TECHNISCHE UNIVERSITÄT MÜNCHEN

TUM SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY

Improving Analysis and Optimization of Finite-Precision Programs

Anastasia Isychev

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität Munchen zur Erlangung einer

Doktorin der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitz:
Prof. Tobias Nipkow, Ph.D.

Prüfer*innen der Dissertation:
1. Prof. Dr. Helmut Seidl
2. Prof. Dr. Eva Darulova
3. Prof. Dr. Zachary Tatlock

Die Dissertation wurde am 20.06.2023 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 04.10.2023 angenommen.

Acknowledgments

First and foremost, I would like to thank my two advisers Eva Darulova and Helmut Seidl, without whom my doctoral studies would not have been possible. Helmut always had a fresh perspective on my ideas and kept me on my toes (in a good sense). Eva was an incredible mentor and a role model, when I grow up (metaphorically) I want to be like her: brave, talented, supportive and knowledgeable. I think it's safe to say my life would have been very different if she didn't convince me to go for a PhD seven years ago in Saarbrücken. I am forever grateful that she believed in me! I would further like to thank Zach Tatlock for taking the time to review my thesis.

I was lucky to have not one, but two groups of colleagues to go through the ups and downs of doctoral studies. I would like to thank Debasmita, Heiko and Rosa from the AVA group at MPI-SWS for always being there to discuss the weirdness of finite-precision computations and share some laughs. Even though we were in different towns, they were just one Mattermost message away. I want to especially thank Debasmita for reading my drafts and sharing the struggle of thesis writing!

My second home, TUM, was also full of nice people. All the past and present members of the I2 and I7 labs made my years at TUM special. I am especially grateful to have had amazing officemates Raphaela and Yannick, who created a warm atmosphere in our shared 02.07.053 for both chatting and hard work. Special thanks to all German native speakers at the I2 lab who put up with my (sometimes broken) German on a daily basis.

Apart from my work family, I want to thank my actual family, my mom, godmother and my sister, who supported me in exploring student life (again) 3000km away from them. My friends from Saint Petersburg—Tania, Vika, Julia—who encouraged and celebrated me despite the (geographical) distance between us. They have always made me feel at home again, whenever I returned to Saint Petersburg.

Academia so far has been a great source of friends. I am especially thankful to have met Marijana, my soulsister, at the conference where I gave a talk about my first published paper. She was always there to support me when I felt down and tell me I'm smart. I want to thank my friends outside of academia—Toni, Vlad, Ira, Kati—for sharing the amazing memories we've made together during my time in Munich. And special thanks to Ralf who tolerated my long working hours and made sure I had dinner after a day of writing.

Finally, I am also thankful to a person who back in 2017 did not update the web page of PUMA graduate program that falsely claimed there are 12 open PhD-student positions, which led to me contacting Helmut. The rest is history.

Abstract

Numerical software has many forms: from smartphones and wearable devices that analyze our heartbeat and sleeping behavior to complex robots performing tasks in outer space, simulations predicting earthquakes, and many more. All of these programs run on hardware with limited resources, therefore, real-valued algorithms have to be approximated with finite-precision implementations. Naturally, this causes discrepancies between ideal and finite-precision results, which are called *rounding errors*. While some applications tolerate errors well, others require the computations to be performed accurately, as the cost of getting the result wrong is high. For these programs, one needs a guaranteed way to bound the errors.

Today's tools can estimate guaranteed worst-case rounding errors. The computed error bounds are used to assess an implementation (for instance, whether it is accurate enough) and to optimize it. However, the analyses do not scale well to large programs and only provide limited support for loops. These limitations extend to state-of-the-art optimizations; even for straight-line code, they have modest effects when optimizing programs defined on a large input domain, or if they include mathematical library function calls.

In this thesis, we improve static analyses and optimizations of numerical programs in different directions. We advance the *analyses* by handling bounded and unbounded loops more efficiently. First, we improve the scalability of the rounding error analysis on bounded loops over large vectors and matrices. Secondly, we introduce a method that quickly generates tight inductive invariants for unbounded loops and has wider applicability than state-of-the-art tools.

In the second part of the thesis, we improve sound *optimizations* of straight-line numerical programs. We propose an optimization that speeds up numerical kernels by customizing elementary function calls while providing a guarantee on the overall error. We also introduce a method that boosts the existing sound optimizations for accuracy and performance by applying them to small parts of a program's (original) large input domain, thus resulting in more aggressive optimizations.

We have developed open-source prototype tools for our techniques and evaluated them on benchmarks from scientific computing, machine learning, and other domains. Our results are encouraging and demonstrate a significant improvement in analyses' scalability and applicability and a boost in the optimization objectives.

Keywords: finite precision, numerical software, static analysis, floating-point arithmetic, fixed-point arithmetic, rounding errors, sound optimization, performance

Zusammenfassung

Numerische Software gibt es in vielen Formen: von Smartphones und tragbaren Geräten, die unseren Herzschlag und Schlafverhalten analysieren, bis hin zu komplexen Robotern, die Aufgaben im Weltraum ausführen, Simulationen, die Erdbeben vorhersagen, und viele mehr. Alle diese Programme werden auf Hardware mit begrenzten Ressourcen ausgeführt, daher müssen reellwertige Algorithmen durch Implementierungen mit endlicher Genauigkeit angenähert werden. Dies führt zu Abweichungen zwischen den idealen und den endlichen Präzisionsergebnissen, die als Rundungsfehler bezeichnet werden. Während einige Programme Fehler gut tolerieren, müssen andere die Berechnungen genau durchführen, da die Folgen eines falschen Ergebnisses teuer sind. Für solche Programme benötigt man eine Methode zur garantierten Beschränkung der Fehler.

Die heutigen Tools können garantierte Worst-Case-Rundungsfehler abschätzen. Die berechneten Fehlergrenzen werden verwendet, um eine Implementierung zu bewerten (z. B. ob sie genau genug ist), und um sie zu optimieren. Die Analysen lassen sich jedoch nicht gut auf große Programme übertragen und bieten nur begrenzte Unterstützung für Schleifen. Diese Einschränkungen gelten auch für Optimierungen; selbst für Basisblöcke haben sie nur schlichte Auswirkungen, wenn man Programme, die auf einer großen Eingabedomäne definiert sind, oder Funktionsaufrufe aus der mathematischen Bibliothek enthalten, optimieren möchte.

In dieser Arbeit verbessern wir statische Analysen und Optimierungen von numerischen Programmen auf verschiedene Weisen. Wir verbessern die Analysen, indem wir beschränkte und unbeschränkte Schleifen effizienter behandeln. Erstens, verbessern wir die Skalierbarkeit der Rundungsfehleranalyse bei beschränkten Schleifen über große Vektoren und Matrizen. Zweitens, stellen wir eine Methode vor, die strengere induktive Invarianten für unbeschränkte Schleifen schnell generiert, und eine breitere Anwendbarkeit als die besten bisherigen Methoden bietet.

Im zweiten Teil der Arbeit verbessern wir Optimierungen von geradlinigen numerischen Programmen. Wir schlagen eine Optimierung vor, die numerische Kernel durch Anpassung elementarer Funktionsaufrufe beschleunigt und gleichzeitig eine Garantie für den Gesamtfehler bietet. Außerdem führen wir eine Methode ein, die die bestehenden Optimierungen für Genauigkeit und Leistung verbessert, indem sie auf kleine Teile der (ursprünglichen) großen Eingabedomäne eines Programms angewendet werden, was zu aggressiveren Optimierungen führt.

Wir haben Open-Source-Prototypen für unsere Techniken entwickelt und sie anhand von Benchmarks aus den Bereichen wissenschaftliches Rechnen, maschinelles Lernen und anderen Bereichen bewertet. Unsere Ergebnisse sind vielversprechend und zeigen eine deutliche Verbesserung der Skalierbarkeit und Anwendbarkeit der Analysen sowie der zu optimierenden Kennziffern.

