We want to bound the absolute error on the result of evaluating $f(x)$ in finite precision arithmetic:

$$|f(x) - \tilde{f}(\tilde{x})| \quad \text{where } |x - \tilde{x}| \le \lambda$$

6.2.1 Separation of Errors

Approaches based on interval or affine arithmetic treat all errors equally in the sense that initial errors are propagated in the same way as round-off errors which are committed during the computation. We propose to separate these errors as follows:

$$\begin{aligned} |f(x) - \tilde{f}(\tilde{x})| &= |f(x) - f(\tilde{x}) + f(\tilde{x}) - \tilde{f}(\tilde{x})| \\ &\le |f(x) - f(\tilde{x})| + |f(\tilde{x}) - \tilde{f}(\tilde{x})| \end{aligned} \tag{6.1}$$

The first term captures the error in the result of f caused by the initial error between x and $\tilde{x}$. The second term covers the round-off error committed when evaluating f in finite-precision, but note that we compute this round-off error from a accurate input without initial error. Thus, we separate the overall error into the propagation of existing errors, and the newly committed round-off errors. Figure 6.2 illustrates this separation.

We denote by $\sigma_f : \mathbb{R}^n \to \mathbb{R}$ the function which returns the round-off error committed when evaluating an expression in finite-precision arithmetic: $\sigma_f(\tilde{x}) = |f(\tilde{x}) - \tilde{f}(\tilde{x})|$. We omit the subscript f, when it is clear from the context. We use the round-off error computation from section 6.1. Further, $g : \mathbb{R}^n \to \mathbb{R}$ denotes a function which bounds the difference in f, given a difference in its inputs: $|f(x) - f(y)| \le g(|x - y|)$. When f is multivariate, the absolute value is component-wise, i.e. $g(|x_1 - y_1|, \ldots, |x_n - y_n|)$, but when it is clear from the context, we will write $g(|x - y|)$ for clarity. Thus, the overall numerical error is given by:

$$|f(x) - \tilde{f}(\tilde{x})| \le g(|x - \tilde{x}|) + \sigma(\tilde{x}) \tag{6.2}$$

This procedure extends naturally to vector-valued functions, where we compute errors component-wise:

$$|f_i(x) - \tilde{f}_i(\tilde{x})| \le g_i(|x - y|) + \sigma_i(\tilde{x}) \qquad \text{where } g, \sigma_f : \mathbb{R}^n \to \mathbb{R}^n.$$

Note that the separation in Equation 6.1 is not unique. For instance, we could have chosen $|f(x) - \tilde{f}(x)| + |\tilde{f}(x) - \tilde{f}(\tilde{x})|$. The first term now corresponds to round-off errors, but the second requires bounding the difference of $\tilde{f}$ over a certain input interval. In the separation that we have chosen, we need to compute the difference over the real-valued f, which is a much simpler function than its finite-precision counterpart.

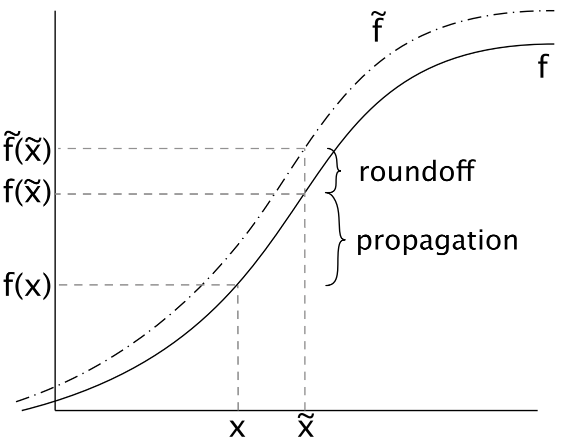


Figure 6.2 – Illustration of the separation of errors

6.2.2 Computing Propagation Coefficients

We instantiate Equation 6.2 with $g(x) = K \cdot x$ and bound the deviation on the result by a linear function in the input errors:

$$|f(x) - f(y)| \leq K|x - y|$$

We will use this definition for most of the rest of this paper. The constant K is to be determined for each function individually, and is usually called the Lipschitz constant. We will also use the in this context more descriptive name *propagation coefficient*. Note that we need to compute the propagation coefficient K for the mathematical function f and not its finite-precision counterpart $\tilde{f}$.

At a high level, error amplification or diminution depends on the derivative of the function at the inputs. The steeper the function, i.e. the larger the derivative, the more the errors are magnified. We formally derive the computation of the propagation coefficients K_i for a multivariate function f in the following.

Let $h : [0,1] \to \mathbb{R}$ such that $h(\theta) := f(y + \theta(z - y))$. Without loss of generality, assume $y < z$. Then $h(0) = f(y)$ and $h(1) = f(z)$ and

$$\frac{d}{d\theta} h(\theta) = \nabla f(y + \theta(z - y)) \cdot (z - y)$$

By the mean value theorem:

$$f(z) - f(y) = h(1) - h(0) = h'(\zeta) \qquad \text{where } \zeta \in [0,1]$$

Then taking absolute values on both sides:

$$\begin{aligned}
&|f(z) - f(y)| = |h'(\zeta)| \\
&\quad = |\nabla f(y + \zeta(z - y)) \cdot (z - y)| \\
&\quad = \left| \left(\frac{\partial f}{\partial x_1}\bigg|_w, \ldots, \frac{\partial f}{\partial x_n}\bigg|_w \right) \cdot (z - y) \right|, \quad w = y + \zeta(z - y) \\
&\quad = \left| \frac{\partial f}{\partial x_1} \cdot (z_1 - y_1) + \cdots + \frac{\partial f}{\partial x_n} \cdot (z_n - y_n) \right| \\
&\quad \leq \left| \frac{\partial f}{\partial x_1} \right| \cdot |z_1 - y_1| + \cdots + \left| \frac{\partial f}{\partial x_n} \right| \cdot |z_n - y_n| \qquad (**)
\end{aligned}$$

where the partial derivatives are evaluated at $w = y + \zeta(z - y)$, which we omit for readability. We compute the partial derivatives symbolically, but before we can evaluate the propagation error, we need to determine over which inputs we need to evaluate them.

Bounding Ranges of Partial Derivatives In the above, the value of w is constraint to be in $w \in [y, z]$, so for a sound analysis we have to determine the maximum absolute value of the partial derivative over $[y, z]$. y and z in our application range over the values of x and $\tilde{x}$ respectively, so we compute the maximum absolute value of $\frac{\partial f}{\partial x_i}$ over the interval that contains the intervals of x and $\tilde{x}$. Both interval and affine arithmetic suffer from possibly large over-approximations, which is why we use the range computation from section 6.1 to bound the ranges of the derivatives. Finally, we have $|y_i - z_i| \leq \lambda_i$ and we obtain

$$|f(x) - f(\tilde{x})| \leq \sum_{i=1}^{n} K_i \lambda_i \tag{6.3}$$

where $K_i = \sup_{x,\tilde{x}} \left| \frac{\partial f}{\partial x_i} \right|$. We show an example of this computation in subsection 6.2.3.

Additional Constraints The propagation coefficients are computed using the input ranges. As we have already seen, we can often restrict the inputs further by additional constraints. Since we are using an SMT solver to bound the ranges of the partial derivatives, these additional constraints can naturally be taken into account to compute more accurate propagation coefficients.

Sensitivity to Input Errors Beyond providing a way to compute the propagated initial errors, Equation 6.3 also makes the sensitivity of the function to input errors explicit, at least to a linear approximation. That is, we can read off from Equation 6.3 by how much initial errors get magnified or diminished by the computation. The user can use this knowledge, for example, to determine which inputs need to be determined more accurately, e.g. by more accurate measurements. We report the values of K back to the user.

6.2.3 Relationship with Affine Arithmetic

Both our presented propagation procedure and propagation using affine arithmetic perform approximations. The question arises then, when is it preferable to use one over the other? Section 6.2.5 and our experience show empirically that for longer nonlinear computations, error propagation based on Lipschitz continuity gives better results, whereas for shorter and linear computations this is not the case. In this section, we present an analysis of this phenomenon based on an example.

Suppose we want to compute $x * y - x^2$. For this discussion we consider propagation only and disregard round-off errors. We consider the case where x and y have an initial error of $\delta_x \epsilon_1$ and $\delta_y \epsilon_2$ respectively, where $\epsilon_i \in [-1, 1]$ are the formal noise symbols of AA. Without loss of generality, we assume $\delta_x, \delta_y \geq 0$. We first derive the expression for the error with affine arithmetic and take the definition of multiplication from subsection 6.1.2. We denote by $[x]$ the evaluation of the *real-valued* range of the variable x.

The total range of x is then the real-valued range plus the error: $[x] + \delta_x \epsilon_1$, where $\epsilon_1 \in [-1, 1]$. Multiplying out, and removing the $[x][y] - [x]^2$ term (since it is no error term), we obtain the expression for the error of $x * y - x^2$:

$$\begin{aligned}&\big([y]\delta_x\epsilon_1 + [x]\delta_y\epsilon_2 + \delta_x\delta_y\epsilon_3\big) - (2[x]\delta_x\epsilon_1 + \delta_x\delta_x\epsilon_4)\\&\quad = ([y] - 2[x])\delta_x\epsilon_1 + [x]\delta_y\epsilon_2 + \delta_x\delta_y\epsilon_3 + \delta_x\delta_x\epsilon_4\end{aligned} \tag{6.4}$$

ϵ_3 and ϵ_4 are fresh noise symbols introduced by the nonlinear approximation. Now we compute the propagation coefficients:

$$\frac{\partial f}{\partial x} = y - 2x \qquad \frac{\partial f}{\partial y} = x$$

The error is then given by

$$\Big|[y + \delta_y\epsilon_2 - 2(x + \delta_x\epsilon_1)]\Big|\delta_x + \Big|[x + \delta_x\epsilon_1]\Big|\delta_y \tag{6.5}$$

We obtain this expression by instantiating Equation (**) with the range expressions of x and y. Note that the ranges used as the inputs for the evaluation of the partial derivatives include the errors. Multiplying out Equation 6.5 we obtain:

$$\Big|[y - 2x]\Big|\delta_x + \Big|[x]\Big|\delta_x + \delta_x\delta_y + \delta_x\delta_x + \delta_x\delta_x \tag{6.6}$$

With affine arithmetic we compute ranges for propagation at each computation step, i.e. in Equation 6.4 we compute $[x]$ and $[y]$ separately. In contrast, with our new technique, the range is computed once, taking all correlations into account between the variables x and y. It is these correlations that improve the computed error bounds. For instance, if we choose $x \in [1,5]$ and $y \in [-1,2]$ and we know that $x < y$, then by a step-wise computation we obtain $[y] - 2[x] = [-1,2] - 2[1,5] = [-11,0]$ whereas taking the correlations into account, we can narrow down the range of x to $[1,2]$ and obtain $[y-2x] = [-1,2]-2[1,2] = [-5,0]$. Hence, since we compute the maximum absolute value of these ranges for the error computation, AA will use the factor 11, whereas our approach will use 5.

On the other hand, comparing Equation 6.6 with Equation 6.4, we see that one term $\delta_x\delta_x$ is included twice with our approach, whereas in the affine propagation it is only included once. We conclude that a Lipschitz-based error propagation is most useful for longer computations where it can leverage correlations. In other cases we keep the existing affine arithmetic-based technique. It does not require a two-step computation, so we want to use it for smaller expressions. We remark that for linear operations the two approaches are equivalent.

6.2.4 Higher Order Taylor Approximation

In subsection 6.2.2 we presented one possible instantiation of the error propagation function g. The resulting propagation function is a function in the input errors. The errors do, however, also depend on the ranges of the inputs. This fact is only implicitly reflected in the computed coefficients via the ranges used for bounding the partial derivatives. We can in fact make this relationship more explicit. Recall Taylor's Theorem in several variables:

Taylor's Theorem Suppose $f : \mathbb{R}^n \to \mathbb{R}$ is of class C^{k+1} on an open convex set S. If $a \in S$ and $a + h \in S$, then

$$f(a+h) = \sum_{|\alpha| \le k} \frac{\partial^\alpha f(a)}{\alpha!} h^\alpha + R_{\alpha,k}(h)$$

where the remainder in Lagrange's form is given by:

$$R_{\alpha,k}(h) = \sum_{|\alpha|=k+1} \frac{\partial^\alpha f(a+ch)}{\alpha!} h^\alpha$$

for some $c \in (0,1)$. □

Computing the Taylor expansion of $f(\tilde{x})$ to first order in our setting:

$$f(\tilde{x}) = f(x) + \sum_{j=1}^{n} \partial_j f(x) h_j + \frac{1}{2} \sum_{j,k=1}^{n} \partial_j \partial_k f(w) h_j h_k$$

and taking absolute values, we obtain

$$\left| f(\tilde{x}) - f(x) \right| \le \left| \sum_{j=1}^{n} \partial_j f(x) h_j \right| + \frac{1}{2} \left| \sum_{j,k=1}^{n} H_{jk}(w) h_j h_k \right|$$

where w is in the interval containing x and $\tilde{x}$, and H is the Hessian matrix of f. If we consider the expansion for $k = 1$, we obtain an expression for computing the upper bound on the propagated error which is also a function of the *input values.*

We observe that the second order taylor remainder is in general small, due to the fact that we take the square of the initial errors, which we assume to be small in our applications. We can bound the remainder with the same technique we use to compute the propagation coefficients. Then, together with the partial derivatives of f, we obtain error specifications which can be used for a more accurate modular verification process.

Application to Interprocedural Analysis Having more accurate specifications enables us to re-use methods across different call-sites, with possibly different constraints on the arguments. We present an example in subsection 6.2.5 which demonstrates the effectiveness of this summarization technique. We are not aware of other work that is capable of computing such summaries for numerical errors. [74] presents an approach to compute method summaries based on affine arithmetic evaluation and instantiation. These summaries, however, capture the real-valued ranges only and not the numerical errors.

6.2.5 Experimental Results

We perform our experiments mostly with double precision unless otherwise specified, as this is a common choice for numerical programs. Note however, that Rosa supports both floating-point arithmetic with different precisions, as well as fixed-point arithmetic with different bitlengths. In our experience, while the absolute errors naturally change with varying precisions and data types, the relative differences in comparisons remain very similar.

We compare our results against those obtained by Fluctuat, which is the only other tool that we are aware of that can compute sound numerical error bounds. We further compare two versions of Rosa: one where the round-off error computation is based on only affine arithmetic with accurate ranges, as described in section 6.2, and the other where we use our Lipschitz-based error computation. We will call the latter version Rosa+. Unless otherwise stated, all numerical error values are rounded. Experiments were performed on a desktop computer running Ubuntu 12.04.4 with a 3.5GHz i7 processor and 16GB of RAM.

Straight-line Nonlinear Computation

We first evaluate our error propagation technique for straight-line nonlinear code on our benchmarks from section 6.1. The results are summarized in Table 6.3. The error computations in Rosa and in Fluctuat are very similar, and essentially differ only in how the ranges of variables are constrained. We use an SMT solver to narrow down ranges, whereas Fluctuat uses a logical product of the affine forms with an abstract domain [70]. We also list an under-approximation of the errors, which we obtained with a simulation with 10^7 random inputs. We are not aware of such a thorough comparison having been performed before.

The initial errors in the top benchmarks of Table 6.3 are round-off errors only, in the bottom section we add an initial absolute error of $1e-11$ to all inputs. We can see that Rosa+ can improve the computed error estimates in most cases. Even for benchmarks where the initial errors are only round-offs, we can still sometimes halve the error obtained with previous tools. For benchmarks with additional initial errors the effect is even larger.

Furthermore, we investigate the effect of refactoring expressions and applying our error propagation technique to each subexpression. For example, in the case of the doppler benchmark, we consider two formulations:

```
(- (331.4 + 0.6 * T) *v) / (((331.4 + 0.6 * T) + u)*((331.4 + 0.6 * T) + u))
```

which is often the formulation produced by code generation tools, and

```
val tmp = 331.4 + 0.6 * T
(-tmp * v) / ((tmp + u)*(tmp + u))
```

In the second case, we apply the error propagation twice, once for computing the error on `tmp` and once for the error on the result. The hope is to compute intermediate values

benchmark	Simulated	Fluctuat	Rosa	Rosa+
doppler	7.11e-14	**3.90e-13**	4.36e-13	4.29e-13
dopplerRefactored		3.90e-13	4.19e-13	**2.68e-13**
jetengine	5.46e-12	4.08e-8	1.16e-8	**5.33e-9**
jetengineRefactored		4.08e-8	1.16e-8	**4.91e-9**
rigidBody1	2.28e-13	3.22e-13	3.22e-13	3.22e-13
rigidBody2	2.19e-11	3.65e-11	3.65e-11	3.65e-11
rigidBody2Refactored		3.65e-11	3.65e-11	3.65e-11
sine	4.45e-16	7.97e-16	6.40e-16	**5.18e-16**
sineOrder3	3.34e-16	1.15e-15	1.23e-15	**9.96e-16**
sqroot	4.45e-16	3.21e-13	3.09e-13	**2.87e-13**
turbine1	1.07e-14	9.20e-14	8.87e-14	**5.99e-14**
turbine1Refactored		9.26e-14	8.87e-14	**5.15e-14**
turbine2	1.43e-14	1.29e-13	1.23e-13	**7.67e-14**
turbine2Refactored		1.34e-13	1.23e-13	**6.30e-14**
turbine3	5.33e-15	6.99e-14	6.27e-14	**4.62e-14**
turbine3Refactored		7.03e-14	6.27e-14	**4.01e-14**
verhulst	2.23e-16	**4.24e-16**	4.74e-16	4.67e-16
predatorPrey	1.12e-16	2.09e-16	2.08e-16	**1.98e-16**
carbonGas	3.73e-9	4.02e-8	3.35e-8	**1.60e-8**
with added errors				
dopplerRefactored	1.65e-11	5.45e-11	5.29e-11	**2.08e-11**
jetengineRefactored	3.64e-8	4.67e-4	1.41e-4	**3.36e-7**
turbine1Refactored	4.09e-10	1.82e-9	1.88e-9	**4.60e-10**
turbine2Refactored	5.10e-10	2.82e-9	2.90e-9	**5.86e-10**
turbine3Refactored	2.10e-10	1.24e-9	1.27e-9	**3.32e-10**

Table 6.3 – Comparison of computed absolute errors from Fluctuat, and our two error computations in Rosa on straight-line, nonlinear benchmarks. Simulations were performed with 10^7 random inputs. We mark the best result per benchmark in bold.

more accurately with our technique and thus improve the overall bounds even further. The experimental results confirm the benefit of this step-wise error computation. Rosa+ currently performs the error propagation only for expressions defined as `val`s or final expressions, as too fine-grained steps would increase the running time unnecessarily or degrade the computed results.