Schlüsselworte: Endliche Genauigkeit, numerische Software, statische Analyse, Gleitkommaarithmetik, Festkommaarithmetik, Rundungsfehler, Optimierung, Leistung

List of Original Publications

This thesis includes the content of these original publications:

1. Izycheva, A., Darulova, E., Seidl, H. Synthesizing Efficient Low-Precision Kernels. In: *Automated Technology for Verification and Analysis. ATVA 2019.*
   `https://doi.org/10.1007/978-3-030-31784-3_17`

2. Izycheva, A., Darulova, E., Seidl, H. Counterexample- and Simulation-Guided Floating-Point Loop Invariant Synthesis. In: *Static Analysis Symposium. SAS 2020.*
   `https://doi.org/10.1007/978-3-030-65474-0_8`

3. Rabe, R., Izycheva, A., Darulova, E. Regime Inference for Sound Floating-Point Optimizations. In: *ACM Transactions on Embedded Computing Systems*, vol.20, 5s, Special Issue ESWEEK 2021, Article 81 (EMSOFT 2021).
   `https://doi.org/10.1145/3477012`

4. Isychev, A., Darulova, E. Scaling up Roundoff Analysis of Functional Data Structure Programs. In: *Static Analysis Symposium. SAS 2023.*
   `https://doi.org/10.1007/978-3-031-44245-2_17`

Contents

1. Introduction

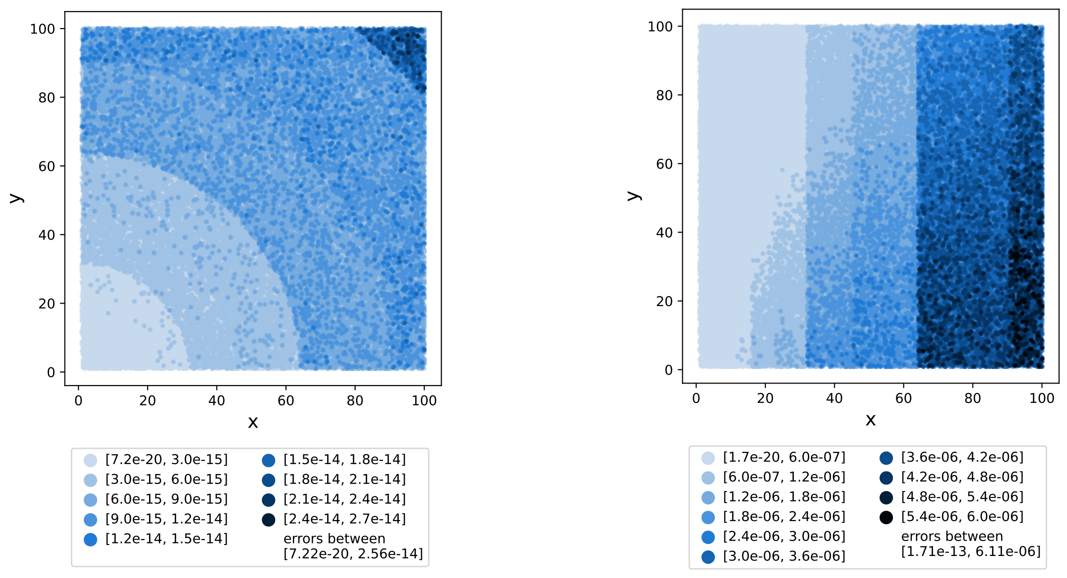
Numerical software has a large variety of applications: scientific computations [1], control of embedded systems and smart devices [2], modeling of physical and chemical processes [3, 4], financial applications [5, 6], machine-learning classifiers [7], and many more. Some of these numerical programs are used in high-risk scenarios, where the integrity of the hardware, or even a human life depends on the correctness of the computations and controls given by software [8, 9]. Ensuring correctness manually is difficult, therefore, it is essential to have tools that support developers in writing bug-free reliable numerical programs.

When implementing real-valued numerical algorithms in a program, developers have to decide on the numbers' representation. The tricky part is that computers only have a finite number of bits to represent infinite-precision reals. Therefore, real numbers must be approximated in some way. The choice of the finite-precision approximation will influence the programs' performance, accuracy, memory consumption, and code readability [10]. For instance, an algorithm implemented in 8-bit fixed-point precision is in general faster and requires less memory but is much less accurate than the same algorithm implemented in double floating-point precision.

Even for small programs, choosing the right finite precision is a non-trivial task. We illustrate it on an example. Consider a real-valued algorithm `cartesianToPolar_radius` that converts Cartesian coordinates into polar and returns the value of radius:

```
x∈[1.0, 100.0]
y∈[1.0, 100.0]
radius:= √(x² + y²)
```

When an algorithm is implemented in finite precision, it inevitably introduces rounding errors that accumulate and propagate through the computations, and that may significantly affect the result. We measure the absolute errors for `cartesianToPolar_radius` implemented in double floating-point precision on 10^5 random inputs and depict the errors in Figure 1.1a, darker colors denote larger errors. We then change the precision of just one input variable `x` from double to single floating point, sample the errors again and plot them in Figure 1.1b. This seemingly small adjustment has a dramatic effect on the distribution of errors. The largest errors originate in a different part of the input domain, and the overall error has increased by **seven** orders of magnitude from $[7.22e\text{-}20, 2.56e\text{-}14]$ to $[1.71e\text{-}13, 6.11e\text{-}06]$. Predicting this effect without measuring errors for each precision configuration is hard for an average user, and even for experts is next to impossible when a program is larger than a few lines [12].



(a) Double floating-point precision

(b) x with single floating-point precision

Figure 1.1.: Sampled absolute errors in different precision configurations for `cartesianToPolar_radius`. The errors are computed as the difference between an implementation with the corresponding precision configuration and a 300-bit implementation using the MPFR library [11].

Moreover, as Figure 1.1 illustrates, the errors are not evenly distributed over the input domain, and there are visible borders between sub-domains where the error jumps a magnitude. Such discrete behavior is natural for finite-precision functions, but unintuitive for users, and requires tooling to reason about finite-precision implementations.

Before explaining the contributions of this thesis, we summarize the state of the art in rounding error analysis and the optimizations based on these analyses. An overview is shown in Table 1.1, where our contributions are marked with the puzzle symbol. Sound rounding error analysis can be considered solved for numerical kernels (marked with '+') but is still work in progress for programs with loops (marked with '+/-'), such that even a partial solution of open challenges is a major advancement. Sound optimizations for performance and accuracy can be improved for straight-line code as well as programs with loops and branches (marked with '+/-'). In this thesis, we propose solutions to parts of these open problems and introduce our contributions in section 1.2 and section 1.3.

1.1. State of the Art

To improve the understanding of how algorithms behave when implemented in a particular finite precision, there has been a continuous effort to analyze [13–18], repair [19–21]

	Kernels	Loops	Branching
Analysis	+	🧩	+/-
Optimization	🧩	+/-	+/-

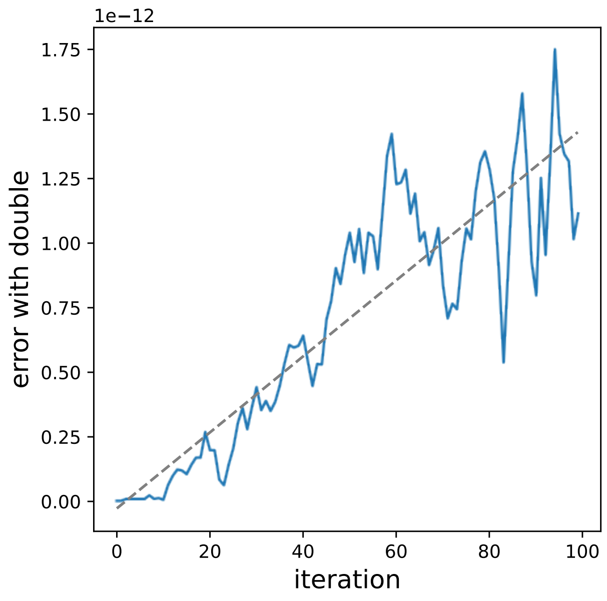
Table 1.1.: Overview of the state of the art in rigorous support for numerical programs. This thesis' contributions are marked with the puzzle symbol. State-of-the-art methods (outside of our contributions) are marked with '+' for solved problems; '+/-' marks problems that are only partially addressed and where there is room for improvement.

and optimize numerical programs [22–25]. Many of these methods focus specifically on floating-point programs, some support fixed points [13] and alternative finite precisions (e.g., Bfloat16, TensorFloat, posits) [26].

Analyses State-of-the-art analyzers of finite-precision programs can detect out-of-the-ordinary behaviors, such as division by zero, overflow, and operations with NaN (not-a-number) and infinities [27,28]. Beyond that, many analyzers compute *rounding error bounds* using dynamic [28–35] and static methods [13–18]. Dynamic analysis in its naive version is comparatively easy to perform. It can help estimate the magnitude of the values of variables and errors on them (as we did for plotting the error distribution) but does not provide any guarantees on the computed bounds, as it only reasons about a set of sampled values.