Overall, we remark that many of the soundly computed errors actually come quite close to the true errors, as indicated by the simulation results. We thus believe that our technique and our accuracy is very well suitable for the analysis of numerical computation kernels.

Sensitivity The propagation coefficients provide information about the sensitivity of a function to input errors. For example, for the non-refactored doppler benchmark, the computed coefficients for `u, v, T` are

3.216238, 0.006881929, 0.7557531

respectively. Thus, absolute input errors on `u` get magnified by approximately 3.22 in the worst case, whereas input errors on `v` have a much more favorable factor of below one. This information can, for instance, be used to direct optimization efforts, and we report the computed propagation coefficients in comments in the generated code.

Running times

Table 6.4 compares the running times of Rosa and Rosa+ for selected benchmarks. Fluctuat in general computed the result within one second, since it is not using an SMT solver internally. We have not specifically optimized our implementation, and improvements are certainly possible. Nevertheless, while our two-step computation clearly increases the runtime, we believe that the times do remain acceptable for a static verification approach.

First-order Method Summaries Section 6.2.4 introduced a possible extension of the propagation coefficients to postconditions where the errors are functions of both the initial errors and the ranges of the corresponding variable. Here we give a possible scenario how these 'Taylor summaries' can be used. Recall that Rosa's verification framework is modular in that each method is verified separately, and method postconditions are used, where possible, at call sites. The specifications have to be general however, to allow a method to be used in many instances, yet accurate enough to facilitate a successful verification.

For example, consider the following seventh order approximation to the sine function, as it may be used in an embedded system, where trigonometric functions are often approximated.

```
def sine(x: Real): Real = {
  require(-3.5 < x && x < 3.5 && x +/- 1e-8)

  x - (x*x*x)/6.0 + (x*x*x*x*x)/120.0 -
     (x*x*x*x*x*x*x)/5040.0
} ensuring(res => -1.0 < ~res && ~res < 1.0 && res +/- 2e-7)
```

Runtime	Fluctuat	Rosa	Rosa+
doppler		5	23
dopplerRefactored		5	19
jetEngine		120	454
jetEngineRefactored		118	400
rigidBody1		0.1	0.2
rigidBody2		6	8
rigidBody2Refactored		6	7
sine		3	4
sineOrder3	< 1	0	1
sqroot		1	2
turbine1		1	18
turbine1Refactored		1	2
turbine2		1	6
turbine2Refactored		1	2
turbine3		1	19
turbine3Refactored		1	5
verhulst		4	10
predatorPrey		1	24
carbonGas		4	26

Table 6.4 – Running times of our approach compared with Rosa (in seconds)

The postcondition is successfully verified for the given range and input error. But what if, at a call site, the range or the initial error is smaller? Consider two calls to `sine`

```
require(-0.5 <= y && y <= 0.5 && y +/- 1e-8)
...
 sineTaylor(y)


require(-3.0 <= z && z <= 1)
...
 sineTaylor(z)
```

With Rosa, one can either use the postcondition with given error on the result of 2e-7, or inline the function and essentially re-do the error computation. In contrast, our approach described in subsection 6.2.4 will instead use the computed summaries and determine the error for the first case to be 1.000e-8 and for the second case 4.945e-15, improving the error bounds by more than 6 decimal orders of magnitude. This illustrates the benefits of relational summaries that our approach computes.

Conclusion

In this chapter we have presented techniques for an accurate static analysis of numerical errors and ranges. They crucially rely on the use of a nonlinear SMT solver to incorporate correlations between variables from the computation and from additional (external) constraints in the computation of ranges. Furthermore, we show that a separation of errors strategy opens up possibilities for more specialized techniques that can compute tighter error estimates. Effectively, our error propagation computation can employ analytical information about the real-valued mathematical function and improve the error bounds substantially. We believe that our experimental results show that this effort is worth spent.

7 Handling Control Structures

Until now, we have considered numerical error estimation for straight-line code without branches and loops, which we cover in this chapter. The material in this chapter is based on [47] and subsequent extensions [46].

7.1 Discontinuity Errors

By *discontinuity error* we mean the difference between the ideal and actual computation due to uncertainties on computing branch conditions and the resulting different paths taken. So unlike the runtime solution presented in subsection 3.1.1, which merely detects that control flow may diverge, here we are interested in a more rigorous treatment in the sense that we want to *quantify* the resulting difference between the real-valued and the finite-precision computation. Quantifying discontinuity errors is hard due to many correlations between variables of the two branches, but also due to nonlinearity.

Embedded systems often use piece-wise approximations of more complex functions. In Figure 7.1 we show a possible piece-wise polynomial approximation of the jet engine controller from Figure 5.5. We obtained this approximation by fitting a polynomial to a sample of values of the original function. For many applications it is crucial that the error introduced by the discontinuity remains small, since we may otherwise risk unstable behavior. The real-valued difference between the two branches is at most 0.21021. However this is not a sound estimate for the discontinuity error in the presence of round-off and initial errors. With our tool, we can confirm that the discontinuity error is bounded by 0.21202, with all errors taken into account.

We note that the direct encoding of the computation constructed according to Section 5.2.1 automatically includes discontinuity errors. Recall that we encode the ideal and actual computations such that they are independent except for the initial conditions. Because of this independence it is possible that they follow different paths through the program. When we cannot encode the actual computation and approximate it instead, we compute the error on individual paths and have to consider the error due to diverging paths separately.

```
def jetApproxGoodFit(x: Real, y: Real): Real = {
  require(-5 <= x && x <= 5 && -5 <= y && y <= 5 &&
    x +/- 1e-11 && y +/- 1e-11)

  if (y < x)
    -0.367351 + 0.0947427*x + 0.0917402*x*x - 0.00298772*y +
      0.0425403*x*y + 0.00204213*y*y
  else
    -0.308522 + 0.0796111*x + 0.162905*x*x + 0.00469104*y -
      0.0199035*x*y - 0.00204213*y*y
}
```

Figure 7.1 – Piece-wise approximation of the jet engine controller

We propose two algorithms to explicitly compute the difference between the ideal and the actual computation across paths. Neither method assumes continuity or any other specific property of the function, i.e. the algorithm allows us to compute error bounds even in the case of non-smooth or non-continuous functions. For simplicity, we present here the algorithms for the case of one conditional statement:

```
if (c(x)<0) f1(x) else f2(x)
```

They generalize readily to more complex expressions by rewriting the branches into this form. Using our previous notation, let f_1 and f_2 be the real- valued functions corresponding to the **if** and the **else** branch respectively. Then, the discontinuity error is given by $|f_1(x) - \tilde{f}_2(\tilde{x})|$ That is, the real computation takes branch f_1, and the finite-precision one f_2. The opposite case is analogous:

```
def getDiscontinuityError:
Input: pre: ($x \in [a,b] \wedge x \pm n$)
     program: (if (cond(x) < 0) f1(x) else f2(x))

  val discError1 = computeDiscError(pre, cond, f1, f2)
  val discError2 = computeDiscError(pre, ¬ cond, f2, f1)
  return max (discError1, discError2)
```

7.1.1 Precise Constraint Solving

Formulating it as a separation of errors, our technique computes the following:

$$|f_1(x) - \tilde{f}_2(\tilde{x})| \leq |f_1(x) - f_2(\tilde{x})| + |f_2(\tilde{x}) - \tilde{f}_2(\tilde{x})|$$

In order to obtain a useful result, our first technique constructs an accurate constraint relating the variables from f_1 to the variables in f_2 while computing the first difference. The second part is then simply the round-off error from evaluating f_2 in finite precision.

```
def computeDiscError(pre, c, f1, f2):
  ([c], err_c) = evalWithError(pre, c)

  varConstraint = c(x_1) ∈ [−err_c, 0] ∧
                  c(x_2) ∈ [0, err_c] ∧
                  relate(x_1, x_2, err_c)

  [diff] = getRange(pre ∧ varConstraint, f1(x_1) - f2(x_2) )

  ([f2]_float, err_float) = evalWithError(pre ∧ c(x_2) ∈ [0, err_c], f2)

  return: max | [diff] | + err_float
```

Figure 7.2 – Computation of errors due to diverging paths. Quantities denotes by [x] are intervals.

W.l.o.g. we assume that the condition is of the form $c(x) < 0$. Indeed, any conditional of the form $c(x) == 0$ would yield different results for the ideal and actual computation for nearly any input, so we do not allow it in our specification language.

The actual finite-precision computation commits a certain error when computing the condition of the branch and it is this error that causes some executions to follow a different branch than the corresponding ideal one would take. Figure 7.2 presents our first algorithm to compute the discontinuity error. The idea is to consider the computation along f_1 and f_2 as separate computations with independent variables x_1 and x_2, which are related for the computation of the difference $f_1 - f_2$ to be at most err_c apart. Additionally, x_1 and x_2 are restricted such that $c(x_{1,2}) \in [-\mathsf{err}_c, \mathsf{err}_c]$, that is, we compute the difference only for the values for which the computation could be diverging. We compute the difference as an interval, since the inputs x_1 and x_2 are also interval valued. The overall error is then the real-valued difference plus the round-off error committed by f_2. The algorithm extends naturally to several variables.

In the case of several paths through the program, this error has to be, in principle, computed for each pair of paths. We use Z3 to rule out infeasible paths up front so that the path error computation is only performed for those paths that are actually feasible. Rosa implements this algorithm while merging paths at every branch. It also uses a higher default accuracy and number of iterations threshold during the binary search in the range computation as this computation requires in general very tight intervals as the errors can be small.

We identify two challenges for performing this computation:

1. As soon as the program has multiple variables, the inputs for the different branches are not two-dimensional intervals anymore, which makes an accurate evaluation of the difference difficult.

2. The inputs for the two branches are inter-dependent. Simply evaluating the two branches with inputs that are in the correct ranges, but are not correlated, yields pessimistic results when computing the difference (line 6).

We overcome the first challenge with our range computation which takes into account additional constraints. For the second challenge, we use our range computation as well. Unfortunately, Z3 fails to tighten the final range to a satisfactory accuracy due to timeouts for more complex examples, especially with several variables. We still obtain much better error estimates than with interval arithmetic alone, as the ranges of values for the individual paths are already computed much more accurately.

Fluctuat also includes a procedure to evaluate discontinuity errors and takes a similar route. It does not use an SMT solver, but it constrains the noise terms of the real and floating-point computation in its abstract domain based on a logical product with the interval domain [73]. We will show later in the experimental results that this approach does not yield satisfactory results either. Furthermore, the underlying domain is linear, hence nonlinearity in f_1 and f_2 cause further over-approximations.

7.1.2 Trading Accuracy for Scalability

While our first approach is accurate for unary functions and when they are not too complex, the accurate constraint which correlates x_1 and x_2 becomes quickly too complex. We trade some of the accuracy that this constraint provides for scalability, by taking the separation of errors idea further. We now separate the error in three parts:

$$\begin{aligned}&|f_1(x) - \tilde{f}_2(\tilde{x})|\\&\qquad \le |f_1(x) - f_1(\tilde{x})| + |f_1(\tilde{x}) - f_2(\tilde{x})| + |f_2(\tilde{x}) - \tilde{f}_2(\tilde{x})|\end{aligned} \tag{7.1}$$

Figure 7.3 illustrates this separation. The individual components are

i) $|f_1(x) - f_1(\tilde{x})|$: the difference in f_1 due to initial errors. We can compute this difference with our previously defined propagation coefficients: $|f_1(x) - f_1(\tilde{x})| \le K|x - \tilde{x}|$.

ii) $|f_1(\tilde{x}) - f_2(\tilde{x})|$: the real-valued difference between f_1 and f_2. We can bound this value by our range computation.

iii) $|f_2(\tilde{x}) - \tilde{f}_2(\tilde{x})|$: the round-off error when evaluating f_2 in finite-precision arithmetic. We use the same procedure as in chapter 6.

We expect the individual parts to be easier to handle for the underlying SMT-solver, since we reduce the number of variables and correlations. Fluctuat and our first technique compute the discontinuity error as *one* difference between the computations on the two paths of a branch. In contrast, here we split the error and compute its parts separately, obtaining a more scalable

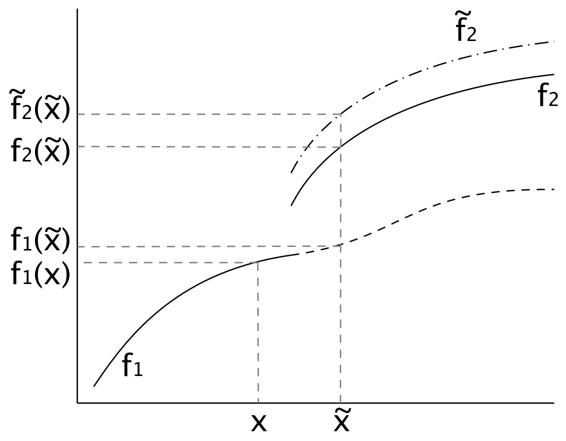


Figure 7.3 – Illustration of error computation for conditional branches

procedure. On the other hand, we clearly introduce an additional over-approximation, but we will show in our experiments that this is in general small, even for benchmarks where the accurate approach from the previous section performs well. For more complex benchmarks our tool outperforms the more accurate approach by far.

We perform our analysis pairwise for each pair of paths in the program. While this gives in the worst-case an exponential number of cases to consider, we found that many of these cases are infeasible due to inconsistent branch conditions and can be eliminated early. Note also, that because we treat loops analytically as opposed to by unrolling, we do not encounter many nested conditionals in practice. Our tool accepts any branch condition that is of the form `e1 ;∘; e2`, where `e1` and `e2` are arithmetic expressions and $\circ \in \{<, \leq, >, \geq\}$.

Determining 'Critical Inputs'

As in the previous section, it is crucial to determine the ranges of $x, \tilde{x}$ over which to evaluate the individual parts of Equation 7.1. Recall that for the analysis, x and $\tilde{x}$ are real-valued. A sound approach would be to use the same bounds as for the straight-line case, but this would lead to unnecessary over-approximations. In general, not all inputs can exhibit a divergence between the real-valued and the finite-precision computation. We call those which possibly do the 'critical inputs'. They are determined by the branch conditions and the errors on the variables. Consider the branch condition **`if`** `(e1 < e2)` and the case where the real-valued path takes the if-branch, i.e. variable x satisfies $e1 < e1$ and $\tilde{x}$ satisfies $e1 \geq e2$. The constraint for the finite-precision variables $\tilde{x}$ is then

$$e1 + \delta_1 < e2 + \delta_2 \wedge e1 \geq e2$$

where δ_1, δ_2 are error intervals capturing the numerical error on evaluating $e1$ and $e2$ respectively. This constraint expresses that we want those values which satisfy the condition $e1 \geq e2$, but are "close enough" to the boundary such that their corresponding ideal real value could take the other path. In practice, we will merge the two error variables and only add one to the constraint: $\delta = \delta_2 - \delta_1$. The procedure for other branch conditions is analogous.

We create such a constraint both for the variables representing finite-precision values ($\tilde{x}$), as well as the real-valued ones x and use them as additional constraints when computing the individual parts of Equation 7.1.

The constraints just described essentially express the correlations between x and $\tilde{x}$, but they do not necessarily narrow down their overall ranges. These are, however, important for the computation of the round-off errors (third part of Equation 7.1), since these depend directly on the ranges. If the branch condition is a "range constraint", that is, if it is of the form $x < c$, where c is a constant, then we use these constraints to tighten the variable ranges. If, however, the branch condition is a relative condition such as $x < y$, then the ranges of x and y remain unconstraint, i.e. x and y individually can still take all the values in their input ranges. In this case, we have to accept the resulting over-approximation, but we found that in general the contribution of the round-off error term to be relatively small anyway.

7.1.3 Experimental Results

We compare the results of our techniques for computing discontinuity errors against those implemented in Fluctuat in Table 7.1. Rosa 1st and 2nd denotes the first and second technique respectively. The first part of the table contains unary functions. `simpleInterpolator` and `squareRoot` are taken from [73], and we added the `squareRoot3` and `cubic spline` benchmarks which we show in Figure 7.4. All of these examples are representative of code one may find in numerical programs such as embedded systems. We observe that our first technique indeed can leverage its more accurate constraint and compute tighter bounds.

The second half of the table presents results for benchmarks in two variables. We have derived these benchmarks by piece-wise approximating a complex function, a common pattern seen in embedded systems. Figure 7.4 shows two representative benchmarks. (0.1) indicates that inputs have an added initial error of 0.1, otherwise we assume round-off errors only. (X) indicates that the benchmark's branch condition is relational (e.g. `x < y`). We can see, that except for the benchmark quadratic fit2 our new technique outperforms the existing ones significantly, sometimes by several orders.