To rigorously reason about an implementation's safety and correctness, however, one needs to know *guaranteed (sound) worst-case* error bounds that are computed with static methods and take into account all possible values (of inputs, outputs and intermediate computations). Ideally, the computed bound should be close to the actual maximum error occurring in the program to avoid false-positive warnings about unsafe behavior. Given a worst-case bound and a maximum allowed (or expected) error one could check whether the implementation's error meets this specification or potentially exceeds it. The maximum allowed error may be derived from the sensitivity of actuators, or stability proofs for controllers [36].

State-of-the-art tools successfully compute sound bounds for errors in straight-line programs, however, they fall short when analyzing programs with conditionals and loops [13]. Because of the rounding errors in conditional statements (in branching or exit conditions for loops), the control flow can take different paths in the ideal real-valued computation and in the finite-precision one. The difference in computation results due to diverging paths is called a *discontinuity* error (alternatively, instability error). Some existing methods can quantify discontinuity errors on if-then-else statements [13,16,37] and compute the probability of the finite-precision computation taking the wrong path [38], however, they often report pessimistic bounds or do not scale well.



(a) Errors are within $[0.0, 1.75e - 12]$

```
for i in range(0,100):
  a[i] := randomValueIn(0.0, 100.0)

sum := 0

for i in range(0,100):
  sum := sum + a[i]
```

(b) Example program

Figure 1.2.: Sampled absolute errors for a sum over 100 numbers in the range of $[0.0, 100.0]$. The errors are computed as the difference between double floating-point precision and a 300-bit precision implementation using the MPFR library.

Moreover, even when the conditions do not include uncertainties, errors in loops are hard to quantify. Consider a simple example program that computes a sum over array elements and let us assume that each array element is a number from an arbitrarily selected range $[0.0, 100.0]$. Figure 1.2 shows how errors change in the first 100 iterations of the loop. The depicted error varies with every iteration, but the overall trend is increasing (shown as the dashed line). As the example illustrates, errors in loops are generally not bounded, and beyond that, the relation between the number of iterations and the overall error is non-trivial to compute [13].

State-of-the-art tools handle loops by reducing them to straight-line code [13,16,39] in one of the following ways. First, for some loops it is sufficient to analyze the loop body once. This is the case if the errors do not propagate through iterations, for instance, because every iteration reads a new sensor value. Secondly, loop iterations can be unrolled. This strategy works if the number of iterations is known or can be deduced statically. However, when loops have more than a few hundred iterations, the unrolled straight-line program can become prohibitively large, such that today's tools come to their limits and time out (as evidenced by our experiments in section 3.5). Thirdly, when loops cannot be unrolled one could abstract them using an inductive invariant. An inductive invariant captures the ranges of variables' values and errors on them before the first iteration and after every consequent iteration. Such invariants do not always exist, that is, not all loops converge to some stable state. And even for loops that do stabilize after a finite number of iterations, finding a non-trivial inductive invariant is challenging [40–43]. State-of-the-art tools attempt to simplify this hard problem by relying on user-provided templates [44, 45] and target value ranges [46], or finding invariants only for linear loops [47, 48].

In this thesis, we address some of these challenges in analyzing numerical loops. We discuss our contributions in section 1.2.

Optimizations Beyond programs' correctness and safety, which can be checked with computed error bounds, one may be interested in a program's efficiency. Numerical software is often used in a setting where resources are limited [49]. For instance, when building an embedded system the hardware computing control signals should work fast and reliably with a limited amount of memory and CPU power that fits on a small chip. For specialized hardware, like an FPGA, the size of the chip needed to execute a program is directly influenced by the program's complexity and size. Hence, it is crucial to keep programs small such that they could be deployed on a smaller chip. Moreover, the calculated control signals must be available when they are needed, thus, computed quickly. Last but not least, the computed control must be accurate enough for the smooth functioning of the whole system.

Each of these objectives is tied to one another and cannot be considered in isolation. Using high finite precision to increase accuracy may cause a performance drop [22], using mathematical libraries to decrease the size of the code may increase the performance for some programs and decrease it for others, depending on the library implementation and available precisions [50].

When optimizing numerical programs for any objective, it is important that the results computed by the program remain meaningful, i.e. the errors potentially introduced by the optimization must stay bounded. Prior work on optimizing numerical kernels uses error bounds computed with both dynamic analysis [19–21, 51] and sound static methods [22, 23, 25, 52] to guide the optimizations. In the scope of this work, we focus only on sound optimizations, i.e. with respect to sound error bounds.

A popular choice for an optimization objective is performance, which can be improved soundly, for instance, with *mixed-precision tuning* [22, 23, 25]. Executing programs with low precision is generally faster than with high precision, and many arithmetic operations can tolerate some accuracy loss. Mixed-precision tuning finds a fast and *accurate enough* precision configuration, where operations and variables are potentially assigned different precisions (hence, mixed precision). This approach is implemented in several sound optimizers and shows significant improvements in running time, especially when the programs can tolerate relatively large errors [22, 23, 25].

Orthogonally to performance, one can optimize the accuracy of numerical programs. Existing sound accuracy optimization tools make use of the fact that finite-precision arithmetic does not obey the real-valued rules and different orders of evaluation result in different errors [22, 52]. The rewriting optimization rearranges the original arithmetic expression according to real-valued identity rules and selects an expression with the smallest rounding error.

In the scope of this thesis, we explore how today's sound optimizations can be improved and extended. A more detailed account of our contributions is given in the following sections.

1.2. Contributions in Analysis of Programs with Loops

In this thesis, we improve analyses of bounded loops over large arrays and numerically stable unbounded loops.

Contribution 1: Bounded loops. Loops over arrays appear frequently in embedded systems [53], statistical computations [54] and image processing [55, 56]. When arrays are small, loops over them can be easily unrolled and analyzed by state-of-the-art tools, however, for large arrays full unrolling is not feasible: tools take too long, time out or return a trivial error bound of $[-\infty, \infty]$ as a result (as confirmed by our experiments in section 3.5).

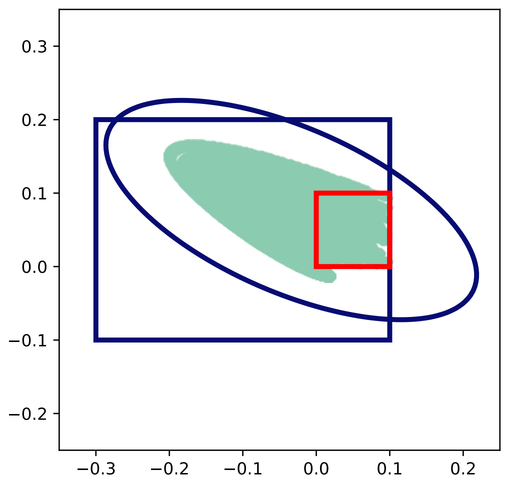
In this work, we scale up rounding error analysis on loops over vectors and matrices. Our key observation is that *functional* iterators carry implicit semantic information about how data flows through the iterations. For instance, when applying a functional `map(`$\lambda\, x.f(x)$`)`, the function f is applied to each element of the data structure separately. Thus, the values and the errors on them do not propagate to subsequent iterations, which means that it is sufficient to analyze the iterator's body once (per input range of x). State-of-the-art tools would treat a `map` as any other loop and unroll it, effectively losing the information about data flow and unnecessarily repeating the computations.

We propose to use the implicit semantic information from functional iterators to reduce the re-computation of ranges during the analysis. To facilitate this we propose a functional domain-specific language (DSL) that specifies higher-order functions over real-valued vectors and matrices. Additionally, we introduce an abstraction that partitions data structures' elements based on their ranges. Combining the semantic information about iterators with the data structure abstraction, our analysis computes sound error bounds faster and for more programs than the state-of-the-art tools [39, 57].

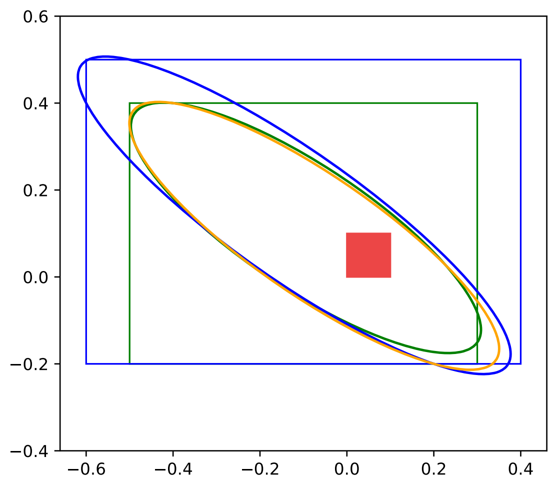
For instance, given a simple program that computes an average of 10K vector elements, state-of-the-art analyzers Fluctuat [39] and Satire [57] take 8.5 and 24 minutes, while our approach reports error bounds in 2 seconds. Moreover, the error bound computed with our approach for the example program is of the same order of magnitude as the ones computed with Fluctuat and Satire (10^{-11}).