For benchmarks marked with *, we can confirm that the result determined by our tool closely match the discontinuity present in the real-valued function. Our tool, of course, also considers round-off errors, but for benchmarks with small input errors, real-valued discontinuity error dominate (and often arises from the piece-wise approximation approach used to construct the benchmark). The results thus show that our three-way separation of errors lets us decouple the mathematical problem from the finite-precision implementation and treat each part appropriately.

```
def cubicSpline(x: Real): Real = {
  require(-2 <= x && x <= 2)
  if (x <= -1)
    0.25 * (x + 2)* (x + 2)* (x + 2)
  else if (x <= 0)
    0.25*(-3*x*x*x - 6*x*x +4)
  else if (x <= 1)
    0.25*(3*x*x*x - 6*x*x +4)
  else
    0.25*(2 - x)*(2 - x)*(2 - x)
}

def squareRoot3Invalid(x: Real): Real = {
  require(0 < x && x < 10 && x +/- 1e-10 )

  if (x < 1e-4) 1 + 0.5 * x
  else sqrt(1 + x)
}

def quadraticFitWithError(x: Real, y: Real): Real = {
  require(-4 <= x && x <= 4 && -4 <= y && y <= 4 && x +/- 0.1 && y +/- 0.1)

  if (x <= 0)
    if (y <= 0) {
      -0.0495178 - 0.188656*x - 0.0502969*x*x - 0.188656*y +
       0.0384002*x*y - 0.0502969*y*y
    } else {
      0.0495178 + 0.188656*x + 0.0502969*x*x - 0.188656*y +
       0.0384002*x*y + 0.0502969*y*y
    }
  else
    if (y <= 0) {
      0.0495178 - 0.188656*x + 0.0502969*x*x + 0.188656*y +
       0.0384002*x*y + 0.0502969*y*y
    } else {
      -0.0495178 + 0.188656*x - 0.0502969*x*x + 0.188656*y +
       0.0384002*x*y - 0.0502969*y*y
    }
}

def styblinski2(x: Real, y: Real): Real = {
  require(-5 <= x && x <= 5 && -5 <= y && y <= 5)

  if (y < x)
    -2.60357 + 0.25*x + 0.369643*x*x + 0.889286*y - 4.32143e-16*x*y - 0.896429*y*y
  else
    -3.76071 + 0.25*x + 0.369643*x*x + 0.889286*y + 5.08864e-16*x*y - 0.703571*y*y
}
```

Figure 7.4 – Representative benchmarks

benchmark	Fluctuat	Rosa 1st	Rosa 2nd
simpleInterpolator	3.45e-5	**2.346e-5**	3.401e-5
squareRoot	0.0394	**0.02365**	0.02382
squareRoot3	0.429	**1.308e-9**	1.313e-9
cubic spline	12.01	**1.499e-15**	**1.499e-15**
linear fit	1.721	**0.6374**	**0.6374***
quadratic fit	10.61	3.218	**0.2548***
quadratic fit (0.1)	11.25	3.226	**0.2904**
quadratic fit2 (X)	0.6322	**9.195e-16**	1.094e-15
quadratic fit2 (X, 0.1)	0.7781	**0.05583**	0.08554
styblinski	223.5	70.41	**0.6320***
styblinski (0.1)	239.91	71.01	**3.250**
styblinski2 (X)	30.495	18.69	**3.665***
styblinski2 (X, 0.1)	33.19	19.36	**4.863**
jetApprox	25.4	10.91	**0.1702***
jetApprox - good fit (X)	5.73	4.255	**0.2121***
jetApprox - bad fit (X)	15.32	3.822	**1.358***

Table 7.1 – Comparison of computed absolute errors with our new Lipschitz constant-based approach against state of the art on benchmarks with discontinuities.

Table 7.2 shows that the runtimes of our two techniques are comparable. Fluctuat returns very fast, so we do not list it here. It's results are, however, significantly worse than Rosa's and we believe that the performance trade-off by using an SMT-solver is well justified. The runtimes also show that we can gain accuracy by performing a three-way separation of errors without loosing performance notably over the two-way approach, which makes the second technique clearly preferable on more complex examples.

benchmark	Rosa	Our tool
simpleInterpolator (float)	1	1
squareRoot (F)	3	22
squareRoot3	6	5
squareRoot3 invalid	7	7
cubic spline	14	17
natural spline	18	18
linear fit	3	4
quadratic fit	45	54
quadratic fit (0.1)	66	70
quadratic fit2	14	54
quadratic fit2 (0.1)	19	20
styblinski	118	52
styblinski (0.1)	220	74
sortOfStyblinski	53	33
sortOfStyblinski (0.1)	73	25
jetApprox	126	139
jetApprox (0.1)	161	236
jetApprox - good fit	46	33
jetApprox - good fit (0.1)	38	26
jetApprox - bad fit	108	219
jetApprox - bad fit (0.1)	105	157

Table 7.2 – Comparison of runtimes on discontinuity benchmarks. Running times are in seconds.

7.2 Loops

For loops where errors grow without a constant absolute bound, current tools are forced to unroll the loops or apply widening, often returning a trivial upper bound of ∞. Even if the loop is bounded, unrolling often scales poorly. We propose to compute the numerical errors as a function of the number of iterations. This allows us to derive a closed form expression on the loop's error which constitutes an inductive loop invariant and also characterizes the loop's behavior. This expression can also be used to compute concrete error bounds for any given number of loop iterations, often returning better results than unrolling.

In order to derive the closed-form expression, we apply again the idea of propagation of errors. We want to compute the overall error after m-fold iteration f^m of f. We define for any function H: $H^0(x) = x$, $H^{m+1}(x) = H(H^m(x))$. We are thus interested in bounding:

$$|f^m(x) - \tilde{f}^m(\tilde{x})|$$

f, g and σ are now vector-valued: $f, g, \sigma : \mathbb{R}^n \to \mathbb{R}^n$, because we are nesting the potentially multivariate function f.

Theorem 7.1 *Let g be such that $|f(x) - f(y)| \leq g(|x - y|)$, it satisfies $g(x + y) \leq g(x) + g(y)$ and is monotonic. Further, σ and λ satisfy $\sigma(\tilde{x}) = |f(\tilde{x}) - \tilde{f}(\tilde{x})|$ and $|x - \tilde{x}| \leq \lambda$. The absolute value is taken component-wise. Then the numerical error after m iterations is given by*

$$|f^m(x) - \tilde{f}^m(\tilde{x})| \leq g^m(|x - \tilde{x}|) + \sum_{i=0}^{m-1} g^i(\sigma(\tilde{f}^{m-i-1}(\tilde{x}))) \tag{7.2}$$

Proof: We show this by induction. The base case $m = 1$ has already been covered in subsection 6.2.1. By adding and subtracting $f(\tilde{f}^{m-1}(\tilde{x}))_1$ we get

$$\begin{pmatrix} |f^m(x)_1 - \tilde{f}^m(\tilde{x})_1| \\ \vdots \\ |f^m(x)_n - \tilde{f}^m(\tilde{x})_n| \end{pmatrix}$$

$$\leq \begin{pmatrix} |f^m(x)_1 - f(\tilde{f}^{m-1}(\tilde{x}))_1| \\ \vdots \\ |f^m(x)_n - f(\tilde{f}^{m-1}(\tilde{x}))_n| \end{pmatrix} + \begin{pmatrix} |f(\tilde{f}^{m-1}(\tilde{x}))_1 - \tilde{f}^m(\tilde{x})_1| \\ \vdots \\ |f(\tilde{f}^{m-1}(\tilde{x}))_n - \tilde{f}^m(\tilde{x})_n| \end{pmatrix}$$

Applying the definitions of g and σ

$$\leq g\begin{pmatrix} |f^{m-1}(x)_1 - \tilde{f}^{m-1}(\tilde{x})_1| \\ \vdots \\ |f^{m-1}(x)_n - \tilde{f}^{m-1}(\tilde{x})_n| \end{pmatrix} + \sigma(\tilde{f}^{m-1}(\tilde{x}))$$

then using the induction hypothesis and monotonicity of g,

$$\leq g\left(g^{m-1}(\vec{\lambda}) + \sum_{i=0}^{m-2} g^i(\sigma(\tilde{f}^{m-i-1}(\tilde{x})))\right) + \sigma(\tilde{f}^{m-1}(\tilde{x}))$$

then using $g(x+y) \leq g(x) + g(y)$, we finally have

$$\begin{aligned}&\leq g^m(\vec{\lambda}) + \sum_{i=1}^{m-1} g^i(\sigma(\tilde{f}^{m-i-1}(\tilde{x}))) + \sigma(\tilde{f}^{m-1}(\tilde{x}))\\ &= g^m(\vec{\lambda}) + \sum_{i=0}^{m-1} g^i(\sigma(\tilde{f}^{m-i-1}(\tilde{x}))) \qquad \blacksquare\end{aligned}$$

In words, the overall error after m iterations can be decomposed into the initial error propagated through m iterations, and round-off error from the i^{th} iteration propagated through the remaining iterations.

7.2.1 Closed Form Expression

We instantiate the propagation function g as before and would like to derive a closed-form expression for the error, as Equation 7.2 is not very evaluation friendly. In fact, evaluating Equation 7.2 as given, with a fresh set of propagation coefficients for each iteration i amounts to loop unrolling, but with a loss of correlation between each loop iteration.

Suppose we can compute K as a matrix of propagation coefficients, and similarly obtain $\sigma(\tilde{f}^i) = \sigma$ as a vector of constants, both valid over all iterations. Then we obtain a closed-form for the expression of the error:

$$\begin{aligned}|f^m(x) - \tilde{f}^m(\tilde{x})| &\leq K^m\lambda + \sum_{i=1}^{m-1} K^i\sigma + \sigma\\ &= K^m\lambda + \sum_{i=0}^{m-1} K^i\sigma\end{aligned}$$

where λ is the vector of initial errors. If $(I-K)^n$ exists,

$$|f^m(x) - \tilde{f}^m(\tilde{x})| \leq K^m\lambda + ((I-K)^{-1}(I-K^m))\sigma$$

We obtain K^m with power-by-squaring and compute the inverse with the Gauss-Jordan method with rational coefficients to obtain sound results. When $K = I$, g becomes the identity function and so

$$|f^m(x) - \tilde{f}^m(\tilde{x})| \leq \lambda + \sum_{i=1}^{m-1} \sigma + \sigma = \lambda + m \cdot \sigma$$

Now the question remains how to determine K and σ. As before, this boils down to determining the ranges of the variables $x, \tilde{x}$, over which to compute the coefficients of $K_{ij} = \sup_{x,\tilde{x}}(\frac{\partial f_i}{\partial x_j})$ and which to use for the round-off error computation. We consider two cases.

Inductive Constant Ranges When the ranges of the variables of the loop are inductive, that is, both the real-valued and the finite-precision values remain within the initial ranges, then these are clearly the ranges for the computation of K and round-offs σ. We require the user to specify both the real-valued ranges of variables (e.g. `a <= x && x <= b`) as well as the actual finite-precision ones (`c <= ~x && ~x <= d`). We also require that the actual ranges always include the real ones ($[a,b] \subseteq [c,d]$), hence it is the actual ranges ($[c,d]$) that are used for the computation of K and σ.

Iteration-dependent Ranges For many loops however, inductive constant ranges either do not exist or are very hard to prove. Or it may be that only the real-valued ranges are inductive, but, because of round-off errors, the actual ones are not. We still want to analyze these loops, but it is clear that the validity of the computed K, σ and final errors is limited to the validity of the ranges which we use for their computation.
Our tool attempts to verify that the range bounds are valid, i.e. inductive. Should it not succeed, it will nonetheless perform the error computation with the user-given actual ranges. The generated code will, however, include the precondition `require(c <= x && x <= d)`, which in Scala is checked at runtime. We believe that this is a reasonable compromise, as these assertions are fast to check and in many applications ranges do stay bounded.

7.2.2 Truncation Errors

What if round-off errors are not the only errors? If the real-valued computation given by the specification is also the *ideal* computation, we can simply add the errors in the same way as round-off errors. If the real-valued computation is, however, already an approximation of some other *unknown* ideal function, say f_*, it is not directly clear how our error computation applies. This may be the case, for example, for truncation errors. Let us suppose that we can compute (or at least overestimate) these by a function $\tau : \mathbb{R}^n \to \mathbb{R}^n$, i.e. $\tau_{f_*}(x) = |f_*(x) - f(x)|$.

In the following we consider the one-dimensional case $n = 1$ for simplicity of exposition, but it generalizes as before to the n-dimensional case. We can apply a similar separation of errors as before:

$$\begin{aligned}
&|f_*(x) - \tilde{f}(\tilde{x})| \\
&\qquad \leq |f_*(x) - f(x)| + |f(x) - f(\tilde{x})| + |f(\tilde{x}) - \tilde{f}(\tilde{x})| \\
&\qquad = \tau(x) + g(|x - \tilde{x}|) + \sigma(\tilde{x})
\end{aligned}$$

which lets us decompose the overall error into the truncation error, the propagated initial error and the round-off error. If we now iterate, we find by a similar argument as before:

$$\begin{aligned}
&|f_*^m(x) - \tilde{f}(\tilde{x})| \\
&\leq g^m(|x-\tilde{x}|) + \sum_{j=0}^{m-1} g^j\left(\tau(f_*^{m-j-1}(x))\right) + g^j\left(\sigma(\tilde{f}^{m-j-1}(\tilde{x}))\right) \\
&= g^m(|x-\tilde{x}|) + \sum_{j=0}^{m-1} g^j\left(\tau(f_*^{m-j-1}(x)) + \sigma(\tilde{f}^{m-j-1}(\tilde{x}))\right)
\end{aligned}$$

The result essentially means that our previously defined method can also be applied to the case when truncation (or similar) errors are present. We do not pursue this direction further however, and leave a proper automated treatment of truncation errors to future work.

7.2.3 Experimental Results

We have implemented our proposed technique inside Rosa and evaluate it for the case of loops on a number of examples which demonstrate several features of our system.

Newton-Raphson Method We begin with an example of a loop in which we wish to show that the value of a variable always remains in a certain range. Such a property is important, for instance, in the case of iterative algorithms and controllers, where unboundedness suggests that the system diverges. The following function taken from [56] implements a Newton-Raphson approximation. (We abbreviate e.g. `x*x*x` as x^3 for readability.)

```
def newton(x: Real, k: LoopCounter): Real = {
  require(-1.0 < x && x < 1.0 && -1.0 < ~x && ~x < 1.0)
  if (k < 10)
    newton(x - (x - (x^3)/6.0 + (x^5)/120.0 + (x^7)/5040.0) /
      (1 - (x*x)/2.0 + (x^4)/24.0 + (x^5)/720.0), k + 1)
  else
    x
} ensuring(res => -1.0 < res && res < 1.0 && -1.0 < ~res && ~res < 1.0)
```

Note that the precondition is also the loop invariant we wish to check. Our tool can automatically verify that this specification is inductive, also in the presence of round-off errors. Fluctuat, relying on affine arithmetic cannot prove that even one iteration remains in the given bound, and applying it to an unbounded loop produces ∞ as the error bound. Our result holds for any number of iterations and does not rely on unrolling. If we were to change the range specification to `-1.2 < x && x < 1.2 && -1.2 < ~x && ~x < 1.2`, the verification (correctly) fails and our tool reports a counter-example. Thus, our system can be used to establish boundedness of values in loops even if the number of iterations is unknown at compile time.

```
@model def nextItem: Real = {
  ????
} ensuring (res => -1200 <= res && res <= 1200 && res +/- 1e-8)

def mean(n: Int, m: Real): Real = {
  require(-1200 <= m && m <= 1200 && 2 <= n && n <= 1002 &&
        -1200.5 <= ~m && ~m <= 1200.5)
  if (n < 102) {
    val x = nextItem
    val m_new = ((n - 1.0) * m + x) / n
    mean(n + 1, m_new)
  } else
    m
} ensuring (res => -1200.00 <= res && res <= 1200.00)
```

Figure 7.5 – Running average computation

Running Average Now we return to numerical error estimation. Figure 7.5 shows the implementation of an online computation of the average of numbers coming from the range $[-1200, 1200]$. The method `nextItem` models a fresh value from an undetermined source. Round- off errors allow the computed value to go outside of $[-1200, 1200]$, we thus use the actual range $[-1200.5, 1200.5]$ for the error computation. We chose this range optimistically to cover a substantial number of iterations.

Table 7.3 compares the errors and runtimes computed by Rosa against the errors determined by Fluctuat. We compare with Fluctuat using complete unrolling of loops, because we have found that abstract interpretation with the domains in Fluctuat does not stabilize on these kinds of loops due to the unboundedly growing errors. In contrast, our system can discover parametric bounds that grow as a function of the loop iteration. In particular, the overall error is given as

$$K^m \lambda + ((I - K)^{-1}(I - K^m))\sigma$$

where Rosa determines $K = 0.999001996$, $\sigma = 1.5082\text{e-}10$ and λ=1.1369e-13 for example for the case of no added initial errors. If we try to unroll the loop within Rosa, it does not finish even for small numbers of iterations in reasonable time, hence we do not list it here.

In this example, the computation depends on the loop counter n, making the constraints different for different loop bounds and thus different bounds on n. This may make the constraint more or less hard to solve for Z3, which results in the, maybe surprising, variation in running times.

The comparison in Table 7.3 shows the trade-off our technique makes between accuracy and scalability. Recall that our proposed technique makes two approximations: firstly, it approximates the effect of each loop iteration in isolation, potentially loosing correlation

# iterations	Fluctuat	time	Rosa	time
without additional error				
100	1.522e-11	0.5	7.486e-10	20
500	7.742e-11	9	2.393e-8	28
1000	1.545e-10	35	9.544e-8	28
2000	3.085e-10	180	1.306e-7	19
3000	4.554e-10	425	1.436e-7	19
4000	6.166e-10	814	1.484e-7	19
with 1e-8 error				
100	9.916e-9	0.5	3.203e-7	20
500	1.006e-8	9	1.608e-6	28
1000	1.014e-8	39	3.260e-6	28
2000	1.030e-8	176	4.460e-6	19
3000	1.045e-8	423	4.903e-6	19
4000	1.062e-8	813	5.066e-6	19

Table 7.3 – Comparison of absolute errors and running times computed by Fluctuat and Rosa on the mean benchmark. Loop unrolling with Rosa times out. Time is measured in seconds.

information and secondly, the propagation coefficients are computed across the whole input range. While Fluctuat, performing loop unrolling and keeping correlations computes smaller error bounds, our tool is significantly faster and more scalable for a larger number of iterations (> 1000).

Pendulum Figure 7.6 shows a Runge Kutta order 2 simulation of a pendulum, where `t` and `w` are the angle the pendulum forms with the vertical and the angular velocity respectively. We approximate the sine function with its Taylor series polynomial, and consider two versions: the order 3 and order 7 Taylor approximation. In both cases we focus on round-off errors between the system following the dynamics given by the polynomial approximation, and the system following the same dynamics but implemented in finite precision.

The precondition now specifies two sets of ranges. `-2 <= t` constrains the *real-valued* variable t, whereas `-2.01 <= ~t` specifies that its actually implemented finite-precision counterpart remains in a slightly larger range. This latter interval includes all errors including round-offs and is therefore larger.

```
def sine(x: Real): Real = {
  require(-5 <= x && x <= 5)
  x - x*x*x/6 //+ x*x*x*x*x/120
}

def pendulum(t: Real, w: Real, n: LoopCounter): (Real,Real)={
  require(-2 <= t && t <= 2 && -5 <= w && w <= 5 &&
    -2.01 <= ~t && ~t <= 2.01 && -5.01 <= ~w && ~w <= 5.01)

  if (n < 1000) {
    val h: Real = 0.01
    val L: Real = 2.0
    val m: Real = 1.5
    val g: Real = 9.80665
    val k1t = w
    val k1w = -g/L * sine(t)
    val k2t = w + h/2*k1w
    val k2w = -g/L * sine(t + h/2*k1t)
    val tNew = t + h*k2t
    val wNew = w + h*k2w

    pendulum(tNew, wNew, n + 1)
  } else {
    (t, w)
  }
}
```

Figure 7.6 – Pendulum simulation with sine approximation

During the analysis, our tool determines the following propagation coefficient matrix *K* for the order 3 approximation:

$$\begin{array}{cc} 1.0002500818333124 & 0.01 \\ 0.05250059956010406 & 1.0002625029978005 \end{array}$$

and for the order 5 approximation:

$$\begin{array}{cc} 1.0000833441850905 & 0.01 \\ 0.04903325006220655 & 1.0000872967293692 \end{array}$$

Notice that the latter numbers are smaller, partly by an order of magnitude, which strongly suggests that this approximation is preferable with respect to numerical errors.

Table 7.4 compares computed error bounds by Fluctuat, again with loop unrolling, against the results obtained by our tool. Rosa's loop unrolling again fails to provide results in acceptable time. Fluctuat is able to compute error bounds for smaller number of iterations, although our bounds are more accurate nearly throughout. For larger numbers of iterations, Rosa is

# iter	Fluctuat	time	Our tool	time
order 3				
5	1.46e-15	0.04	1.47e-15	4
10	2.88e-15	0.13	2.87e-15	4
15	4.53e-15	0.39	4.44e-15	4
20	6.51e-15	0.89	6.19e-15	4
25	8.98e-15	2	8.15e-15	4
50	5.07e-14	16	2.21e-14	4
100	∞	-	9.07e-14	4
250	∞	-	3.11e-12	4
500	∞	-	9.58e-10	4
1000	∞	-	9.02e-5	4
order 5				
5	1.47e-15	0.06	1.47e-15	8
10	2.92e-15	0.35	2.88e-15	8
15	4.68e-15	1	4.45e-15	8
20	6.95e-15	3	6.21e-15	8
25	1.01e-14	5	8.18e-15	8
50	2.43e-13	49	2.21e-14	8
100	∞	-	8.82e-14	8
250	∞	-	2.67e-12	8
500	∞	-	6.54e-10	8
1000	∞	-	3.89e-5	8

Table 7.4 – Comparison of absolute errors on t and running times computed by Fluctuat and our tool on the pendulum benchmark. Time is measured in seconds.

the only one that can still compute meaningful error bounds. Also note, that while Fluctuat's computation time grows with the number of iterations, our time is constant, since the propagation coefficient matrix is the same for all iterations in this case, and using our closed-form expression for the errors, the final loop bound computation is fast.

Gravity Simulation Figure 7.7 shows the code of our Jupiter simulation, which we adapted from the `nbody` benchmark from [6]. We have abbreviated `-6 <= x && x <= 6` by `x∈[-6,6]` for readability. Note that the precondition includes the clause `x*x + y*y + z*z >= 20.0` imposing a minimum distance between planets, which ensures that taking the square root is safe, even in the presence of numerical errors, and that dividing by the results is similarly

```
def step(x:Real, y:Real, z:Real, vx:Real, vy:Real, vz:Real,
  i:LoopCounter): (Real, Real, Real, Real, Real, Real) = {
  require(x ∈ [-6, 6] && y ∈ [-6, 6] && z ∈ [-0.2, 0.2] &&
     vx ∈ [-3, 3] && vy ∈ [-3, 3] && vz ∈ [-0.1, 0.1] &&
     x*x + y*y + z*z >= 20.0 &&
     ~x ∈ [-6, 6] && ~y ∈ [-6, 6] && ~z ∈ [-0.2, 0.2] &&
     ~vx ∈ [-3, 3] && ~vy ∈ [-3, 3] && ~vz ∈ [-0.1, 0.1]
      (~x)*(~x) + (~y)*(~y)+ (~z)*(~z) >= 20.0)

  if (i < 100) {
    val dt = 0.1
    val solarMass = 39.47841760435743
    val distance = sqrt(x*x + y*y + z*z)
    val mag = dt / (distance * distance * distance)

    val vxNew = vx - x * solarMass * mag
    val vyNew = vy - y * solarMass * mag
    val vzNew = vz - z * solarMass * mag
    val x1 = x + dt * vxNew
    val y1 = y + dt * vyNew
    val z1 = z + dt * vzNew
    step(x1, y1, z1, vxNew, vyNew, vzNew, i + 1)
  } else {
    (x, y, z, vx, vy, vz)
  }
}
```

Figure 7.7 – Planet orbiting the Sun simulation

well behaved. At the same time, this constraint also makes the analysis of the loop errors challenging, as does the presence of square root and a large number of variables. The plot from the introduction (Figure 1.2) reports the errors of this benchmark for one specific initial configuration (corresponding to the position of Jupiter). The analysis done by our tool in this example covers *all* configurations where the planet's position and velocity coordinates satisfy the constraints in the precondition. The constraint `x*x + y*y + z*z ≥ 20.0` also limits the possible orbits we want to do the analysis for. The following table compares the final errors computed for the given constraint and for the case when we relax the condition to `x*x + y*y + z*z ≥ 15.0` for 100 iterations. As the number of possible cases our tool has to consider is larger, we also expect the over-approximation committed during the analysis to grow.

$x^2 + y^2 + z^2$	≥ 15	≥ 20
x	8.473e-07	1.894e-08
y	1.174e-07	2.087e-09
z	9.190e-07	2.635e-08

As the table shows, the over-approximation is modest and our tool still computes meaningful results. To compare to the simulation from the introduction, the errors on `x, y, z` after 100 iterations were on the order of `1e-14`. These errors are however, obtained for one specific run, whereas our tool performs a worst-case analysis for a large number of runs at the same time. To our knowledge, our tool is the only one that can handle programs of this complexity. Fluctuat returns with an error bound of $[-\infty, \infty]$, and Rosa again does not finish unrolling in a reasonable time.

Conclusion

We have presented techniques for computing the errors due to discontinuities and we have derived a closed-form expression for numerical errors in unbounded loops. Both techniques rely crucially on a separation of errors and a range computation which can take into account additional constraints on variables. Our experimental results show that our approach substantially improves over the current state-of-the-art.

8 Certifying Results of Iterative Algorithms

Much of numerical software uses iterative algorithms to solve for example systems of nonlinear equations and many are available as highly optimized black-box functions in (commercial) scientific software packages such as MATLAB [113], Octave [3] or Mathematica [5]. Often the source code is not available and the functionality and its mathematical meaning are not well documented, making it hard to verify the computed results.

Many iterative algorithms are self-stabilizing in that individual iterations are independent of each other and errors from one iteration get corrected in subsequent ones [95]. It thus does not make much sense to track round-off errors throughout the entire computation. Iterative algorithms however suffer from truncation errors: the exact result can be obtained only in the limit and at some point the iteration has to be stopped. Our runtime solution aims at quantifying these errors rigorously by using theorems from validated numerics. We integrate the error computation with Scala macros which perform a part of the computation already at compile time, yielding a sound and efficient method to certify solutions of systems of nonlinear equations.

This chapter is based on the paper [45]. Source code for our library, called Cassia, as well as all examples are available from `github.com/malyzajko/cassia`.

8.1 Introduction

To understand the notion of method errors we address, consider an iterative method that performs a search for the solution of $f(x) = 0$ by computing a sequence of approximations $x_0, x_1, x_2, \ldots$ One common stopping criterion for an iteration is finding x_k for which $|f(x_k)| < \varepsilon$, for a given error tolerance ε. From a validation point of view, however, we are ultimately interested not in ε but in τ such that $|x - x_k| < \tau$, where x is the actual solution in real numbers. Fortunately, we can estimate τ from ε using a bound on the derivative of f in an interval conservatively enclosing x and x_k.

A tempting approach is to perform the entire computation of x_k using interval or affine arithmetic. However, this approach would be inefficient, and would give too pessimistic error bounds. Instead, our method uses a runtime checking approach. We allow any standard non-validated floating point code to compute the approximation x_k. We perform only the final validation of an individual candidate solution x_k using a range-based computation. In this way we achieve efficiency and reusability of existing numerical routines, while still providing rigorous bounds on the total error. The bounds certified by our system are always sound for the given execution. Our system thus realizes a new kind of assertion, appropriate for numerical computation: an assertion that verifies "this was precise enough" in a way that takes into account both the numerical problem and floating point semantics.

8.2 Examples

We illustrate our techniques with several examples that model physical processes, taken from [151, 43, 133]. These examples illustrate the applicability of our techniques and introduce the main features of our runtime library. For space reasons we abbreviate the Scala `Double` type with `D` (the code snippets remain valid Scala code using the rename-on-import Scala feature). We include variable type declarations for expository purposes, even though the Scala compiler can infer all but the function parameter types. Method names printed in bold are part of our library.

Stress on a turbine rotor We illustrate the basic features of our library on the following system of three non-linear equations with three unknowns (v, ω, r). An engineer may need to solve such a system to compute the stress on a turbine rotor [151].

$$3+\frac{2}{r^2}-\frac{1}{8}\frac{(3-2v)}{1-v}\omega^2 r^2 = 4.5$$

$$6v-\frac{1}{2}\frac{v}{1-v}\omega^2 r^2 = 2.5$$

$$3+\frac{2}{r^2}-\frac{1}{8}\frac{(1+2v)}{1-v}\omega^2 r^2 = 0.5$$

Given some (blackbox) library function `computeRoot` and our library for certifying solutions, the engineer can write the following code:

```
val f1 = (v: D,w: D,r: D) ⇒ 3 + 2/(r*r) - 0.125*(3-2*v)*(w*w*r*r)/(1-v)-4.5
val f2 = (v: D,w: D,r: D) ⇒ 6*v - 0.5 * v * (w*w*r*r) / (1-v)-2.5
val f3 = (v: D,w: D,r: D) ⇒ 3 - 2/(r*r) - 0.125*(1+2*v)*(w*w*r*r) / (1-v)-0.5
val x0 = Array(0.75, 0.5, 0.5)
val roots: Array[D] = computeRoot(Array(f1,f2,f3), jacobian(f1,f2,f3), x0, 1e-8)
val err:Array[Interval] = assertBound(f1,f2,f3, roots(0),roots(1),roots(2),1e-8)
```

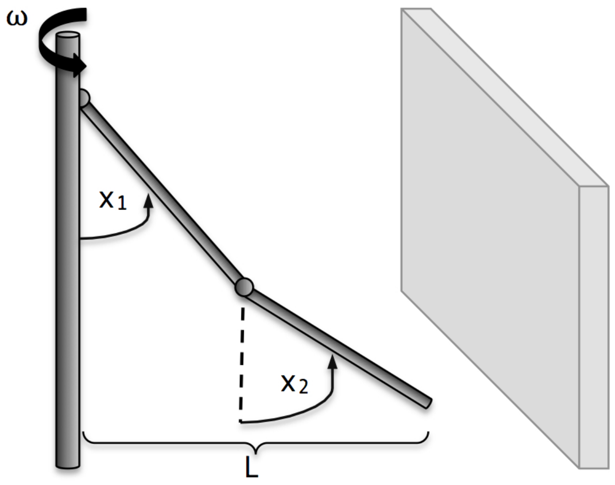


Figure 8.1 – A double pendulum standing close to an obstacle

The method **`assertBound`** takes as input the three functions of our system of equations, the previously computed roots and a tolerance and returns sound bounds on the true errors on the roots. In the case where these errors are larger than the tolerance specified, the method throws an exception and acts like an assertion. Our library also includes the method **`jacobian`**, which computes the Jacobian matrix of the functions f_1, f_2 and f_3 symbolically at compile time.

The true roots for v, w and r are 0.5, 1.0 and 1.0 respectively, and the roots and maximum absolute errors computed by the above code are

```
0.5, 1.0000000000018743, 0.9999999999970013
2.3684981521893e-15, 1.8806808806556e-12, 3.0005349681420e-12
```

Note that the error bounds that were computed are, in fact, smaller than the tolerance given to the numerical method used to compute the root.

Double pendulum The following example demonstrates how our library fits into a runtime assertion framework. A double pendulum rotates with angular velocity ω around a vertical axis (like a centrifugal regulator)[43]. At equilibrium the two pendulums make the angles x_1 and x_2 to the vertical axis. It can be shown that the angles are determined by the equations

$$\tan x_1 - k(2\sin x_1 + \sin x_2) = 0$$
$$\tan x_2 - 2k(\sin x_1 + \sin x_2) = 0$$

where k depends on ω, the lengths of the rods and gravity. Suppose the pendulum is standing close to a wall (as in Figure 8.1) and we would like to verify that in the equilibrium position it cannot hit the wall. Also suppose that the distance to the center of the pendulum is given by a function `distancePendulumWall`. Then the following code fragment verifies that a collision is impossible in the real world, not just in a world with floating-points.

```
val distancePendulumWall : SmartFloat = ...
val length = ... //length of bars
val tolerance = 1e-13; val x0 = Array(0.18, 0.25)
val f1 = (x1: D, x2: D) ⇒ tan(x1) - k * (2*sin(x1) + sin(x2))
val f2 = (x1: D, x2: D) ⇒ tan(x2) - 2*k * (sin(x1) + sin(x2))
val r: Array[D] = computeRoot(Array(f1,f2), jacobian(f1,f2), x0, tolerance)
val roots: Array[SmartFloat] = certify(r, errorBound(f1, f2, r(0), r(1), tolerance))

val L: SmartFloat = _sin(roots(0)) * length + _sin(roots(1)) * length
if (certainly(L <= distancePendulumWall)) {
  // continue computation
} else {
  // reduce speed of the pendulum and repeat
}
```

To account for all sources of uncertainty, we use the `SmartFloat` data type from chapter 3. `SmartFloat` performs a floating point computation while additionally keeping track of different sources of errors, including floating point round-off errors, as well as errors arising from other sources, for example, due to the approximate nature of physical measurements.

In our example, `distancePendulumWall` and **`certify`** both return a `SmartFloat`; the first one captures the uncertainty on a physical quantity, and the second one the method error due to the approximate iterative method. If the comparison in line 9 succeeds, we can be sure the pendulum does not touch the wall. This guarantee takes into account round-off errors committed during the calculation, as well as the error committed by the `computeRoot` method and their propagation throughout the computation.

State equation of a gas Values of parameters may only be known within certain bounds but not exactly, for instance if we take inputs from measurements. Our library provides guarantees even in the presence of such uncertainties. Equation 8.1 below relates the volume V of a gas to the temperature T and the pressure p, given parameters a and b that depend on the specifics of the gas, N the number of molecules in the volume V and k the Boltzmann constant [133].

$$[p + a(N/V)^2](V - Nb) = kNT \tag{8.1}$$

If T and p are given, one can solve the nonlinear Equation 8.1 to determine the volume occupied by the (very low-pressure) gas. Note however, that this is a cubic equation, for which closed-form solutions are non-trivial, and their approximate computation may incur substantial round-off errors. Using an iterative method, whose result is verified by our library, is thus preferable:

```
val T = 300; val a = 0.401; val b = 42.7e-6;
val p = 3.5e7; val k = 1.3806503e-23; val x0 = 0.1
val N: Interval = 1000 +/- 5
val f = (V: D) ⇒ (p + a * (N.mid / V) * (N.mid / V)) * (V - N.mid * b)
   - k * N.mid * T
val V: D = computeRoot(f, derivative(f), x0, 1e-9)
val Vcert: SmartFloat = certify(V, assertBound(f, V, 0.0005))
```

We make the assumption that we cannot determine the number of molecules N exactly, but we are sure that our number is accurate at least to within ±5 molecules (line 3). We compute the root as if we knew N exactly, using the middle value of the interval and the standard Newton's method and only check a posteriori that the result is accurate up to $\pm 0.0005m^3$, for all N in the interval $[995, 1005]$. Our library will confirm this providing us also with the (certified) bounds on V: `[0.0424713, 0.0429287]`.

8.3 Certification Technique

Our certification technique is based on several theorems from the area of validated numerics. It can verify roots of a system of nonlinear equations computed by an arbitrary black-box solution or estimation method.

In the following, we denote computed approximate solutions by $\tilde{x}$ and true roots by x, as before. $\mathbb{IR}$ denotes the domain of intervals over the real numbers $\mathbb{R}$ and variables written in bold type, e.g. $\mathbf{X}$, denote interval quantities. For a function f, we define $f(\mathbf{X}) = \{f(x) \mid x \in \mathbf{X}\}$. All errors are given in absolute terms. Error tolerance, that is, the maximum acceptable value for $|\tilde{x} - x|$, will be denoted by τ or `tolerance` in code. We will use the term *range arithmetic* to mean either interval arithmetic or affine arithmetic. The material presented in this section is valid for any such arithmetic, as long as it computes guaranteed enclosures containing the result that would be computed in real numbers. We wish to compute a guaranteed bound on the error of a computed solution, that is, determine an upper bound on $\Delta x = \tilde{x} - x$. Note that Δx is different from τ, because Δ considers the sign of the difference.

8.3.1 Unary Case

For expository purposes, consider first the unary case $f : \mathbb{R} \to \mathbb{R}$, f differentiable, and suppose that we wish to solve the equation $f(x) = 0$. Then, by the Mean Value Theorem

$$f(\tilde{x}) = f(x + \Delta x) = f(x) + f'(\xi)\Delta x$$

where $\xi \in \mathbf{X}$ and $\mathbf{X}$ is a range around $\tilde{x}$ sufficiently large to include the true root. Since $f(x) = 0$,

$$\Delta x \in \frac{f(\tilde{x})}{f'(\mathbf{X})} \tag{8.2}$$

We now have set membership instead of equality because the right-hand side is now a range-valued expression, which takes into account the fact that ξ in the Mean Value Theorem is not known exactly. The following theorem (stated in the formulation from [137]) formalizes this idea.

Theorem 8.1 *Let a differentiable function $f : \mathbb{R} \to \mathbb{R}$, $\mathbf{X} = [x_1, x_2] \in \mathbb{IR}$ and $\tilde{x} \in \mathbf{X}$ be given, and suppose $0 \notin f'(\mathbf{X})$. Define*

$$N(\tilde{x},\mathbf{X}) := \tilde{x} - f(\tilde{x})/f'(\mathbf{X}). \tag{8.3}$$

If $N(\tilde{x},\mathbf{X}) \subseteq \mathbf{X}$, then $\mathbf{X}$ contains a unique root of f. If $N(\tilde{x},\mathbf{X}) \cap \mathbf{X} = \emptyset$, then $f(x) \neq 0$ for all $x \in \mathbf{X}$.