Contribution 2: Unbounded loops. Unbounded loops cannot be fully unrolled and have to be abstracted, for instance, by an inductive invariant. Importantly, finite-precision inductive invariants only exist for *numerically stable* loops, for which a small error on an input or an intermediate value does not cause a major change in the computation result.

Given an invariant, one can prove its inductiveness by showing for all values in the invariant range $I(x)$ that after one iteration of a loop $L(x)$ the resulting values x' are still



(a) An inductive invariant at the intersection of the blue box with an ellipse. Green points are simulated values of the loop.



(b) Multiple inductive invariants for the non-linear dynamic system. Each color corresponds to one invariant.

Figure 1.3.: Inductive invariants of the non-linear dynamic system. Red boxes mark the input range $x \in [0.0, 0.1], y \in [0.0, 0.1]$.

in that range: $I(x) \wedge L(x) \rightarrow I(x')$. The exact representation of ranges in I may affect whether or not the inductiveness can be proven. Ideally, $I(x)$ should only include the values that actually appear during the loop execution.

For instance, consider the following non-linear dynamic system with starting values of x and y in $[0.0, 0.1]$:

```
while (true) {
  x := x + 0.01 * (-2*x - 3*y + x*x)
  y := y + 0.01 * (x + y)
}
```

The simulated values of the first 10K loop iterations, shown in green in Figure 1.3a, indicate that the invariant has a shape of an ellipsoid. If we represent the ranges as a set of intervals (a blue box), it will include additional values that do not appear during the loop executions (white space inside the box). Proving that $x \in [-0.3, 0.1] \wedge y \in [-0.1, 0.2]$ is an inductive invariant fails, but if we remove (some of) the values that do not occur in the loop with an additional polynomial inequality constraint, we can prove that the following invariant is inductive:

$$-0.03x - 0.13y + 0.35x^2 + 0.7xy + y^2 \leq 0.01 \wedge x \in [-0.3, 0.1], y \in [-0.1, 0.2]$$

We propose a method that automatically generates an inductive invariant in the form of a polynomial inequality $\mathcal{P}(x) \leq 0$ and a set of ranges $\mathcal{R}_i$ for each program variable x_i:

$$\mathcal{P}(x_1, ..., x_n) \leq 0 \wedge x_1 \in \mathcal{R}_1 \wedge \ldots \wedge x_n \in \mathcal{R}_n \tag{1.1}$$

From such invariants, one may compute rounding error bounds using complementary techniques [13]. In this thesis, we focus on generating a finite-precision inductive invariant itself and leave the error computation to future work.

A key observation behind our invariant synthesis method is that each loop has more than one invariant, and we only need to find one of them. Our method generates an invariant using a form of counterexample-guided synthesis (CEGIS) that proposes a candidate invariant and repeatedly refines it using simulation and counterexamples. We implemented the algorithm in an open-source Python library Pine. Pine requires minimal input from users and, unlike state-of-the-art tools, derives tight invariants for *both linear and non-linear* loops. On linear loops Pine generated invariants that are on average 20x and 2.7x tighter than the state-of-the-art invariant generators SMT-AI [47] and Pilat [48].

1.3. Contributions in Optimization of Numerical Kernels

Since analysis of complex programs is still an open problem, we focus on optimizing straight-line numerical programs (kernels), for which sound analysis already exists. Straight-line kernels appear frequently in embedded systems design [58], machine learning models [7], etc. Optimizing kernels that are frequently called by the rest of a program may bring significant improvements in performance and accuracy to the overall results.

Contribution 3: Performance of elementary functions. Existing performance optimizations, such as mixed-precision tuning, handles arithmetic operations well but has limited effects on library function calls. When implementing an algorithm that contains an elementary function (sine, exponent, logarithm, etc.), most applications rely on mathematical libraries, which implement each function only in a limited number of precisions. Moreover, library implementations of elementary functions are designed to produce accurate results on all inputs (modulo restrictions of the function itself, such as only allowing non-negative inputs for a logarithm). At the call site, however, function arguments can be limited to a rather small range and will not use the "full power" of the library function. An implementation customized to a smaller range may therefore be faster.

Let us illustrate the effect of elementary function calls with an example program. The algorithm `forwardk2jX` converts the angles of two joints of a robotic arm into the X-axis coordinate:

```
theta1 ∈ [0.01, 1.5]
theta2 ∈ [0.01, 1.5]
x:= 0.5 * cos(theta1) + 0.5 * cos(theta1 + theta2)
// allow max error 8.39e-07
```

Assume that a user has specified the maximum allowed error on this program to be $8.39e-07$ and wants to generate a fast implementation with fixed-point precision. The state-of-the-art tool Daisy [14] determines that 25 bits are sufficient to satisfy the target error with uniform precision, and with tuning it further lowers the precision of two constants (while the rest stays the same). Compiled with the Vivado High-Level-Synthesis (HLS) tool [59] for an FPGA chip, the kernels with both uniform and mixed precision compute the value of `x` in **64 machine cycles**. Unfortunately, mixed-precision tuning failed to improve the performance of `forwardk2jX`.

Our key observation is that mathematical library function calls can be unnecessarily accurate. Both implementations of `forwardk2jX` (uniform and mixed-precision) call the library function `cos()` from `hls_math.h` that produces accurate results for all possible inputs. However, the maximum range of all arguments to the `cos()` occurring in the program is $[0.01, 3.0]$, which is comparatively small, thus, the implementation of `cos()` only needs to be accurate for this small input range.

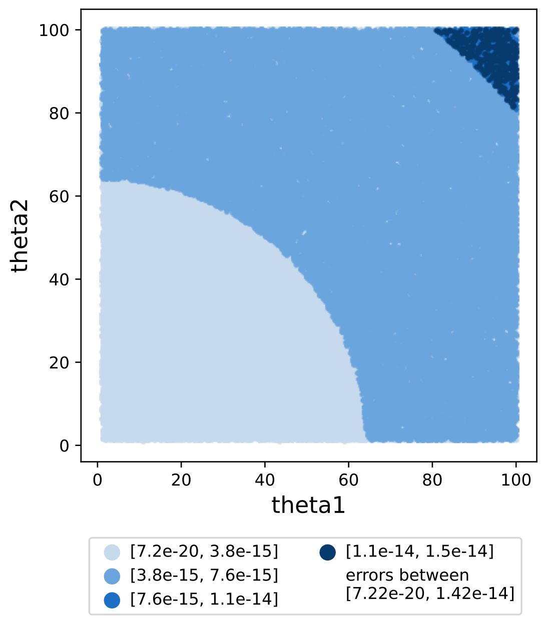
In this thesis, we propose a method that replaces elementary function calls with polynomial approximations. We tailor the approximations to the input ranges, thus trading off accuracy for performance more efficiently than with mixed-precision tuning alone. The user only has to specify the maximum allowed error for the whole program. When a program contains multiple elementary function calls, our algorithm automatically assigns a portion of the overall tolerated error to each call before generating an approximation. Our method is not specific to a particular choice of finite precision, in this work, we focus on fixed points to allow fine-grained tuning.

For the example program `forwardk2jX` our algorithm generates two approximate versions of `cos()`, for inputs in $[0.01, 1.5]$ for the first call of `cos()`, and $[0.02, 3.0]$ for the second. Our final implementation of `forwardk2jX` with the approximations finishes in **24 machine cycles** (computed with the Vivado HLS tool), which is 2.6x faster than the program using the library implementation of `cos()`.

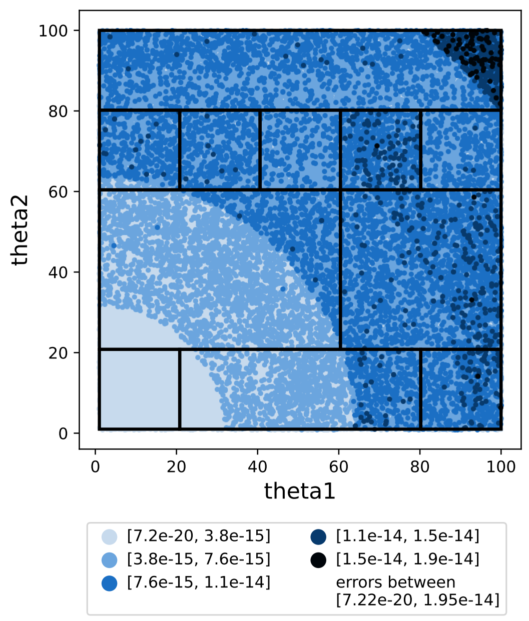
Contribution 4: Meta-Optimization. Existing sound optimizations—independent of the objective—attempt to find a "one-size-fits-all" implementation that works for the whole input domain. However, different parts of the input domain induce errors of different magnitudes and may allow different optimization strengths.