Claim 8.2 *If, following Equation 8.2, we compute an interval $\Delta\mathbf{x} = f(\tilde{x})/f'(\mathbf{X})$ enclosing the upper bound on the error Δx, and if $\Delta\mathbf{x} \subseteq [-\tau,\tau]$, then the approximately computed result $\tilde{x}$ is indeed within the specified precision τ.*

Indeed, choose $\mathbf{X} = [\tilde{x} - \tau, \tilde{x} + \tau]$, i.e. the computed approximate solution plus or minus the tolerance we want to check, and compute $\Delta\mathbf{x} = \frac{f(\tilde{x})}{f'(\mathbf{X})}$. Then the condition $N(\tilde{x},\mathbf{X}) \subseteq \mathbf{X}$ from Theorem 8.1 becomes

$$N(\tilde{x},\mathbf{X}) = \tilde{x} - \Delta\mathbf{x} \subseteq \mathbf{X} = [\tilde{x} - \tau, \tilde{x} + \tau]$$

If $\Delta\mathbf{x} \subseteq [-\tau,\tau]$, this condition holds, and thus the computed result is within the specified precision.

Our assertion library uses the procedure in Figure 8.2 for unary problems. Note that we not only check that errors are within a certain error tolerance, but we also return the computed error bounds. As we show in Section 8.5, the computed error bounds tend to be much tighter than the user-required tolerance. As Section 3 illustrates, this error bound can be used in subsequent computations to track overall errors more precisely.

```
def assertBound (Function, Derivative, xn, τ)
  X = [xn ± τ]
  error = Function(xn) / Derivative(X)
  if error ∩ [-τ, τ] = ∅ throw SolutionNotIncludedException
  if ¬(error ⊂ [-τ, τ]) throw SolutionCannotBeVerifiedException
  return error
```

Figure 8.2 – Algorithm for computing errors in the unary case

8.3.2 Multivariate Case

Our error estimates for the unary case follow from the Mean Value Theorem, which also extends to n dimensions. Theorem 8.3 follows the interval formulation of [137] where J_f is the Jacobian matrix of f. If $\mathbf{D} = (\mathbf{x_1}, \ldots, \mathbf{x_n}) \in \mathbb{IR}^n$, let $\bar{\mathbf{D}}$ denote $\mathbf{x_1} \times \ldots \times \mathbf{x_n}$. For $a, b \in \bar{\mathbf{D}}$, define the convex union as $a \underline{\cup} b = \{a + \lambda b \mid \lambda \in [0,1]\}$. For $A \subseteq \bar{\mathbf{D}}$, define $hull(A) := \bigcap\{\mathbf{Z} \in \mathbb{IR}^n \mid A \subseteq \mathbf{Z}\}$.

Theorem 8.3 *Let there be given a continuously differentiable $f : \bar{\mathbf{D}} \to \mathbb{R}^n$ with $\mathbf{D} \in \mathbb{IR}^n$ and $x, \tilde{x} \in \bar{\mathbf{D}}$. Then for $\mathbf{X} := hull(x \underline{\cup} \tilde{x})$*

$$f(x) \in f(\tilde{x}) + J_f(\mathbf{X})(x - \tilde{x})$$

We extend our method for computing the error on each root in a similar manner:

$$\delta \in J_f^{-1}(\mathbf{X}) \cdot (-f(\tilde{x})) \tag{8.4}$$

where $\delta = x - \tilde{x}$ is the vector of errors on our tentative solution. Since we now must consider the Jacobian of f instead of a single derivative function, we can no longer solve for the errors by a simple scalar division. We wish to find the maximum possible error, so we need a way to compute an upper bound on the right-hand side of Equation 8.4. Computing the inverse of a Jacobian matrix in range arithmetic typically does not yield a useful result, due to over-approximation. Instead, we use the following Theorem 8.4, which is originally due to [96], but we use the formulation by [137].

Theorem 8.4 ([137]) *Let $A, R \in \mathbb{R}^{n \times n}$, $b \in \mathbb{R}^n$ and $\mathbf{E} \in \mathbb{IR}^n$ be given, denote by I the identity matrix. Assume*

$$Rb + (I - RA)\mathbf{E} \subset int(\mathbf{E}). \tag{8.5}$$

where $int(\mathbf{E})$ denotes the interior of the set $\mathbf{E}$. Then the matrices A and R are non-singular and $A^{-1}b \in Rb + (I - RA)\mathbf{E}$.

We instantiate Theorem 8.4 with all possible matrices A such that $A \in J_f(\mathbf{X})$ and all possible vectors b such that $b \in -f(\tilde{x})$, where $J_f(\mathbf{X})$ and $-f(\tilde{x})$ are both evaluated in range arithmetic. Combining with Condition 8.4, we obtain

$$\delta \in J_f^{-1}(\mathbf{X}) * -f(\tilde{x}) \subseteq Rb + (I - RA)\mathbf{E}, \tag{8.6}$$

provided that Condition 8.5 is satisfied in range arithmetic.

Matrix R in Theorem 8.4 can be chosen arbitrarily as long as Condition 8.5 holds. A common choice is to use an approximate inverse of A. In our case, A is range-valued, so we first compute the matrix whose entries are the midpoints of the intervals of A, and use its inverse as R. It

```
def assertBound (functions, Jacobian, xn, $\tau$)
 Xn = [xn ± $\tau$]
 A = Jacobian(Xn)
 // goal is to certify that xn is a zero of 'functions' up to $\tau$
 b = - functions(xn)
 R = inverse(mid(A)) // calculated in ordinary floating points
 E = [0 ± $\tau$]
 errors = R*b + (I - RA)E // Theorem 8.4
 if errors ∩ [-$\tau$, $\tau$]$^n$ = $\varnothing^n$ throw SolutionNotIncludedException
 if ¬(errors ⊆ [-$\tau$, $\tau$]$^n$) throw SolutionCannotBeVerifiedException
 return errors
```

Figure 8.3 – Algorithm for computing errors in the multivariate case

now remains to determine **X**. We choose it to be the vector where the i^{th} entry is the interval around $\tilde{x}_i$ with width τ. If we can then show that Condition 8.6 holds, we have proven that **X** indeed contains a solution. Moreover, we have computed a tighter upper bound on the error. We obtain the procedure in Figure 8.3 for computing error bounds for systems of equations. The variables `Xn, A, b, E, errors` are all range valued.

Our approach requires the derivatives to be non-zero, respectively the Jacobian to be non-singular, in the neighborhood of the root. This means that at present we can only verify single roots. Verifying multiple roots is an ill-conditioned problem by itself, and thus requires further approximation techniques, as well as dealing with complex values. We leave this for future work. Our library does distinguish the cases when an error is provably too large from the case when our method is unable to ensure the result: we use two different exceptions for this purpose.

8.4 Implementation

Now that we have the theoretical building blocks the question is how to integrate it into a general-purpose programming language like Scala such that the resulting assertion framework for real numbers is intuitive to use but at the same time efficient. In particular, Algorithms 8.2 and 8.3 require the computation of derivatives and their evaluation in range arithmetic, but we do not want to require the user to provide two differently typed functions, one in `Doubles` for the solver and one in `Intervals` for our verification method. Also, the solver may not actually require derivatives or the Jacobian, hence this computation should be performed automatically and symbolically at compile time. As the verification is a dynamic one, we identify the Scala macro framework as an ideal candidate for the integration of our library.

8.4.1 Scala Macros

Scala version 2.10 introduced a macro facility [31]. To the user macros look like regular methods, but their code is executed at compile time and performs a transformation on the Scala compiler abstract syntax tree (AST). Thus, by passing a regular function to a macro, we can access its AST and perform the necessary transformations. The type checker runs after the macro expansion which means that the resulting code retains all guarantees from Scala's strong static typing. Our library provides the following functions

```
def errorBound(f: (Double ⇒ Double), x: Double, tol: Double): Interval
def assertBound(f: (Double ⇒ Double), x: Double, tol: Double): Interval
def certify(root: Double, error: Interval): SmartFloat
```

and similarly for functions of 2, 3 and more variables. The function **`assertBound`** computes the guaranteed bounds on the errors using Algorithms 1 and 2. **`errorBound`** removes the assertion check and only provides the computed error; the programmer is then free to define individual assertions. **`certify`** wraps the computed root(s) including their associated errors in the `SmartFloat` data type and hereby provides the link to our assertion checking framework. We also expose the automatic symbolic derivative computation facility:

```
def derivative(f: Double ⇒ Double): (Double ⇒ Double)
def jacobian(f1: (Double, Double) ⇒ Double, f2: (Double, Double) ⇒ Double):
   (Array[Array[(Double, Double) ⇒ Double]])
...
```

The functions passed to our macros have type `(Double*) => Double` and may be given as anonymous functions, or alternatively defined in the immediately enclosing method or class. The functions may use parameters, with the same restrictions on their original definitions. This is particularly attractive, as it allows us to write concise code as presented in the code snippets in section 8.2.

Integration With SmartFloats We combine the result certification with the `SmartFloat` runtime library. If no exceptions are thrown, the program would take the same path if real numbers were used instead of floating-points and the values computed are within the bounds computed by the `SmartFloat` data type. This assertion language thus tracks two sources of errors

- quantization errors due to the discrete floating-point number representation
- method errors due to the approximate numerical method

The bounds on computed values are ensured by using `SmartFloats` throughout the straight-line computations. Note that the numerical methods for computing the roots can still use only `Doubles` since we verify the result a posteriori.

8.4.2 Computing Derivatives

We now turn our attention to efficiency. Given the function ASTs, we compute the derivatives or Jacobian matrices already at compile time, and thus need to do this symbolically. The straightforward runtime option is to use automatic differentiation [76]. We will show however that this incurs an unnecessary computation cost. It turns out that fairly simple optimizations on top of the usual derivative rules already provide the needed precision and efficiency:

- constants are pulled outside of multiplications (before derivation)
- multiplications of the same terms are compacted into a power function (before derivation)
- multiplication and addition of zeros or ones arising from the differentiation are simplified (after derivation)
- powers with integers are evaluated by repeated multiplication (at runtime)

Overall, the effect is that the resulting expressions of derivatives do not blow up. This is important for evaluation efficiency, since each operation carries a computation cost (see Table 8.3). On the other hand, precision may be affected as well, since the over-approximation committed by range arithmetic may depend on the formulation of the expression. We have compared the errors computed with our symbolic differentiation routine against the results obtained with manually provided derivatives. The latter have the format one would compute by hand on paper. We did the comparison on our unary benchmark problems (Table 8.1), and it turns out that except for two instances, the errors computed are exactly the same. For the two other functions, our manual derivatives actually compute an error that is worse, but the precision is still sufficient to prove solutions are correct to within the given tolerance.

8.4.3 Uncertain Parameters

Theorem 8.4 also holds for range-valued A and b. It is thus natural to extend our macro functions to also accept range-valued parameters. The `SmartFloat` data type already has the facility to keep track of manually user-added errors so we can track external uncertainties as a third source of errors. Consider again the gas state equation example from Section 8.2, especially the following two lines:

```
val N = 1000 +/− 5
val f = (V: D) ⇒ (p + a ∗ (N.mid / V) ∗ (N.mid / V)) ∗ (V - N.mid ∗ b)
   - k ∗ N.mid ∗ T
```

The `+/-` method, in this context, returns an `Interval`, which in turn defines the `mid` method. Thus, the function type checks correctly and can be passed for example to a solver, but inside the macro we can use the interval version of the parameter.

8.5 Evaluation

Accuracy The theorems from Section 8.3 provide us with sound guarantees regarding upper bounds. In practice however, we also need our method to be precise. Since our library computes error bounds and not only binary answers for assertions, we are interested in obtaining as precise error estimates as possible. We have evaluated the precision of our approach in the following way. We compute a high-precision estimate of the root(s) using the QuadDouble library [16], which allows us to compute the true error on the computed solutions with high confidence. We compare this error to the one provided by our library. The results on a number of benchmark problems chosen from numerical analysis textbooks are presented in Tables 8.1 and 8.2. We are able to confirm the error bounds specified by the user in all cases. In fact, on all examples that we tried, our library only failed in the case of a multiple root for the reasons explained in Section 8.3 and never for precision reasons.

We split the evaluation between the unary case and the multivariate case because of their different characteristics. All numbers are the maximum absolute errors computed. The numbers in parentheses are the tolerances given to the solvers and have been chosen randomly to simulate the different demands of the real world. We highlight the better error estimates in bold.

First of all we note that the precision of the error estimates we obtain is remarkably good. Another perhaps surprising result of our experiments is that using interval arithmetic is generally more precise (in the unary case) or not much worse (in the multivariate case) than affine arithmetic, although the latter is usually presented as the superior approach. Indeed, for the tracking of round-off errors we have shown affine arithmetic to provide (sometimes much) better results that interval arithmetic in chapter 3. The reason why intervals perform as well is that for transcendental functions they are able to compute a tighter range, since affine arithmetic has to compute a linear approximation of those functions. The exceptions in the unary case are the degree 6 polynomial and the carbon gas state equation example, which confirms our hypothesis, since in that case the dependency tracking of affine arithmetic can recover some of the imprecision in the long run.

For the multivariate case, affine arithmetic performs generally better because the computation consists to a large part of linear arithmetic. Due to the larger computation cost, however, we leave it as a choice for the user which arithmetic to use and select interval arithmetic as a default.

8.5.1 Performance

Table 8.3 compares the performance of our implementation when using affine, interval arithmetic, or interval arithmetic without the differentiation optimizations listed in Section 8.4.2. Switching off the optimizations is similar to performing automatic differentiation. We can see that our optimizations actually make a big difference in the runtimes, improving by up to 37%

Problem (tolerance specified)	certified (affine)	certified (interval)	true errors
system of rods (1e-10)	7.315e-13	**1.447e-13**	1.435e-13
Verhulst model (1-e9)	4.891e-10	**9.783e-11**	9.782e-11
predator-prey model (1e-10)	7.150e-11	**7.147e-11**	7.146e-11
carbon gas state equation (1e-12)	**1.422e-17**	2.082e-17	1.625e-26
Butler-Volmer equation (1e-10)	4.608e-15	**3.8960e-15**	3.768e-17
$(x/2)^2 - sin(x)$ (1e-10)	7.4e-16	**5.879e-16**	1.297e-16
$e^x(x-1) - e^{-x}(x+1)$ (1e-8)	5.000e-10	5.000e-10	5.000e-10
degree 3 polynomial (1e-7)	7.204e-9	**1.441e-9**	1.441e-9
degree 6 polynomial (1e-5)	**2.741e-14**	3.538e-14	2.258e-14

Table 8.1 – Comparison of errors for unary benchmarks. All numbers are rounded.

Problem (tolerance specified)	certified (affine)	certified (interval)	true errors
stress distribution (1e-10)	3.584e-11 4.147e-11	3.584e-11 4.147e-11	3.584e-11, 4.147e-11
sin-cosine system (1e-7)	6.689e-9 6.655e-9	6.689e-9 6.655e-9	6.689e-9 6.6545e-9
double pendulum (1e-13)	**4.661e-15** **6.409e-15**	5.454e-15 7.449e-15	5.617e-17 9.927e-17
circle-parabola intersection (1e-13)	**5.5510e-17** 1.110e-16	1.110e-16 1.110e-16	8.0145e-51 5.373e-17
quadratic 2d system (1e-6)	**2.570e-12** 3.025e-09	3.326e-12 3.025e-9	2.192e-12 3.024e-9
turbine rotor (1e-12)	**1.517e-13** **1.707e-13** **1.908e-14**	1.523e-13 1.724e-13 1.955e-14	1.514e-13 1.703e-13 1.887e-14
quadratic 3d system (1e-10)	**4.314e-16** **5.997e-16** **4.349e-16**	6.795e-16 1.632e-15 5.127e-16	1.2134e-16 7.914e-17 7.441e-17

Table 8.2 – Comparison of errors for multivariate benchmarks. All numbers are rounded.

Problem set	affine	interval	interval w/o optimizations	quadruple precision
unary problems	2.170ms	0.459ms	0.733ms	17.196ms
2D problems	2.779ms	0.984ms	1.240ms	4.446ms
3D problems	3.563ms	1.063ms	1.515ms	16.605ms

Table 8.3 – Average runtimes for of the benchmark problems from Tables 8.1 and 8.2. Averages are taken over 1000 runs.

Problem	affine	interval
carbon gas state equation	0.272ms	0.084ms
double pendulum problem	0.784ms	0.228ms
turbine problem	2.643ms	0.644ms
degree 3 polynomial	0.116ms	0.044ms
quadratic 2d system	0.425ms	0.200ms
quadratic 3d system	0.943ms	0.460ms

Table 8.4 – Runtimes for individual problems. Averages are taken over 1000 runs.

for unary functions and 30% for our 3D problems over pure differentiation. On the other hand, the table clearly shows that affine arithmetic is much less efficient than interval arithmetic (factor 3-4.5 approx.), so should only be used if precision is of big importance.

We have also included the runtimes of re-computing the root(s) in quadruple precision. That is we have used approximately 64 decimal digits for all calculations of the numerical method. The runtimes illustrate that this approach for computing trustworthy results is clearly unsuitable from the performance point of view, and would not actually provide any guarantees on errors either, merely more confidence.

Table 8.4 illustrates the dependence of runtimes on the complexity of the problems. The first three problems are those from our examples in section 8.2 and the second set is comprised of relatively short polynomial equations. Clearly, runtimes depend both on the type of equations, transcendental functions being more expensive, as well as on the size of the system of equations. It should be noted however, that the increases are clearly appropriate given the increase of complexity of the problems.

8.5.2 Application to Optimization

The presented techniques are also applicable to verifying whether solutions to an optimization problem are close enough to the true local solution. We illustrate this with a case study where the goal is to estimate the abstract state of a power system through physical measurements. Today's power grids are more and more distributed, and instead of a few power plants that inject energy into the system, many small producers, such as home owners with solar panels, contribute as well. This makes it challenging to provide a sufficient but only necessary amount of power in the grid. It is thus necessary to always have a up-to-date measurements of the current state.