Recall the error distribution of the example program `cartesianToPolar_radius` in Figure 1.1a. When optimized for performance, today's mixed-precision tuning will try to find a precision assignment that works on the whole domain. However, the top-right corner with values of `x,y` close to `100` induces higher errors than the rest of the domain and will therefore require higher precision. Since mixed-precision tuning optimizes with respect to worst-case error, given a maximum allowed error of 2.5e-14 all operations are assigned a uniform quad (128-bit float) precision even if it is unnecessary for most of the inputs.

Our key observation is that small parts of the input domain may potentially block sound optimizations. We propose to tailor sound optimizations to smaller parts of the input domain and call this meta-optimization *regime inference* [19]. We show that our regime inference improves optimization results for both performance and accuracy optimizations also for numerically stable code, i.e. without irregularly large errors. For instance, when applied to `cartesianToPolar_radius` and the same target error of 2.5e-14



(a) Optimized with mixed-precision tuning - uniform quad precision



(b) Regime inference with mixed-precision tuning

Figure 1.4.: Sampled absolute errors for the optimized versions of `cartesianToPolar_radius`. The maximum allowed error for both optimizations was set to 2.5e-14. The errors are computed with respect to the 300-bit MPFR implementation.

with existing mixed-precision tuning, our regime inference splits the input domain into 11 parts, each of which is assigned one of four generated precision configurations. We plot the sampled errors for the program optimized with mixed-precision alone and with regimes on top of mixed-precision tuning in Figure 1.4. Note the difference in error behavior, for regime-based implementation there is a clear increase in error between some sub-domains (for instance, with $theta1 \in [60, 80], theta2 \in [60, 80]$). The optimized program with regimes runs 80% faster than the one optimized with mixed-precision tuning alone.

1.4. Outline

This thesis extends rounding error analysis to handle programs with loops and improves sound optimizations of numerical kernels. We have implemented these techniques in research prototypes, evaluated them on (partially overlapping) sets of benchmarks, and published them as open-source tools. The thesis is organized in two parts: analysis and optimization. Precisely, the chapters are organized as follows:

Chapter 2 provides necessary background on finite-precision formats and existing techniques for analyzing and optimizing finite-precision programs. We explain the rounding error analysis and optimizations that we use as a starting point, and build on top of these techniques in the later chapters.

Chapter 3 presents a scalable rounding error analysis of bounded loops over vectors and matrices.
The work from this chapter is based on our publication at SAS'23 [60]. We have performed an additional evaluation after submitting the paper and included it in subsection 3.5.4.

Chapter 4 presents our inductive invariant synthesis algorithm for numerically stable unbounded loops.
This chapter is based on our publication at SAS'20 [61]. We implemented the algorithm in a Python library PINE available at `https://github.com/izycheva/pine`.

Chapter 5 describes our performance optimization where elementary function calls are replaced by approximations.
The content of this chapter is based on our publication at ATVA'19 [50]. Source code for our implementation is available under `https://github.com/malyzajko/daisy/tree/approx`.

Chapter 6 describes our regime inference technique—a meta-optimization for sound optimizations of finite-precision programs.
This chapter describes a joint work with a Bachelor student that has been published at EMSOFT'21 [62]. My contribution to the publication includes generating conceptual ideas for our method, providing support to the student when he had questions regarding the implementation, and formulating the ideas into a coherent text. The code for our implementation is available under `https://github.com/malyzajko/daisy/tree/regimes`.

This content of the thesis is based on the work published in peer-reviewed conferences in the original papers listed in the preamble. I am the main author and a co-author in all papers[1], however, I will use the academic "we" throughout the thesis, because the publications would not be the same without the work and support of my collaborators.

[1]In the course of my studies I have (officially) changed the spelling of my name from Anastasiia Izycheva to Anastasia Isychev, some publications list me as an author with the old spelling.

2. Preliminaries

Before explaining the contributions of this thesis in detail, we provide the necessary background information. In particular, we summarize the representations of floating- and fixed-point numbers and the state of the art in rounding error analysis and optimization.

2.1. Number Systems

Most numerical algorithms are designed with real-valued arithmetic in mind. However, implementing real numbers on hardware with a finite number of precision bits, developers must decide on a finite representation.

There exist many alternative representations of numbers (or number systems) that present a trade-off between precision with which numbers can be expressed using the number system, and its efficiency when the code is being executed.

We review the two most frequently used number systems in numerical software: floating-point and fixed-point numbers. While there exist alternative representations, they are not as well-known and are used comparatively rarely [63]. We review the alternative representations in chapter 7 and note how our methods can be used with these representations. We start with the most popular choice for representing real numbers in finite precision—floating points.

2.1.1. Floating-Point Numbers

Floating-point numbers are used widely by developers with different levels of expertise. What makes floating points so attractive is their simplicity from a user perspective. They allow to represent a (relatively) large range of values, and most languages support floating-point arithmetic operations as well as a variety of mathematical libraries. Additionally, the behavior of floating-point operations is standardized and language-agnostic. The IEEE-754 standard [64] describes how floating points should be implemented in software and hardware, how the arithmetic operations should be evaluated, and how real values and results of the operations must be rounded.

On a bit level, a floating-point value appears as:

$$sb \;\; e_0 \dots e_k \;\; m_0 \dots m_l, \tag{2.1}$$

where sb is a sign bit, e_i are bits used for the exponent, and m_i are the significand (or mantissa) bits.

Floating points include several special values, which are used when arithmetic operations on the operands cannot be evaluated to a number (NaN, not a number), if the

	single	double	quad
total bits	32	64	128
precision bits (p)	24	53	113
e_{max}	127	1023	16383
e_{min}	-126	-1022	-16382

Table 2.1.: Binary floating-point precision parameters

value is larger or smaller than what precision can represent ($\pm\infty$), and if the value is so close to zero that it requires a different encoding (subnormal numbers). Each of these values has an individual pattern in the exponent bits to signal that the represented value is special. However, the majority of values represented in floating points are normal numbers.

A normal floating-point number represented by a triple (sb, e, m) encodes a unique value:

$$(-1)^{sb} \cdot m \cdot \beta^{e-p+1}. \tag{2.2}$$

Here, β represents the base of the encoding, and p stands for precision. The IEEE-754 standard defines floats for two possible bases $\beta = 2$ and $\beta = 10$, in this work we will consider only binary floating-point numbers (with the base 2), as they are used much more frequently than the decimal. The standard lists three "basic" precisions: single (32 bits), double (64 bits) and quad precision (128 bits), an IEEE-754-conform language must implement at least one of them. Each floating-point precision is additionally specified by the number of precision bits p and the maximum and minimum exponent values e_{max} and e_{min}. Table 2.1 shows values of p, e_{max} and e_{min} for the three standard precisions.

Apart from the bit-level representation, an important part of operating with floats is *correct rounding*. According to IEEE-754, a result of an operation is said to be correctly rounded if it represents the value obtained by first performing the operation with infinite precision and then rounding it to the floating-point precision with a given rounding mode. In this thesis, we consider the default rounding "ties-to-even"—if the infinite-precision value is equally near to two floating-point values, the one with an even least significant digit will be selected.

Floating-point arithmetic can be efficiently executed on modern hardware thanks to special floating-point units (FPU). Without such a unit, however, the conversion of bits into rounded values has to be simulated in software (often the case for the quad precision), which is costly.

2.1.2. Fixed-Point Numbers

In the absence of floating-point units, for instance, on accelerators, developers may choose alternative number systems. Fixed-point numbers provide a good alternative.

While for floating-point numbers the position of the binary (or decimal) point changes depending on the magnitude of the value (hence, floating point), in fixed-point numbers the binary point has a fixed position. We call a total number of bits used to represent a number a fixed-point *precision*, and the distribution of bits between integer and fractional parts a *format*. A fixed-point format is usually denoted as $\langle W, I\rangle$, where W is the total number of bits, one bit is reserved for the sign, I for the integer part, and remaining $F = W - 1 - I$ for the fractional part [65]. Because of this simple internal representation fixed-points arithmetic can be efficiently executed by the hardware: arithmetic operations on numbers with matching fixed precision are implemented the same way as on integer numbers.