In our case study, in collaboration with the Computer Communications and Applications Laboratory 2 (LCA2) at EPFL [38], the system state is given by a vector x_i, with $i = 1\ldots5$, and cannot be measured directly. Instead, it is possible to derive it from noisy physical measurements in the network. The method used in our example for computing the most likely state given the measurements is weighted least-squares. Instead of describing the problem-specific details, we give here the final equation which is to be minized:

$$
\begin{aligned}
g(x_1,x_2,x_3,x_4,x_5) = {} & \frac{1}{\epsilon_1}(z_1-x_1)^2 + \frac{1}{\epsilon_2}(z_2-x_2)^2 + \frac{1}{\epsilon_3}(z_3-x_3)^2 + \frac{1}{\epsilon_4}(z_4-x_4)^2 + \frac{1}{\epsilon_5}(z_5-x_5)^2 \\
& + \frac{1}{\epsilon_6}\left(z_6 - A_{13}x_4x_1\cos(B_{13}-x_5) - A_{23}x_4x_2\cos(B_{23}+x_3-x_5) - A_{33}x_4^2\cos(B_{33})\right)^2 \\
& + \frac{1}{\epsilon_7}\left(z_7 + A_{13}x_4x_1\sin(B_{13}-x_5) + A_{23}x_4x_2\sin(B_{23}+x_3-x_5) + A_{33}x_4^2\sin(B_{33})\right)^2
\end{aligned}
$$

$z_1,...,z_7$ are measurements and $\epsilon_1,...\epsilon_5 = 10^{-10}$ and $\epsilon_6,\epsilon_7 = 10^{-8}$ are the weights and capture the noise on the measurements. The constant matrices A, B are given for each problem and are determined by the network. Then, given a set of measurements and a solution to the minimization problem computed from MATLAB, we want to check that it is indeed a solution.

Our framework as presented in this chapter is not applicable as-is for verifying optimization problems. However, if we want to optimize a function f then any (local) optimum y has to satisfy $f'(y) = 0$. This problem fits nicely the capabilities of our system. For this example, we have computed the partial derivative of g by hand, but our tool can be in principle straightforwardly extended to handle certification of optimization problems fully automatically. We can then check the MATLAB-computed solution by calling our library:

```
val error = errorBound(List(dg1, dg2, dg3, dg4, dg5), computedState, tolerance)
```

where `dg1, ..., dg5` are the partial derivatives of g. The error tolerance here is $10e-8$. For example, given the following set of measurements and the computed state:

```
val z1 = 1.000000000110817; val z2 = 0.998647460735060
val z3 = -0.001756247355916; val z4 = 0.998479207559061
val z5 = -0.001992728994125; val z6 = -0.017997984487926
val z7 = -0.005257996439913
val computedState = Array(1.000000000110817, 0.998647460656304,
                   -0.001756247550707, 0.998479207637711,
                   -0.001992728799334)
```

our tool can certify that the computed state is indeed a solution to the optimization problem within the tolerance and can narrow the errors on x even further:

error on	
x1	[-4.4408921429095500e-16,4.4408921429095500e-16]
x2	[-2.9093174937193904e-13,2.9103233737160703e-13]
x3	[-2.7284766119102693e-13,2.7206444647703290e-13]
x4	[-2.9142440391968894e-13,2.9231758426003640e-13]
x5	[-2.7206423473908830e-13,2.7284788313395123e-13]

We believe that this shows that our technique is widely applicable and can be quite useful when integrated with systems like MATLAB.

Conclusion

In this chapter we have shown how theorems from validated numerics can be adapted and integrated into a programming language for certifying solutions of nonlinear equations. Furthermore, by recognizing that self-correcting iterative methods require a different verification approach than forward computations, we can provide a better adapted technique. Finally, this chapter also shows another integration strategy for verification methods into a programming language.

9 Related Work

In this chapter we would like to provide a representative (but certainly only partial overview) of alternative approaches for the analysis, verification and synthesis of numerical programs. Many of these have different goals than ours and we view them as complementary to our work.

9.1 Applicability and Portability

Our techniques and tools are implemented in Scala and for the analysis of Scala programs, but are directly applicable and straight-forwardly portable to other programming languages which support the IEEE 754 floating-point standard. Scala was chosen in part for its flexibility in both writing application programs as well as the analysis tools themselves. That said, we do not fundamentally rely on any feature that is not also available in most other programming languages. In particular, the back-end of our tool Rosa can easily be changed to generate code in another programming language, which is then compiled further by the appropriate compiler.

Most programming languages support the IEEE 754 standard to the (limited) extent that our analyses rely upon, including the more commonly used languages for numerical programs C and Fortran. Our techniques further rely on compilers not re-arranging numerical computations, as this changes the roundoff error accumulation and propagation. This behavior can, in general, be switched off via compiler flags.

9.2 Sound Finite-Precision Computations

A substantial amount of research has gone into proving two types of properties: absence of runtime errors such as division-by-zero exceptions, as well as more expressive functional properties for finite-precision code. Much of the work in this section reduces to or is concerned with determining sound enclosures of mathematical expressions and is thus a prerequisite for an error analysis. We will review work that explicitly quantifies the difference between the ideal real-valued computation and its finite- precision implementation in the next section.

Affine Arithmetic and Other Range Bounding Techniques

We have decided to use affine arithmetic because it seems to us to be a good compromise between complexity and functionality. Our implementation of affine arithmetic can also be used independently from the round-off error computation. Alternative implementations include for example [148] which is the library from the original authors of affine arithmetic [52] and [85]. The latter uses the Chebyshev approximation and is implemented in double floating-point precision, which we both found to be insufficient for our purpose. It further does not appear to use directed rounding, hence it is not clear if the results are entirely sound. Solutions for the over-approximations committed by affine arithmetic have also been proposed. Zhang et al. [152] for example, reduce over-approximations due to multiplication by repeatedly refining the approximation until a desired accuracy is obtained. While the method provides a possible solution to the over-approximation problem, it is also computationally expensive and only applicable to multiplication. Another improvement over standard interval arithmetic is generalized interval arithmetic [79] which is essentially an affine form but with interval valued coefficients. The alternative implementations presented here are used to compute ranges only, that is, as an alternative to interval arithmetic and not for numerical error computations.

Affine arithmetic and similar range computation techniques are often used in validated computations, that is in computations where a sound enclosure of a function is required, given possibly interval valued inputs. For example, other range-based methods are surveyed in [112] in the context of plotting curves. The authors conclude that affine arithmetic has similar accuracy as the other top rated methods. Shou et al. [147] extend affine arithmetic to keep more correlations during multiplications, but the method is specific and only works for up to three variables. Ershov et al. [59] present an interval library where elementary mathematical functions are approximated by Chebyshev and Taylor series and may be a possible alternative for our nonlinear approximations.
Approaches like interval and affine arithmetic are also often used together with a subdivision scheme, which is also supported by Fluctuat [72]. By using an SMT solver which can capture arbitrary correlations (within its supported theory), we would like to avoid subdivisions as they inefficient, especially for larger numbers of variables.

Abstract Interpretation

Abstract interpretation [42] is an approach for computing sound enclosures, which is mostly used in the context of verifying error freedom of (numerical) programs. Whereas validated techniques as discussed previously are specialized for complex numerical computations, abstract interpretation focuses more on handling control flow including branches and loops.

The only work in this area that we are aware of that can also quantify round-off errors is Fluctuat, which we have already reviewed throughout this thesis.

Several abstract domains for finite-precision computations exist which are sound with respect to floating-point computations, that is they compute bounds on the ranges of variables which can then be used to prove the absence of runtime exceptions [41, 36, 89, 117, 61]. They have been successfully used in the verification of safety-critical software [22].
One difference to our work is that we do not define join and meet operations or widening. While we do support an analysis with different paths that performs merging after conditionals, this operation is simple as we only need to compute the convex union of intervals and loops are handled by inductive reasoning, as our focus is on the numerical errors.

Interval Constraint Solving

Duracz and Konecny [57] present a framework and a solver that can prove tight functional properties about floating-point numerical functions. The system generates verification conditions which are similar to the ones which we described in subsection 5.2.1. The verification task is reduced to an interval satisfaction problem, for which they present polynomial intervals, which are polynomial lower and upper bounds enclosing the function one wants to approximate. This allows them to automatically prove precise constraints. Makino and Berz [109] present an alternative arithmetic based on Taylor Models for solving such constraints. It has been so far successfully used for computing validated enclosures for solutions of differential equations. These techniques may be applicable for solving our constraints as well, but it remains to be seen how well they scale on roundoff error estimation problems. Furthermore, the approach from [57] requires the postcondition to be present, whereas with our forward computation we can compute the postcondition with error information automatically.

Solving interval constraints also plays a central role in symbolic execution, where path conditions over floating-points have to be solved to determine feasibility of individual paths and for finding concrete test inputs. Among the tools that are sound with respect to floating-point arithmetic is the FPSE tool [28, 15] which solves floating-point constraints using interval constraint propagation. Borges et al. [26] present a constraint solving approach which combines a meta-heuristic search with interval constraint propagation, where the interval solver is used to provide an initial estimate for the seeds of the search. Lakhotia et al. [98] combine random search and evolutionary techniques for the same purpose.

Decision Procedures

An alternative for solving floating-point constraints are dedicated decision procedures. Rümmer and Wahl [136] formalize floating-point arithmetic for the SMT-LIB format. Bit-precise constraints, however, become very large quickly. Brillout et al. [29] address this problem by using a combination of over- and under-approximations. Haller et al. [78] present an alternative approach in combining interval constraint solving with a CDCL algorithm and Gao et al. [67] present a decision procedure for nonlinear real arithmetic combining interval constraint solving with an SMT solver for linear arithmetic. Anta et al. [9] prove fixed-point constraints with a combination of bit vectors and reals. While these approach can check ranges on finite-precision numerical variables, they do not handle round-off errors or other uncertainties and cannot compute specifications automatically. Furthermore a combination of theories is problematic, and we are not aware of an approach that is able to quantify the deviation of finite-precision computations with respect to reals.

For solving real-valued constraints, several solvers exists which use interval constraint propagation together with different strategies for nonlinear computations are iSAT3 [141], dReal [68] and MetiTarski [7]. These tools and their techniques are possible alternatives for the Z3 solver that we use as a back-end for our real-valued range computation.

9.3 Quantifying Accuracy

We now turn to work with is concerned with the errors of finite-precision computations. Beyond our analysis and Fluctuat's abstract domain, different approaches have been developed for estimating round-off errors. Fang et al. [60] use affine arithmetic with a special model for floating-points to evaluate the difference between a reduced precision implementation and normal floating-point implementation, but uses probabilistic bounding to tackle over-approximations. Furthermore, this work only allows addition and multiplication. Ivancic et al. [87] use bounded model checking together with interval arithmetic to statically detect loss of accuracy in floating- point computations. While less scalable, this approach has the benefit of being able to produce counter-examples. An et al. [8] use Dekker's algorithms [54] for exact addition and multiplication to determine and track the round-off error of one computation by instrumenting the binary. The algorithms essentially rely on the non-associativity of floating-point arithmetic to compute the round-off error at each computation step.

The idea of a separation of concerns has also been successfully used in [11]. Whereas we separate the real-valued computation from the implementation, they separate reasoning about the stability of control systems in the presence of uncertainties from the implementation.

Error Estimation in Numerical Analysis

Error analysis forms an important part of numerical analysis [30, 95], perhaps best demonstrated by the fact that roundoff errors are usually discussed first, before any actual numerical algorithms. Higham gives an extensive collection of stable numerical algorithms [83], however, as he says "There is no simple recipe for designing numerically stable algorithms". Stability is usually proven or determined for each algorithm individually and manually and most work has been done for linear algorithms. Furthermore, the analyses are, as far as we know, for the most part qualitative. Our work is, in contrast, quantitative in nature and more general in the sense that we do not take into account specific (high-level, mathematical) properties of a problem into account and also focus on nonlinear arithmetic computations. Our analysis is thus more low-level, but (hopefully) more accurate and more automated. We have also placed an emphasis on sound reasoning, which is important in the verification of safety-critical systems, whereas error estimation in numerical analysis is often approximate itself. We believe that these approaches are complementary and it would be very interesting to combine ideas from numerical analysis with automated reasoning.

Fixed-point Arithmetic Compilation

Finite-precision round-off errors play an important role during the compilation process from a real-valued expression to fixed-point arithmetic, which reduces to the allocation of the number of integer bits for the variable ranges and the number of fractional bits for the accuracy. To name a few (of many more), Lee et al. [101] use affine arithmetic for bit-width optimization and also provides an overview of related approaches, both static and dynamic. The approach by Mallik et al. [110] is simulation-based and and optimizes bit-widths to reduce power consumption. Kinsman and Nicolici [94] employ a range refinement method based on SMT solvers similar to ours to determine the number of integer bits. Pang et al. [128] combine interval and affine arithmetic and an encoding of polynomials into pseudo-boolean functions. Another approach uses automatic differentiation [65] to symbolically compute the sensitivity of the outputs to the inputs. From this (unsound) approximation they derive bounds for the number of fractional bits, and also apply this technique to determine the number of necessary mantissa bits for floating-point arithmetic.
Controllers are also not unique and Majumdar et al. [108] perform a particle swarm optimization to search for the best controller with respect to several performance criteria. The evaluation of each controller takes into account implementation errors, which are translated into a mixed integer arithmetic problem.

Testing

Testing is also a popular approach for checking the accuracy of floating-point programs. While testing cannot provide guarantees, test input or counter-example generation can be very useful when debugging an application. A common thread is to perturb the numerical computation and observe the results in order to identify computations that are numerically unstable. The CADNA library [90] does this by repeatedly running a computation and randomly choosing the rounding mode for each. The hope is that if the results are consistent, then the computation is stable. However, the stochastic approach does not provide rigorous results, as because round-off errors are not always uniformly distributed (e.g. in loops). Tang et al. [149] test numerical code for accuracy by perturbing low-order bits of values and rewriting the expressions. The idea is to exaggerate initial errors and thus make inaccuracies more visible. Probabilistic arithmetic [144] is a similar approach but it does the perturbation by using different rounding modes.

Yet other techniques exist whose goal is to detect inputs which cause large round-off errors (instead of checking the numerical stability). Chiang et al. [37] developed a guided search to find inputs which maximize errors. Benz et al. [18] propose testing produce to detect accuracy problems by instrumenting code to perform a higher-precision computation side by side with the regular computations. While these approaches are sound with respect to floating-point arithmetic, they only generate or can check individual inputs and are thus not able to verify or compute output ranges or their round-off errors. Paganelli and Ahrendt [127] use a floating- point decision procedure to detect large differences in the result between computations of different precisions. Using a floating-point decision procedure allows to consider whole input ranges at once, instead of single inputs. However, since combining different theories inside the solver is problematic, they can only compare two different floating-point precision implementations and thus do not get sound error bounds with respect to a real-valued semantics. Lam et al. [99] use instrumentation to detect cancellation by monitoring exponent ranges and thus can identify possible places where significant digits may be lost. Bao and Zhang [17] refine this approach by using a cancellation bit to track relative errors through a computation. This bit is set or unset depending on the differences in exponents of the result and its operands. A computation is then flagged as unstable, if a set cancellation bit reaches a predicate such as a branch condition, in which case the user can choose to automatically restart the computation in higher precision. None of these techniques can take into account external numerical errors.

Theorem Proving

Automation in tools comes at the expense of accuracy. When one needs to prove very precise properties about finite-precision codes, then theorem proving is a suitable avenue. Many such tools nowadays include formalizations of floating-point arithmetic which makes the task easier but still requires substantial interaction from an expert user. For example, the Gappa tool [103, 50] generates a proof from source code with specifications which is checkable by the interactive theorem prover Coq [19]. It can reason about properties that can be reduced to reasoning about ranges and errors and internally uses interval arithmetic. Gappa is useful for proving very precise properties of specialized functions, such as software implementations of elementary functions. A similar approach is taken by [13] which generates verification conditions that are discharged by various theorem provers. Boldo and Marche [25] present a whole chain of tools, including Coq and Gappa for proving numerical C programs correct. Boldo and Nguyen [23] also take into account hardware features like fused-multiply instructions and extended precision registers which produce different roundoff errors than a simple evaluation. (The JVM currently does not use these features.) Theorem provers are also successfully and now standardly being used to verify floating-point hardware [119, 140, 80]. Our approach makes a different compromise on the accuracy vs. automation trade-off by being less accurate, but automatic. Interactive theorem provers can be used as complements to our tool: if our tool cannot provide sufficient accuracy, interactive tools can be employed by an expert user on selected methods and the results can then used by our tool in the context of the overall

program.
Our analysis (and most others') for floating-point precisions relies on the hardware and software conformance to the IEEE 754 standard. Furthermore, we rely on the compiler to not have bugs or re-order arithmetic operations arbitrarily. CompCert is a verified C compiler and Boldo et al. [24] present a recent extension of the proof floating-point arithmetic as well and thus able to ensure that our assumptions are valid.

9.4 Synthesis

Ideally, we want to generate numerical programs which are correct by construction, also with respect to error specifications. In this direction go specialized algorithms for computing sums [138] or dot products [126] with better accuracy in floating-point. While extremely useful for some applications, they are also very limited as they consider only one particular type of computation. In this section reviews some other more general work which attempts to improve finite-precision implementations in terms of accuracy or efficiency.

Rewriting

We are not the only ones who have exploited the non-associativity of finite-precision arithmetic. CGPE [121] is a software tool that synthesizes fast and certified code for univariate and bivariate polynomials in fixed-point arithmetic, optimized for a specific target architecture. In contrast to our work, the optimization criterion is execution time and error bounds are merely used to discard final candidate evaluation schemes that do not meet a basic error bound. Martel [111] considers rewriting fixed-point arithmetic expressions. The accuracy measure is the maximum number of bits required to hold the integral part. Ioualalen and Martel [86] refine this idea by developing an abstract domain for representing an under-approximation of mathematically equivalent expressions. Similarly to our fitness computation, their computation of accuracy of each expression uses affine arithmetic. They then use a local greedy search to find expressions with a more accurate formulation in a floating-point implementation. Their search is local in the sense that subexpressions are optimized without considering the global error, and thus may exclude many possible expressions.

Eldib and Wang [58] presents a sketching-like approach to reduce the bit width of an fixed-point expression by rewriting. Given a bit width target, a surrounding region is determined for each AST node that may overflow and is then used in an SMT query which uses test inputs and an AST skeleton to generate a better candidate program. Another call to SMT is used to ensure the two programs are equivalent. In contrast to our work, which is general and considers the global accuracy of the expression, this work only applies to linear programs and performs a local optimization of the bit width.