Another advantage of fixed-point numbers is their efficient usage of bits. While floating points are limited to three standard precisions, and have a fixed number of bits dedicated for mantissa, any positive number of bits can represent a valid fixed-point precision. For programs operating on small ranges of values, using floating points will effectively waste resources, as many of the bits will remain unused. Fixed points, however, are more flexible: every operation can be implemented with as many bits (total and fractional) as needed, which gives developers manual control over each operation.

However, manual control makes fixed-point programs less approachable by non-expert users. The binary point, separating integer and fractional parts, must be fixed for each intermediate value at compile time and cannot be dynamically changed. Therefore, each variable and operation must be assigned a suitable precision and format in advance such that values do not overflow and are sufficiently accurate. In order to do that, a user has to know (or estimate) the potential values of all intermediate computations, which is hard to do manually, so a user may assign higher precision(s) than actually needed. Moreover, to facilitate efficient usage of bits, variables in the same program may have different formats (number of integer and fractional bits). Whenever the formats of two operands of an arithmetic operation do not match, the operands must be aligned with respect to the binary point before performing the computation. Writing such fixed-point programs manually requires a high level of numerical expertise, it is tedious and error-prone. To automate code generation for fixed-point programs, value and rounding error analyses are required [49]; we review them in the next section.

As all finite representations of infinite numbers, fixed-point formats have a rounding mode and an overflow behavior. In this work, we consider the default modes for both: truncation rounding and wrap-around overflow. Note that programs generated with our techniques will not have an overflow, because the analyses presented in this thesis will detect it and assign a suitable higher number of bits for precision.

2.2. Reasoning About Finite-Precision Programs

When representing real numbers, all finite precisions inherently introduce rounding errors. The individual errors accumulate and propagate throughout the computations. When accuracy of the computed result is crucial, one must ensure that the overall error is small enough.

A corresponding maximum error occurring in the program can be computed statically or dynamically. *Dynamic* methods sample the program under analysis on multiple valid inputs and check the computation results [28,32,33,35,66]. Since real-valued results are usually not available, dynamic analysis computes errors with respect to an oracle, usually some arbitrary high precision that simulates the real result. Such methods allow for a quick estimate of error magnitude, however, they only reason about *seen inputs* and not *all possible valid* inputs. As a consequence, dynamic methods alone are insufficient to reason about safety-critical programs where it is crucial to have a guaranteed error bound.

Whenever a finite-precision implementation requires *guarantees* on errors, one must employ *rigorous* analysis and optimization methods. In this thesis, we extend rigorous support for finite-precision programs with both analysis and optimization techniques. Rigorous support takes into account all valid inputs, where the validity of the inputs is determined by whether the input value is in the user-specified range. In order to compute with ranges one needs an appropriate range arithmetic.

2.2.1. Range Arithmetic

We briefly review different range representations with real numbers and how arithmetic operations are defined for them. Note that these arithmetics are usually employed by analysis to obtain the results of numerical function evaluation. We therefore assume each variable is mapped to a range in one of the representations, and use notations "operations on ranges" and "operations on variables" interchangeably.

Interval Arithmetic The most basic and intuitive way to represent ranges is with intervals. An interval $[l, u]$ denotes all values x such that $l \leq x \leq u$ [67]. An interval for which lower and upper bounds coincide $[l, u], l = u$ is called a point interval. Evaluation of arithmetic operations is done by computing the interval bounds and follows a general pattern:

$$\begin{aligned} [l_1, u_1] \circ [l_2, u_2] &= [\min_{l \in R} l, \max_{u \in R} u] \\ \text{where } R &= \{l_1 \circ l_2, l_1 \circ u_2, l_2 \circ l_1, l_2 \circ u_1\} \end{aligned} \tag{2.3}$$

where an operator $\circ \in \{+, -, *, /\}$ is applied to pairs of lower and upper bounds and the largest interval among all combinations is the result. Similarly, unary operations are applied to both lower and upper bound of the argument.

Unfortunately, interval arithmetic often gives pessimistic results. The simple operations evaluation does not keep track of the correlations between variables and their corresponding intervals, which results in over-approximation. For instance, for a variable $x \in [l, u]$ and a function $f(x) = x - x$ the result of the function evaluation is always zero. However, in interval arithmetic subtracting a range from itself will give a different over-approximated result $[l, u] - [l, u] = [l - u, l + u] \neq [0, 0]$.

Despite its "bad reputation" interval arithmetic is easy to implement, and fast to evaluate, which provides a good trade-off between accuracy and performance, which is why it is frequently used by static analysis tools [13,14,27].

Affine Arithmetic When using intervals gives insufficiently accurate results, ranges can be represented in different ways. One alternative is using affine forms [68]:

$$\check{x} = x_0 + \sum_{i=1}^{n} x_i \epsilon_i, \text{ with } \epsilon_i \in [-1, 1] \tag{2.4}$$

where x_0 is the center of the range, and affine coefficients x_i define the magnitude of the i-th noise term around the central value. Range bounds of an affine representation in Equation 2.4 can be computed as:

$$[\check{x}] = [x_0 - \sum_{i=1}^{n} |x_i|,\ x_0 + \sum_{i=1}^{n} |x_i|]. \tag{2.5}$$

Linear operations are evaluated on pairs of terms, central values and the corresponding i-th term ϵ_i for each operator. For a pair of affine ranges $\check{x}$ and $\check{y}$:

$$\alpha\check{x} + \beta\check{y} = (\alpha x_0 + \beta y_0) + \sum_{i=1}^{n} (\alpha x_i + \beta y_i)\epsilon_i \tag{2.6}$$

Because of the noise terms ϵ_i, that can be shared between the operands, affine arithmetic can track linear correlations. The same example function, for which intervals were imprecise, $f(x) = x - x$ in affine arithmetic will be evaluated exactly to zero: $(x_0 + x_1\epsilon_1) - (x_0 + x_1\epsilon_1) = (x_0 - x_0) + (x_1\epsilon_1 - x_1\epsilon_1) = 0$.

Affine representation is essentially a linear combination of terms, which is why linear operations can be evaluated exactly. Non-linear operations, however, cannot be exactly represented as a linear combination of source terms x_0 and $x_i\epsilon_i$ and must be approximated [68]. For instance, a multiplication of two affine ranges $\check{x}$ and $\check{y}$ is defined as:

$$\check{x} \cdot \check{y} = x_0 \cdot y_0 + \sum_{i=1}^{n} (x_0 y_i + y_0 x_i)\epsilon_i + \eta\epsilon_{n+1} \tag{2.7}$$

where $\eta\epsilon_{n+1}$ is a fresh error term that bounds the difference between inherently non-affine $x \cdot y$ function and an affine approximation of it. The freshly introduced term can be computed in different ways, however, it generally loses correlations with the source terms and will be propagated in the subsequent computations. Other non-linear operators are evaluated in a similar fashion [69].

Note that due to the potentially expensive evaluation of the approximate affine parameters compared to simple intervals, intervals may be a better choice for estimating ranges of non-linear operations.

Both interval and affine arithmetic are frequently used to represent variables' ranges in rigorous rounding error analysis. In the remainder of this chapter, we review existing rigorous methods to estimate the worst-case rounding error bounds, and provide details on sound finite-precision optimizations guided by these analysis methods.

2.2.2. Rounding Error Analysis

The worst-case rounding errors can be specified in absolute or relative terms. For a real input x and a result of a real-valued computation $f(x)$, we define the finite-precision version of this input as $\hat{x}$ and the result of the corresponding finite-precision computation as $\hat{f}(\hat{x})$. Then, the worst-case *absolute* rounding error is:

$$e_{abs} = \max_{x \in I} |f(x) - \hat{f}(\hat{x})| \tag{2.8}$$

and the worst-case *relative* rounding error is:

$$e_{rel} = \max_{x \in I} \left| \frac{f(x) - \hat{f}(\hat{x})}{f(x)} \right| \tag{2.9}$$

for inputs x in some range I.

While relative errors may seem a more meaningful measure of the implementation's quality, they are only defined for algorithms where zero does not appear as a possible value of $f(x)$. There exist some methods to bound relative errors [70], however, the majority of state-of-the-art tools focus on absolute errors. For fixed-point numbers absolute errors are the only meaningful measure, because the number of bits for the fractional part is fixed at compile time, therefore, the worst-case error is the same for all values using the same fixed precision. In the rest of the thesis, we will only consider absolute rounding errors and will refer to them simply as rounding errors.