Approximate Computing

Rewriting an expression only changes the execution order, but keeps the same mathematical formulation and also the same data type. For certain applications which can tolerate larger errors, these requirements may be relaxed in favor of improved efficiency. Linderman et al. [103] use affine arithmetic by considering the problem of reducing precision for performance reasons. However, the resulting system still requires interactive effort. Rubio-Gonzales et al. [135] use user-defined inputs as test vectors to determine approximate error bounds which are then used to optimize program efficiency by choosing lower floating-point precision data types for some variables. While our implementation is currently limited to a fixed precision for the whole program, the techniques work, in principle, for mixed precision as well, and could be used to certify programs returned from such optimization procedures. Langou et al. [100] demonstrate that for some iterative linear algorithms it is sufficient to use double floating-point precision for the residual and update computation only, and single precision for the rest. If the condition number of the matrix is small enough, the results obtained are the same as if double precision had been used throughout. This result is specific to iterative linear algebra computations.

Schkufza et al. [143] go even a step further by modifying the computation itself. The paper presents a guided search which explores randomly changing individual instructions of a function's binary. A minimum accuracy, defined with respect to the original program, is ensured by simulation. Zhu et al. [153] show an algorithm for optimizing approximate computations at a higher level by exploring different (user-given and user-specified) implementations, in order to minimize resource consumption while satisfying a probabilistic error specification. Baek and Chilimbi [14] pursue a similar idea but determines the quality of different implementations by simulation with a user-given quality of service evaluation function. Westbrook and Chaudhuri [150] propose a modular framework for quantitatively reasoning about different approximations, with different metrics for different program constructs. We view our work as complementary as it could be used to provide the (sound) specifications of alternative implementations.

Approximations have also been proposed already at the hardware level [20], where one could leverage lower power with occasionally wrong results. In this context, Carbin et al. [32] present a static probabilistic analysis for verifying that a program running on unreliable hardware conforms to a reliability specification.

9.5 Beyond Roundoff Errors

Discontinuity Errors

Ivancic et al.[87] combine abstract interpretation with model checking to check the stability of programs, tracking one input at a time. Majumdar et al. [106] use concolic execution to find two sets of inputs which maximize the difference in the outputs. These approach are based on testing, however, and cannot prove sound bounds. Majumdar and Saha [107] show programs robust by considering all possible execution paths. The paths are obtained by a symbolic execution engine and the robustness property is proven by showing that each pair of paths does not differ more than a specified amount. While their approach can produce witnesses for a violation of this property, this work only considers integer programs and relies entirely on a solver. Chaudhuri et al. [35] develop a framework based on proof rules for showing programs robust in the sense of k-Lipschitz continuity and Gazeau et al. [69] relax this strict definition of robustness to programs with specified uncertainties and presents a framework for proving while-loops with a particular structure robust. Neither of these approaches quantifies numerical errors arising from the arithmetic computations. In our work, we quantify the error and leave it up the user and its application to determine whether it is sufficiently small, without considering notions such as robustness or continuity.

Shewchuk [146] presents techniques for arbitrary-precision and adaptive precision arithmetic, aimed at geometric applications with the goal to decide geometric predicates robustly. Geometric applications usually do not tolerate inaccuracies, for instance for deciding whether a point is left or right of a line, many digits may be necessary. Our work is not suitable for such accurate codes, instead we target applications, where we can statically prove that a certain data type precision is sufficient and take advantage of efficient hardware during the execution.

Truncation Errors

One possible way to deal with truncation errors, is to use self-validated methods, which return guaranteed enclosures of the solution. For example, [137] contains a fairly complete overview and an implementation exists in the INTLAB library [139] for MATLAB. The main difference to our work is that these methods compute the solution, using interval arithmetic throughout the computation. In contrast, we use the underlying theorems as a *verification* method that accepts solutions computed by an arbitrary method. This allows us to leverage the generally good results and efficiency of numerical methods with sound results. Moreover, our implementation performs part of the computation already at compile time, and is thus more efficient.
In the case of systems of linear equations, one can use the linearity for optimizations [123]. The algorithm remains an iterative solver though. Demmel et al. [55] give an iterative refinement algorithm for linear systems that uses higher precision arithmetic to compute the residual. The techniques cannot however be translated to nonlinear systems. Since we do not compute residuals that suffer heavily from cancellation errors in our approach, we believe that the additional cost of higher precision arithmetic is not warranted in order to achieve a slightly better accuracy.

10 Conclusion

Some may believe that round-off errors always distort results extensively so we should not use floating-point or fixed-point arithmetic at all or, at the other extreme, that the errors are very small and can thus be ignored. More likely, reality is somewhere in between. For many applications, finite-precision arithmetic is perfectly adequate, if used carefully and with some error consideration. Furthermore, there is a fundamental trade-off between many orthogonal or conflicting interests: efficiency vs. accuracy of the application, runtime overhead vs. static analysis over-approximation in the verification approach, scalability of the analysis technique vs. counter- example generation, etc. The best technique and implementation for a given mathematical problem may lie anywhere on this spectrum and our goal was to help programmers navigate this space.

We have presented tools and techniques for automated, sound and accurate numerical error analysis and shown how these building blocks can be used for synthesis of numerical programs with explicit error specifications.
On the technical side, we developed and studied in detail different approaches for sound range computation and error quantification. We used and combined more traditional techniques for sound computation (interval and affine arithmetic) with more recently developed nonlinear solvers and investigated their utility and limits. We further showed how such an error computation can be joined with other techniques (genetic programming or validated numerics) to go beyond just round-off error estimation and actively improve accuracy or quantify truncation errors.
On the conceptual side, we argue that the current state of the art of programming numerical codes is too low level for many applications and that today's reasoning capabilities enable a higher-level programming model which is semantically closer to the mathematical domain. Recognizing that one approach does not fit all, we nonetheless explored different ways of integrating sound error analysis into a programming language, both statically and dynamically.

We believe that our results are encouraging and that our work is a successful step towards our goal of helping programmers write numerical codes that do what they are expected to do.

We have mostly focused on round-off errors, although our propagation algorithms are oblivious to the type or the errors. Truncation errors are an integral part of numerical computations and can often be actually (much) larger than round-off errors. Extending our work to *additional sources of errors* would require an appropriate extension of our specification language to specifying the ‘intended’ meaning of a computation. Furthermore, we expect the automated computation of such errors to pose new challenges.

Such an intended meaning of a computation that goes beyond the real-valued vs finite-precision difference that we considered in this thesis is only one example of more *functional verification* targets. Other properties of interest could be the convergence behavior of iterative algorithms or the stability for example of numerical integration algorithms. We believe that our separation of round-off errors from the ideal computation can help free functional verification attempts from the low-level considerations. They can thus focus on the *real-valued* algorithms, while our error estimates can (hopefully) validate the results also for the noisy finite-precision implementation.

We have focused on a sound error analysis, but round-off (and other) errors do not always reach the worst-case magnitude. Relaxing this requirement towards a *probabilistic approach* would allow for tighter (and perhaps more realistic) error estimates. It would be interesting to see if and how our techniques can be used in this context to provide confidence in the analysis' results.

Most *approximations* that we have considered were inevitable; we can sometimes reduce round-off errors, but not eliminate them entirely. The inherent error tolerance of many applications allows us to go the other way, namely to do computations less accurately than what is possible and save resources. Previous work considered optimizations on a rather low level [143] or required the user to come up with candidate approximations [153, 14]. Another possibility is to use the knowledge already available in numerical mathematics and combine it with automated reasoning techniques to generate these candidates at a higher level and with accuracy guarantees.

Finally, there is a lot of potential for *synthesis* of numerical codes. Our rewriting focused on a rather small number of mathematical equivalences, which can be significantly extended, also to other mathematical functions like square root and trigonometry. We can also imagine to apply our techniques to the selection of different algorithms, with different accuracy specifications. We see the challenge both in the huge search space as well as in the numerical error estimation. In would be interesting to investigate, whether ‘rules of thumb’ exist which can improve accuracy in general, even if not optimally. It may also not be possible to find one expression that is optimal for all inputs, rather it may be necessary to consider several of them. It will thus be necessary to develop techniques which determine the ranges of validity so an overall error guarantee can be maintained.

There are still more than a few stones left unturned!

Bibliography

[1] Caliper open source framework. `http://code.google.com/p/caliper/`.

[2] Control Tutorial for Matlab and Simulink. `http://www.library.cmu.edu/ctms/ctms/`.

[3] GNU Octave. `http://www.gnu.org/software/octave/index.html`.

[4] MathWorks MATLAB. `http://www.mathworks.ch/`.

[5] Wolfram Mathematica. `http://www.wolfram.com/mathematica/`.

[6] The Computer Language Benchmarks Game. `http://benchmarksgame.alioth.debian.org/`, accessed May 2014.

[7] Behzad Akbarpour and LawrenceCharles Paulson. MetiTarski: An Automatic Theorem Prover for Real-Valued Special Functions. *Journal of Automated Reasoning*, pages 175–205, 2010.

[8] David An, Ryan Blue, Michael Lam, Scott Piper, and Geoff Stoker. FPInst: Floating Point Error Analysis Using Dyninst. `http://www.freearrow.com/downloads/files/fpinst.pdf`, 2008.

[9] Adolfo Anta, Rupak Majumdar, Indranil Saha, and Paulo Tabuada. Automatic Verification of Control System Implementations. In *EMSOFT*, 2010.

[10] Adolfo Anta and Paulo Tabuada. To Sample or not to Sample: Self-Triggered Control for Nonlinear Systems. *IEEE Trans. Automatic Control*, 55(9), 2010.

[11] Adolfo Anta Martinez, Rupak Majumdar, Indranil Saha, and Paulo Tabuada. Automatic Verification of Control System Implementations. In *EMSOFT*, 2010.

[12] K. J. Astrom and R. M. Murray. *Feedback Systems*. Princeton University Press, 2008.

[13] Ali Ayad and Claude Marché. Multi-prover verification of floating-point programs. In *IJCAR*, 2010.

[14] Woongki Baek and Trishul M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *PLDI*, 2010.

[15] Roberto Bagnara, Matthieu Carlier, Roberta Gori, and Arnaud Gotlieb. Symbolic Path-oriented Test Data Generation for Floating-point Programs. In *ICST*, 2013.

[16] David H. Bailey, Yozo Hida, Xiaoye S. Li, and Brandon Thompson. C++/Fortran-90 double-double and quad-double package. `http://crd-legacy.lbl.gov/~dhbailey/mpdist/`, 2013.

[17] Tao Bao and Xiangyu Zhang. On-the-fly Detection of Instability Problems in Floating-point Program Execution. In *OOPSLA*, 2013.

[18] Florian Benz, Andreas Hildebrandt, and Sebastian Hack. A Dynamic Program Analysis to Find Floating-point Accuracy Problems. In *PLDI*, 2012.

[19] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development*. Springer, 2004.

[20] Jesse Bingham and Joe Leslie-Hurd. Verifying Relative Error Bounds Using Symbolic Simulation. In *CAV*, 2014.

[21] Régis Blanc, Etienne Kneuss, Viktor Kuncak, and Philippe Suter. An Overview of the Leon Verification System: Verification by Translation to Recursive Functions. In *Scala Workshop*, 2013.

[22] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *PLDI*, 2003.

[23] S. Boldo and T. M. T. Nguyen. Hardware-independent proofs of numerical programs. In *Proceedings of the Second NASA Formal Methods Symposium*, 2010.

[24] Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. A Formally-Verified C Compiler Supporting Floating-Point Arithmetic. In *IEEE Symposium on Computer Arithmetic*, pages 107–115, 2013.

[25] Sylvie Boldo and Claude Marché. Formal verification of numerical programs: from C annotated programs to mechanical proofs. *Mathematics in Computer Science*, 5(4), 2011.

[26] Mateus Borges, Marcelo d'Amorim, Saswat Anand, David Bushnell, and Corina S. Pasareanu. Symbolic Execution with Interval Solving and Meta-heuristic Search. In *ICST*, 2012.

[27] Cristina Borralleras, Salvador Lucas, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. Sat modulo linear arithmetic for solving polynomial constraints. *J. Automated Reasoning*, 48(1), 2012.

[28] B. Botella, A. Gotlieb, and C. Michel. Symbolic execution of floating-point computations. *Softw. Test. Verif. Reliab.*, 16(2), 2006.

[29] A. Brillout, D. Kroening, and T. Wahl. Mixed abstractions for floating-point arithmetic. In *FMCAD*, 2009.

[30] Richard L. Burden and J. Douglas Faires. *Numerical Analysis*. Cengage Learning, 2011.

[31] Eugene Burmako. Scala Macros: Let Our Powers Combine!: On How Rich Syntax and Static Types Work with Metaprogramming. In *SCALA*, 2013.

[32] Michael Carbin, Sasa Misailovic, and Martin C. Rinard. Verifying Quantitative Reliability for Programs That Execute on Unreliable Hardware. In *OOPSLA*, 2013.

[33] Swarat Chaudhuri, Azadeh Farzan, and Zachary Kincaid. Consistency Analysis of Decision-making Programs. In *POPL*, 2014.

[34] Swarat Chaudhuri, Sumit Gulwani, and Roberto Lublinerman. Continuity analysis of programs. In *POPL*, 2010.

[35] Swarat Chaudhuri, Sumit Gulwani, Roberto Lublinerman, and Sara Navidpour. Proving Programs Robust. In *ESEC/FSE*, 2011.

[36] Liqian Chen, Antoine Miné, Ji Wang, and Patrick Cousot. Interval Polyhedra: An Abstract Domain to Infer Interval Linear Relationships. In *SAS*, 2009.

[37] Wei-Fan Chiang, Ganesh Gopalakrishnan, Zvonimir Rakamaric, and Alexey Solovyev. Efficient Search for Inputs Causing High Floating-point Errors. In *PPoPP*, 2014.

[38] Konstantina Christakou and Jean-Yves Le Boudec. State estimation problem for power systems. Personal communication, 2013.

[39] Hélène Collavizza, Claude Michel, Olivier Ponsini, and Michel Rueher. Generating Test Cases Inside Suspicious Intervals for Floating-point Number Programs. In *CSTVA*, 2014.

[40] George E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. *Automata Theory and Formal Languages*, 33:134–183, 1975.

[41] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE Analyser. In *ESOP*, 2005.

[42] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*, 1977.

[43] Germund Dahlquist and Åke Björck. *Numerical Methods in Scientific Computing*. Society for Industrial and Applied Mathematics, 2008.

[44] Eva Darulova and Viktor Kuncak. Trustworthy Numerical Computation in Scala. In *OOPSLA*, 2011.

[45] Eva Darulova and Viktor Kuncak. Certifying Solutions for Numerical Constraints. In *RV*, 2012.

[46] Eva Darulova and Viktor Kuncak. On Numerical Error Propagation with Sensitivity. Technical Report 200132, EPFL, 2014.

[47] Eva Darulova and Viktor Kuncak. Sound Compilation of Reals. In *POPL*, 2014.

[48] Eva Darulova, Viktor Kuncak, Rupak Majumdar, and Indranil Saha. Synthesis of Fixed-Point Programs. In *EMSOFT*, 2013.

[49] M. Davis. DoubleDouble.java. `http://tsusiatsoftware.net/dd/main.html`.

[50] Florent de Dinechin, Christoph Lauter, and Guillaume Melquiond. Certifying the Floating-Point Implementation of an Elementary Function Using Gappa. *IEEE Trans. Comput.*, 60(2), 2011.

[51] L. H. de Figueiredo and J. Stolfi. *Self-Validated Numerical Methods and Applications*. IMPA/CNPq, Brazil, 1997.

[52] L. H. de Figueiredo and J. Stolfi. Affine Arithmetic: Concepts and Applications. *Numerical Algorithms*, 37(1-4), 2004.

[53] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *TACAS*, 2008.

[54] T.J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18(3):224–242, 1971.

[55] James W. Demmel, Yozo Hida, William Kahan, Xiaoye S. Li, Soni Mukherjee, and E. Jason Riedy. Error Bounds from Extra Precise Iterative Refinement. Technical report, EECS Department, University of California, Berkeley, 2005.

[56] Vijay D'Silva, Leopold Haller, Daniel Kroening, and Michael Tautschnig. Numeric Bounds Analysis with Conflict-driven Learning. In *TACAS*, 2012.

[57] Jan Duracz and Michal Konečný. Polynomial function intervals for floating-point software verification. *Annals of Mathematics and Artificial Intelligence*, 70(4), 2014.

[58] Hassan Eldib and Chao Wang. An smt based method for optimizing arithmetic computations in embedded software code. In *FMCAD*, 2013.

[59] A. G. Ershov and T. P. Kashevarova. Interval Mathematical Library Based on Chebyshev and Taylor Series Expansion. *Reliable Computing*, 11, 2005.

[60] Claire F. Fang, Rob A. Rutenbar, and Tsuhan Chen. Fast, Accurate Static Analysis for Fixed-Point Finite-Precision Effects in DSP Designs. In *ICCAD*, 2003.

[61] Jérôme Feret. Static Analysis of Digital Filters. In *ESOP*, 2004.

[62] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. MPFR: A Multiple-precision Binary Floating-point Library with Correct Rounding. *ACM Trans. Math. Softw.*, 33(2), 2007.

[63] Martin Fränzle and Christian Herde. HySAT: An efficient proof engine for bounded model checking of hybrid systems. *Formal Methods in System Design*, 30(3), 2007.

[64] Martin Fränzle, Christian Herde, Tino Teige, Stefan Ratschan, and Tobias Schubert. Efficient Solving of Large Non-linear Arithmetic Constraint Systems with Complex Boolean Structure. *Journal on Satisfiability, Boolean Modeling and Computation*, 1, 2007.

[65] A.A. Gaffar, O. Mencer, and W. Luk. Unifying bit-width optimisation for fixed-point and floating-point designs. In *FCCM*, 2004.

[66] Sicun Gao, Jeremy Avigad, and Edmund M. Clarke. Delta-Complete Decision Procedures for Satisfiability over the Reals. In *IJCAR*, 2012.

[67] Sicun Gao, M. Ganai, F. Ivancic, A. Gupta, S. Sankaranarayanan, and E.M. Clarke. Integrating ICP and LRA solvers for deciding nonlinear real arithmetic problems. In *FMCAD*, 2010.

[68] Sicun Gao, Soonho Kong, and Edmund M. Clarke. dReal: An SMT Solver for Nonlinear Theories over the Reals. In *CADE*, 2013.

[69] Ivan Gazeau, Dale Miller, and Catuscia Palamidessi. A non-local method for robustness analysis of floating point programs. In *QAPL*, 2012.

[70] Khalil Ghorbal, Eric Goubault, and Sylvie Putot. A Logical Product Approach to Zonotope Intersection. In *CAV*, 2010.

[71] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1), 1991.

[72] Eric Goubault and Sylvie Putot. Static Analysis of Finite Precision Computations. In *VMCAI*, 2011.

[73] Eric Goubault and Sylvie Putot. Robustness Analysis of Finite Precision Implementations. In *APLAS*, 2013.

[74] Eric Goubault, Sylvie Putot, and Franck Védrine. Modular Static Analysis with Zonotopes. In *SAS*, 2012.

[75] M. Green and D. J. N. Limebeer. *Linear Robust Control.* Prentice Hall, 1994.

[76] Andreas Griewank. A mathematical view of automatic differentiation. *Acta Numerica*, 12:321–398, 2003.

[77] John L. Gustafson. *The End of Error: Unum Computing.* CRC Press, 2015.

[78] L. Haller, A. Griggio, M. Brain, and D. Kroening. Deciding floating-point logic with systematic abstraction. In *FMCAD*, 2012.

[79] E.R. Hansen. A generalized interval arithmetic. In *Interval Mathematics*, volume 29, pages 7–18. Springer Berlin Heidelberg, 1975.

[80] John Harrison. Floating-Point Verification using Theorem Proving. In *Formal Methods for Hardware Verification*, pages 211–242, 2006.

[81] L. Hatton and A. Roberts. How Accurate is Scientific Software? *IEEE Trans. Softw. Eng.*, 20(10), 1994.

[82] Yozo Hida, Xiaoye S. Li, and David H. Bailey. Library for double-double and quad-double arithmetic. Technical report, NERSC Division, Lawrence Berkeley National Laboratory, 2007.

[83] Nicholas J Higham. *Accuracy and Stability of Numerical Algorithms*. Siam, 2002.

[84] Computer Society IEEE. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, 2008.

[85] Germany Institute of Microelectronic Systems, Hannover. aaflib - An Affine Arithmetic C++ Library. `http://aaflib.sourceforge.net/`, 2010.

[86] Arnault Ioualalen and Matthieu Martel. A New Abstract Domain for the Representation of Mathematically Equivalent Expressions. In *SAS*, 2012.

[87] F. Ivancic, M.K. Ganai, S. Sankaranarayanan, and A. Gupta. Numerical stability analysis of floating-point computations using software model checking. In *MEMOCODE*, 2010.

[88] Java-API. Java Platform SE 7 API - java.lang.Math. `http://docs.oracle.com/javase/7/docs/api/`, 2014.

[89] Bertrand Jeannet and Antoine Miné. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *CAV*, 2009.

[90] F. Jézéquel and J.-M. Chesneaux. CADNA: a library for estimating round-off error propagation. *Computer Physics Communications*, 178(12), 2008.

[91] J. Jiang, W. Luk, and D. Rueckert. FPGA-Based Computation of Free-Form Deformations. In *Field - Programmable Logic and Applications*, pages 1057–1061. Springer, 2003.

[92] Dejan Jovanović and Leonardo de Moura. Solving Non-linear Arithmetic. In *IJCAR 2012*, 2012.

[93] W. Kahan. Miscalculating Area and Angles of a Needle-like Triangle. Technical report, University of California Berkeley, 2000.

[94] A.B. Kinsman and N. Nicolici. Finite Precision bit-width allocation using SAT-Modulo Theory. In *DATE*, 2009.

[95] Jaan Kiusalaas. *Numerical Methods in Engineering with MATLAB.* Cambridge University Press, 2005.

[96] R. Krawczyk. Newton-Algorithmen zur Bestimmung von Nullstellen mit Fehlerschranken. *Computing*, 4:187–201, 1969.

[97] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient State Merging in Symbolic Execution. In *PLDI*, 2012.

[98] Kiran Lakhotia, Nikolai Tillmann, Mark Harman, and Jonathan de Halleux. FloPSy - Search-Based Floating Point Constraint Solving for Symbolic Execution. In *ICTSS*, 2010.

[99] Michael O. Lam, Jeffrey K. Hollingsworth, and G.W. Stewart. Dynamic floating-point cancellation detection. *Parallel Computing*, 39(3), 2013.

[100] Julie Langou, Julien Langou, Piotr Luszczek, Jakub Kurzak, Alfredo Buttari, and Jack Dongarra. Exploiting the Performance of 32 Bit Floating Point Arithmetic in Obtaining 64 Bit Accuracy (Revisiting Iterative Refinement for Linear Systems). In *SC*, 2006.

[101] D. Lee, A. A. Gaffar, R. C. C. Cheung, O. Mencer, W. Luk, and G. A. Constantinides. Accuracy-Guaranteed Bit-Width Optimization. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 25(10), 2006.

[102] X. Leroy. Verified Squared: Does Critical Software Deserve Verified Tools? In *POPL*, 2011.

[103] Michael D. Linderman, Matthew Ho, David L. Dill, Teresa H. Meng, and Garry P. Nolan. Towards program optimization through automated analysis of numerical precision. In *CGO*, 2010.

[104] T. Lindholm and F. Yellin. *Java Virtual Machine Specification.* Addison-Wesley Longman Publishing Co., Inc., 2nd edition, 1999.

[105] Sean Luke, Liviu Panait, Gabriel Balan, and Sean et al. Paus. ECJ: A Java based Evolutionary Computation Research System. `http://cs.gmu.edu/~eclab/projects/ecj/`.

[106] R. Majumdar, I. Saha, and Zilong Wang. Systematic Testing for Control Applications. In *MEMOCODE*, 2010.

[107] Rupak Majumdar and Indranil Saha. Symbolic Robustness Analysis. In *RTSS*, 2009.

[108] Rupak Majumdar, Indranil Saha, and Majid Zamani. Synthesis of Minimal-Error Control Software. In *EMSOFT*, pages 123–132, 2012.

[109] K. Makino and M. Berz. Taylor Models and Other Validated Functional Inclusion Methods. *International Journal of Pure and Applied Mathematics*, 4(4), 2003.

[110] A. Mallik, D. Sinha, P. Banerjee, and H. Zhou. Low-Power Optimization by Smart Bit-Width Allocation in a SystemC-Based ASIC Design Environment. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 26(3), 2007.

[111] Matthieu Martel. Enhancing the Implementation of Mathematical Formulas for Fixed-Point and Floating-Point Arithmetics. *Formal Methods in System Design*, 35(3), 2009.

[112] R. Martin, H. Shou, I. Voiculescu, A. Bowyer, and G. Wang. Comparison of interval methods for plotting algebraic curves. *Comput. Aided Geom. Des.*, 19(7), 2002.

[113] MATLAB. Embedded Code Generation. `http://www.mathworks.ch/embedded-code-generation`, 2014.

[114] P. McLane, L. Peppard, and K. Sundareswaran. Decentralized Feedback Controls for the Brakeless Operation of Multilocomotive Powered Trains. *IEEE Trans. Autom. Control*, 21(3), 1976.

[115] Jean Meeus. *Astronomical Algorithms*. Willmann-Bell, 1999.

[116] Kurt Mehlhorn and Stefan Schirra. Exact Computation with leda_real — Theory and Geometrie Applications. In *Symbolic Algebraic Methods and Verification Methods*, pages 163–172. Springer Vienna, 2001.

[117] Antoine Miné. Relational Abstract Domains for the Detection of Floating-Point Run-Time Errors. In *ESOP*, 2004.

[118] D. Monniaux. The pitfalls of verifying floating-point computations. *ACM Trans. Program. Lang. Syst.*, 30(3), 2008.

[119] J. S. Moore, T. W. Lynch, and M. Kaufmann. A Mechanically Checked Proof of the AMD5 K86 Floating Point Division Program. *IEEE Trans. Computers*, 47(9), 1998.

[120] R.E. Moore. *Interval Analysis*. Prentice-Hall, 1966.

[121] C. Mouilleron and G. Revy. Automatic Generation of Fast and Certified Code for Polynomial Evaluation. In *ARITH*, 2011.

[122] James D. Murray. *Mathematical Biology, I. An Introduction*. Springer, 2002.

[123] Hong Diep Nguyen and Nathalie Revol. Solving and Certifying the Solution of a Linear System. *Reliable Computing*, 15(2), 2011.

[124] William L. Oberkampf and Christopher J. Roy. *Verification and Validation in Scientific Computing*. Cambridge University Press, 2010.

[125] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, 2008.

[126] T. Ogita, Siegfried M. Rump, and Shin'ichi Oishi. Accurate Sum and Dot Product. *SIAM J. Sci. Comput.*, 26(6):1955–1988, 2005.

[127] Gabriele Paganelli and Wolfgang Ahrendt. Verifying (In-)Stability in Floating-point Programs by Increasing Precision, using SMT Solving. In *SYNASC*, 2013.

[128] Yu Pang, Katarzyna Radecka, and Zeljko Zilic. An Efficient Hybrid Engine to Perform Range Analysis and Allocate Integer Bit-widths for Arithmetic Circuits. In *ASPDAC*, 2011.

[129] Riccardo Poli, William B. Langdon, and Nicholas F. McPhee. *A Field Guide to Genetic Programming*. Lulu Enterprises, 2008.

[130] R. Pozo and B. R. Miller. Java SciMark 2.0. `http://math.nist.gov/scimark2/about.html`, 2004.

[131] D. M. Priest. Algorithms for Arbitrary Precision Floating Point Arithmetic. In *Proceedings of the 10th Symposium on Computer Arithmetic*, 1991.

[132] Alexander Prokopec. Scalameter. `http://scalameter.github.io/`, 2014.

[133] Alfio Quarteroni, Fausto Saleri, and Paola Gervasio. *Scientific Computing with MATLAB and Octave*. Springer, 3rd edition, 2010.

[134] H. H. Rosenbrock. *Computer-Aided Control System Design*. Academic Press, 1974.

[135] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, and David Hough. Precimonious: Tuning Assistant for Floating-point Precision. In *SC*, 2013.

[136] Philipp Rümmer and Thomas Wahl. An SMT-LIB Theory of Binary Floating-Point Arithmetic. In *Informal proceedings of 8th International Workshop on Satisfiability Modulo Theories (SMT) at FLoC*, 2010.

[137] Siegfried M. Rump. Verification methods: rigorous results using floating-point arithmetic. *Acta Numerica*, 19:287–449, 2010.

[138] Siegfried M. Rump, Takeshi Ogita, and Shin'ichi Oishi. Accurate Floating-Point Summation Part i: Faithful Rounding. *SIAM J. Sci. Comput.*, 31(1):189–224, 2008.

[139] S.M. Rump. INTLAB - INTerval LABoratory. In *Developments in Reliable Computing*. Kluwer Academic Publishers, 1999.

[140] D. M. Russinoff. A Mechanically Checked Proof of Correctness of the AMD K5 Floating Point Square Root Microcode. *Formal Methods in System Design*, 14(1), 1999.

[141] Karsten Scheibler, Alexej Disterhoft, Andreas Eggers, Felix Neubauer, Leonore Winterer, Stefan Kupferschmid, and Tobias Paxian. iSAT3. `https://projects.avacs.org/projects/isat3`, 2014.

[142] Karsten Scheibler, Stefan Kupferschmid, and Bernd Becker. Recent Improvements in the SMT solver iSAT. In *MBMV*, 2013.

[143] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic Optimization of Floating-point Programs with Tunable Precision. In *PLDI*, 2014.

[144] N.S. Scott, F. Jézéquel, C. Denis, and J.-M. Chesneaux. Numerical 'health check' for scientific codes: the CADNA approach. *Computer Physics Communications*, 176(8), 2007.

[145] Charles Severance. An Interview with the Old Man of Floating-Point: Reminiscences elicited from William Kahan. `http://www.cs.berkeley.edu/~wkahan/ieee754status/754story.html`, A condensed version appeared in Computer, 31:114–115, 1998, 1998.

[146] Jonathan Richard Shewchuk. Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. *Discrete & Computational Geometry*, 18(3), 1997.

[147] Huahao Shou, Hongwei Lin, Ralph R. Martin, and Guojin Wang. Modified affine arithmetic in tensor form for trivariate polynomial evaluation and algebraic surface plotting. *Journal of Computational and Applied Mathematics*, 195(1–2), 2006.

[148] Jorge Stolfi. Jorge stolfi's c libraries. `http://www.ic.unicamp.br/~stolfi/EXPORT/software/c/Index.html#libaa`, 2007.

[149] Enyi Tang, Earl Barr, Xuandong Li, and Zhendong Su. Perturbing numerical calculations for statistical analysis of floating-point program (in)stability. In *ISSTA*, 2010.

[150] Edwin M. Westbrook and Swarat Chaudhuri. A Semantics for Approximate Program Transformations. *CoRR*, abs/1304.5531, 2013.

[151] C. Woodford and C. Phillips. *Numerical Methods with Worked Examples*, volume 2nd. Springer, 2012.

[152] L. Zhang, Y. Zhang, and W. Zhou. Tradeoff between Approximation Accuracy and Complexity for Range Analysis using Affine Arithmetic. *Journal of Signal Processing Systems*, 61(3), 2010.

[153] Zeyuan Allen Zhu, Sasa Misailovic, Jonathan A. Kelner, and Martin Rinard. Randomized accuracy-aware program transformations for efficient approximate computations. In *POPL*, 2012.

Eva Darulová

Education

2009–2014	**PhD**, École Polytechnique Fédérale de Lausanne, Switzerland. Laboratory for Automated Reasoning and Analysis (LARA), thesis advisor: Viktor Kuncak
2005–2009	**Honours Bachelor**, University College Dublin, Ireland, First Class Honours. GPA: 4.2/4.2. Joint major in mathematical physics and computer science

Experience

May–Aug 2011	**Software engineering intern**, Google, Zurich, Switzerland. Main responsible for the Youtube EDU project. This included the implementation of the front-end (HTML, CSS, Python, Javascript) and the back-end (C++, Python). I also used machine learning techniques to develop classification and filtering for educational videos.
June–July 2009	**Student intern**, Ericsson GmbH - Eurolab, Aachen, Germany. Support for the test automation group (TTCN-3, C++). I supported the team developing test automation code for telecommunication networks.
July–Aug 2008	**Student intern**, Ericsson GmbH - Eurolab, Aachen, Germany. Design and development of tools for project management (Visual Basic for MS Excel and Word). I worked closely with project managers and developed applications in MS Excel and MS Word that automated common analysis tasks. This included defining the requirements based on discussions, developing a suitable interface for non-experts, implementation and testing.
June–Aug 2007	**Student intern**, Openjaw Technologies, Dublin, Ireland. Software development (HTML, Java, JavaScript) and testing

Publications

Peer-reviewed Conference Papers

E. Darulova, V. Kuncak. Sound Compilation of Reals. In *POPL*, 2014.

E. Darulova, V. Kuncak, I. Saha, R. Majumdar. Synthesis of Fixed-Point Programs. In *EMSOFT*, 2013.

E. Darulova, V. Kuncak. Certifying Solutions for Numerical Constraints. In *RV*, 2012.

E. Darulova, V. Kuncak. Trustworthy Numerical Computation in Scala. In *OOPSLA*, 2011.

Technical Reports

E. Darulova, V. Kuncak. On Numerical Error Propagation with Sensitivity. EPFL TR 200132, 2014.

Posters

E. Darulova. Programming with Uncertainties. In *FMCAD Student Forum*, 2013.

Invited Presentations

June '14	Approximate Computing Workshop, Edinburgh, Scotland
May '14	Alpine Verification Meeting, Frejus, France
Oct '13	Dagstuhl Seminar 13411, Germany
Jan '13	Workshop on Synthesis, Verification and Analysis of Rich Models, Rome, Italy
Mar '12	Geneva observatory, Switzerland

Peer reviews

PPDP 2014 (subreviewer), FMCAD 2014 (subreviewer), DATE 2014 (external reviewer), PLDI 2012 (additional reviewer), VMCAI 2011, (external reviewer), SAS 2011 (external reviewer), FMICS 2010 (external reviewer)

Teaching experience and Mentoring

TA for Programming Principles autumn '13.
Fundamental concepts of functional programming. I was involved in leading exercise sessions at EPFL and designing of exercises. This course was simultaneously also given on Coursera.

Google Summer of Code mentor, summer '13.

Semester project supervision spring '12.

TA for Synthesis, Analysis and Verification in spring '11, '12 and '13.
Basic concepts in software verification including topics such Hoare logic, Abstract Interpretation, Model checking and SMT solving. I was responsible for exercises, creating exams and occasionally gave a lecture.

TA for Compiler Construction in autumn '11 and '12.
Compiler basics such as lexical analysis, parsing, type checking, code generation and program analysis. I was responsible for creating and correcting exercises and exams.

TA for Introduction to Object Oriented Programming in autumn '10.
Introductory programming course for first year computer science students. I was answering questions during exercise sessions, helped create and correct the final exam.

Awards and Honors

2014	Young Researcher participant at the 2nd Heidelberg Laureate Forum
2010	Google EMEA Anita Borg finalist
2009	Irish Undergraduate Awards gold medal winner in the area of computer science
2009	IBM Ireland Open Source competition winner
2008	Fr Ciaran Ryan Prize (best average grade in 5 mathematical physics subjects)
2008	Invitation to IBM EMEA Best Student Recognition Event
2008	BSc Science(Mathematical Physics) and BSc Science(Computer Science) Stage 4 Scholarship
2007	Keating Prize (best in class in mathematical physics)
2007	BSc Science(Mathematical Physics) and BSc Science(Computer Science) Stage 3 Scholarship
2006	BSc Science(Mathematics) and BSc Science(Computer Science) Stage 2 Scholarship

Professional Training

2014 **Introduction to University Teaching**.
Three day workshop presenting the main theoretical concepts and practical approaches to teaching and learning in higher education.

Software

Open Source software from my PhD project as well as other contributions are on Github: `https://github.com/malyzajko`

Technical knowledge

Scala, Java, Python, C++, Javascript, Linux, OS X, HTML/CSS

Languages

Slovak, German (native), English (C2), Czech (C1), French (B2) (european scale)