Equation 2.8 measures the difference between a real value and a finite-precision value directly, however, such a measurement is difficult to perform for a number of reasons. First, the real-valued evaluation of the algorithm is not available for most inputs and algorithms. Secondly, the finite-precision function $\hat{f}(\hat{x})$ is highly discontinuous due to the necessary rounding, since not every real number can be directly represented in finite precision. Instead of bounding the absolute error expression from Equation 2.8 directly state-of-the-art tools replace $\hat{f}(\hat{x})$ with an abstraction.

For each basic arithmetic operation ($\circ \in \{+, -, *, /\}$ and $\sqrt{}$) its finite-precision counterpart $\hat{\circ}$ can be modeled as a noisy version of the original operation. Given two real operands x and y and their finite-precision versions $\hat{x}$, $\hat{y}$, the following holds:

$$\hat{x} = x(1 + e) + d \tag{2.10}$$

$$x \hat{\circ} y = (x \circ y)(1 + e) + d \tag{2.11}$$

$$\hat{\sqrt{x}} = \sqrt{x}(1 + e) + d \tag{2.12}$$

where $|e| \leq \epsilon_M$, and the machine epsilon ϵ_M represents the maximum relative error introduced by rounding at each operation. The value of ϵ_M depends on the number of bits used for the fractional part of the finite-precision precision. For instance, for signed fixed-point format $\langle W, I \rangle$ $\epsilon_M = 2^{I+1-W}$. For floating points the value is determined with the number of bits used for significand, $\epsilon_M = 2^{-24}$ for single precision, $\epsilon_M = 2^{-53}$ and $\epsilon_M = 2^{-113}$ for double and quad precisions.

The second error abstraction component $|d| \leq \delta$ is only required for floating-point numbers, and δ represents the maximum absolute error due to rounding on subnormals, for fixed-point precisions $d = 0$.

Rewriting 2.10 to obtain the error term on the right-hand side of the equation we obtain $\hat{x} - x = x \cdot e + d$. Similarly, for other operations, the rewritten equations clearly show that the magnitude of an error directly depends on the magnitude of the values x and y. Therefore all errors are computed only with respect to a given range for input variables, the intermediate ranges are inferred using range arithmetic.

Using equations 2.10-2.12, a finite-precision function $\hat{f}(\hat{x})$ can be abstracted as $f(x, e, d)$ by chaining the individual operations' abstractions. Note that this abstraction is only valid for (sub-)normal numbers, i.e., for all fixed-points and for floating-points exclusively in the absence of infinities and NaNs. State-of-the-art tools detect the presence of these special values and abort the error bound computation if they are detected. For normal numbers the rounding error ε for inputs $x \in I$ is bounded with:

$$\varepsilon \leq \max_{x \in I, |e| \in \epsilon_M, |d| \in \delta} |f(x) - f(x, e, d)| \tag{2.13}$$

State-of-the-art tools use Equation 2.13 with some variations in the objective function. There are two principally different approaches to bound ε: data-flow static analysis and a global optimization-based approach.

Data-flow Error Analysis The worst-case error on the program can be estimated using forward static analysis. Static analysis traverses the abstract syntax tree (AST) of an input program and evaluates each node with transfer functions defined on some abstract domain. Because in rounding error analysis we are interested in obtaining error bounds, and the inputs are specified as ranges, the abstract domain of choice must represent ranges of values (respectively, errors) for all nodes of the AST.

Data-flow error analysis is usually done in two phases:

1. compute *real-valued ranges* of the operation result, and
2. compute *rounding error ranges*. This parts includes the error committed by the latest evaluated operation, as well as errors propagated from preceding computations.

While it is possible to compute ranges of finite-precision operations directly, it is afterward hard to distinguish which part of the range is due to errors. Computing the ranges separately makes this distinction more transparent and reduces the over-approximation of the propagated errors. Additionally, separate computation allows a better efficiency/accuracy trade-off for the analysis as a whole: small error ranges can be expressed using a precise and less efficient abstract domain, for instance, affine representation [68], while (potentially) large value ranges can be tracked using more efficient and less precise interval domain [67].

Both ranges and errors are usually computed in parallel for each node of the abstract syntax tree. From the real-valued range and the propagated error range the analysis determines the actual range of values that may appear in the finite-precision AST node. For fixed-point programs this finite-precision range of each node is used to determine the number of bits needed for integer and fractional parts.

The dataflow analysis tools differ from one another in several aspects. First, analyzers use different abstract domains to represent value and error ranges. For example, Rosa [13] and Daisy [14] use interval arithmetic for range computations and affine arithmetic for errors, while Fluctuat [39] uses zonotopes (higher-dimension affine representation) for both.

Secondly, analyzers employ different techniques to tighten computed ranges. Having tight estimates on value ranges is a prerequisite for tight error bound computation. It is especially important in the presence of non-linear operations, for which the over-approximation is large with both interval and affine domains. Fluctuat uses range sub-division and Taylor expansion (for unary non-linear operations). Both Rosa and Daisy support sub-division as well and employ a complementary technique that uses SMT solver to gradually tighten the bounds obtained with intervals.

Global Optimization-Based Analysis

Bounding the rounding error with Equation 2.13 can also be solved as an optimization problem [16,17,57,71]. For small finite-precision expressions, it is possible to compute the maximum of the Equation 2.13 directly. However, for larger programs, the size of the objective function becomes intractable for modern solvers. Therefore, optimization-based tools relax the optimization objective.

FPTaylor [71] has introduced *symbolic Taylor expansions*, that uses Taylor approximation to simplify $f(x,e,d)$. For functions that are differentiable twice on some open area in the domain of valid inputs x, the finite-precision function $f(x,e,d)$ can be approximated with:

$$f(x,e,d) = f(x,0,0) + \sum_{i=1}^{k} \frac{\partial f}{\partial e_i}(x,0,0)e_i + R(x,e,d) \tag{2.14}$$

where

$$R(x,e,d) = \frac{1}{2}\sum_{i,j=1}^{2k} \frac{\partial^2 f}{\partial y_i \partial y_j}(x,p)y_i y_j + \sum_{i=1}^{k} \frac{\partial f}{\partial d_i}(x,0,0)d_i$$

and $y_1 = e_1, \ldots, y_k = e_k, y_{k+1} = d_1, \ldots, y_{2k} = d_k$ and $p \in \mathbb{R}^{2k}$ such that $|p_i| \leq \epsilon_M$ for $i = 1 \ldots k$ and $|p_i| \leq \delta$ for $i = k+1 \ldots 2k$. The remainder term R bounds all higher order terms and ensures soundness of the computed error bounds.

A term $f(x,0,0)$ denotes an exactly rounded function $f(x)$ without errors $e = d = 0$. With this, the objective function $|f(x) - f(x,e,d)|$ is further simplified, and the optimization task becomes:

$$\varepsilon = \max_{x \in I, e \leq |\epsilon_M|, d \leq \delta} \left| \sum_{i=1}^{k} \frac{\partial f}{\partial e_i}(x,0,0) e_i \right| + M_R, \tag{2.15}$$

where M_R is an upper bound on the second-order term $R(x,e,d)$. with respect to inputs $x \in I$ and terms e, d bounded for a particular precision.

FPTaylor bounds Equation 2.15 with two procedures: interval arithmetic for M_R and branch-and-bound optimization for the maximization term. A recent tool Satire [57] uses the same approach and extends it with further abstractions to scale the analysis for larger programs (though still limited to straight-line code).

Similar to FPTaylor and Satire, real2Float [17] uses symbolic Taylor expansion to generate the maximization objective and splits the error into two parts (first- and second-order terms). Unlike other tools, real2Float uses a semi-definite programming optimization technique to solve the maximization problem.

PRECiSA [16] formulates the optimization objective in a slightly different fashion: it uses the unit in the last place (*ulp*) to express the error on individual operations. All operations complying with IEEE-754 must be correctly rounded, which means that for the default rounding mode (to nearest) the maximum difference between floating-point $\hat{x}$ and real x is a half of the distance between two neighboring finite-precision values. This distance is quantified by the unit in the last place. Thus, the error on a floating-point computation is:

$$\varepsilon = \max_{x \in I} \epsilon_f(e) + \frac{1}{2} ulp(f(x \pm e)), \tag{2.16}$$

where $\epsilon_f(e)$ computes the function f on the propagated errors, and $1/2ulp(f(x \pm e))$ estimates the error committed by the latest operation. Here, the function $f(x \pm e)$ denotes a non-negative real-valued expression that over-approximates the application of real f to rounded values of x. The terms $\epsilon_f(e)$ and $f(x \pm e)$ are defined for each arithmetic operation individually. The definitions closely match the ones obtained with $x(1+e)+d$ abstraction, and only differ in how the error from previous computations is propagated for some operations, for instance, for the square root:

$$\varepsilon_{\sqrt{x}} = \sqrt{e_x} + \frac{1}{2} ulp(\sqrt{x + e_x}), \text{ for } x \geq 0 \wedge \hat{x} \geq 0 \tag{2.17}$$

where $\epsilon_f(e) = \sqrt{e_x}$ and $f(x \pm e) = \sqrt{x + e_x}$, and e_x is initial (propagated from previous computations) error on x.

PRECiSA's definition for newly committed error (per operation) is identical to the standard abstraction in Equation 2.10: for normal numbers and the default rounding mode (to nearest), rounding error is $\frac{1}{2}ulp(x) = \frac{1}{2}\beta^{1-p} = \beta^{1-p-1} = \beta^{-p} = \epsilon_M$.

Similarly to other global optimization-based approaches, PRECiSA builds a symbolic expression for the error first, and then applies a branch-and-bound solver to quantify the worst-case error bounds. For the computed errors PRECiSA produces correctness certificates checked with an interactive theorem prover PVS [72].

Limitations

While these techniques compute tight error bound estimates for straight-line code, their support beyond basic blocks is limited. Conditional statements are handled using branch-by-branch evaluation [13,37,73] that often reports pessimistic results, or require additional information on the error distribution [38] and do not scale well. Loops larger than several hundreds of iterations that cannot be easily unrolled are beyond what state-of-the-art can do. Daisy currently does not handle loops at all, Rosa only supports a special form of loops: they must be non-nested, contain no conditionals, and the ranges of variables inside the loops must be bounded and fixed statically [13]. For many programs, however, these fixed finite-precision ranges are unlikely to be known in advance (in contrast to real-valued ranges of input variables) and must be precomputed using some complementary technique. Fluctuat can analyze loops of a more general form, however, the resulting error bounds are often trivial ($[-\infty, +\infty]$) or take too long to compute on large programs.

An additional limitation of the global optimization approach is that it is not directly applicable to fixed-point programs. Unlike floats whose dynamic range allows them to represent many values, fixed-points require the integer and fractional bits to be assigned individually for each (sub-)expression. To do that, one needs to know the range of values taken by a (sub-)expression. While it is technically possible to obtain this information in advance (for instance, with some other analysis), this incurs a significant overhead. Therefore, the global optimization-based approach is only applied to floating points.

2.2.3. Optimization of Finite-Precision Programs

The original goal of rounding error analysis is to obtain reachable states (in terms of values of variables), and verify that none of them are unsafe (potentially with the help of other tools). In addition to the verification aspect, analysis results can be further used to identify optimization potential and improve programs. Guided by a sound error analysis, numerical program optimizers target programs' performance or accuracy. The inverse relation between accuracy and performance makes any optimization challenging, the goal of the optimizers is to find a suitable trade-off.

Accuracy - Rewriting Optimization

Finite-precision arithmetic does not exactly match the real-valued one. Many real properties do not hold in the finite precision, for instance, a sum is non-associative `(a + b) + c` $\neq$ `a + (b + c)`. While this makes writing programs in finite precision unintuitive, it also provides an optimization opportunity: by *rewriting* the expression to be evaluated in a different order, one can influence the resulting rounding error on the expression.

Assuming that a numerical algorithm has been designed with real arithmetic in mind, the rewriting optimization applies real semantics-preserving identities to a finite-precision expression and chooses the one with the smallest total rounding error. The semantics of the algorithm do not change, but the accuracy may be increased.

The challenge in this optimization is the huge search space, for an expression with n arithmetic operations and k ways to pair the operands (i.e., put the brackets around them), there are $n! \cdot k$ expression variations with different orders of evaluation. Some of them may have the same end error, some different, enumerating all of them is not feasible. Therefore, an optimizer needs an efficient strategy for selecting candidates.

The sound optimizer Daisy uses genetic programming to facilitate the search of a more accurate order of evaluation [74]. The search is initialized with a predefined number of copies of the original expression, this is the initial population. At each iteration of the search the population is sorted based on their fitness (in this case, rounding error), the fittest candidate's order of evaluation is mutated (randomly, but according to the real-valued identity rules), and the fitness is evaluated again. This process repeats for 30 iterations and reports the expression with the smallest found rounding error. Note that this search is necessarily incomplete, but was shown to be effective [22].

Salsa [52] uses a combination of static analysis and Abstract Program Expression Graphs (APEG): it groups expressions equivalent under real semantics into one class and evaluates them using abstract semantics, at the end one candidate with the smallest rounding error is selected per class. Salsa's approach applies this transformation inter-procedurally, beyond individual arithmetic expressions.

To the best of our knowledge Daisy and Salsa are the only *sound* optimizers that implement rewriting. Other tools employ some forms of rewriting in order to repair large errors [19–21], but do not provide guarantees on the resulting error. Since these repair tools are not sound and have a different objective, we postpone the discussion until section 5.4.

Performance - Mixed Precision Assignment

Numerical computations in many domains have to be executed frequently and require the result quickly, for instance, when computing a control signal for a robotic arm, deciding whether a heart rate pattern should be classified as life-threatening, or whether a wearable device has detected that its user fell down [75]. When performance matters, but computed results must remain meaningful, *mixed-precision tuning* [22, 23] can be applied. The user can define how noisy the results can get to stay meaningful, and this portion of accuracy will be sacrificed for improving performance.

Mixed-precision tuning is based on the premise that generally the higher the (finite) precision used in a program, the slower it runs. Different arithmetic operations, however, contribute to the final error differently—some magnify the propagated error significantly, and some diminish it. Therefore, the individual sub-expressions may require different accuracy and may be implemented in different precisions (hence, mixed precision). The idea behind mixed precision tuning is to use low precision when possible and high precision when necessary. That said, if the lowest available uniform precision assignment already satisfies the specified target error, no higher precision will be assigned.

As for rewriting, the search space for mixed-precision tuning is huge and requires efficient search procedures. Sound state-of-the-art tools search through possible mixed-precision assignments with delta debugging [22], a combination of forwards and backwards static analysis [25], or encode precision assignment as optimization problem [23] and solve it using the industrial strength optimizer Gurobi [76]. Other tools use dynamic analysis to guide precision assignment [10, 51, 77–79] and therefore do not provide guarantees on resulting rounding errors.

2.3. Daisy Framework

Our analysis of loops over vectors and matrices (chapter 3) and two optimizations (chapters 5, 6) are implemented as extensions of the open-source tool Daisy [14]. Here we present the details about Daisy that are relevant for our extensions.

Input Format Daisy's input is a *real-valued* specification of numerical algorithms written in a subset of the functional language Scala [80]. Figure 2.1a shows an example input file. It contains necessary library imports and a top-level object (`RigidBody`) with potentially multiple algorithms to be analyzed and optimized. Each algorithm is defined in a separate function (`rigidBody2`, `anotherAlgorithm`) and is handled by Daisy separately.

An example algorithm `rigidBody2` describes a non-linear controller for the angular velocity of a rigid body [81]. The function `rigidBody2` takes three real-valued inputs `x1`, `x2` and `x3` with input ranges specified by the `require` clause in lines 6-7. It then computes one output (line 8), for which the maximum tolerated error 0.01 is specified by the `ensuring` clause (line 9). Note that this specification describes the ideal real-valued algorithm and cannot be executed as-is.

Output The main functionality of Daisy is twofold: 1) analyze numerical programs for worst-case rounding errors and 2) optimize them. When executed only in analysis mode, Daisy reports absolute rounding error bounds, real-valued range of the resulting value, and relative rounding error bounds for cases when the real range does not include zero.

When used to optimize programs, Daisy reports analysis results for the optimized program and generates executable code in Scala or C (can be configured by the user). The generated code includes the precision assignment (uniform or mixed), casts between precisions (whenever needed for mixed-precision assignment), and the exact order of evaluation (for example, the one obtained with the rewriting optimization). Figure 2.1c shows the output Scala code generated for the example specification `rigidBody` with mixed-precision tuning on floating points.

For fixed-point programs, an additional post-processing step may be required. Whenever the operands of an arithmetic operation have different numbers of fractional bits, Daisy automatically generates bit shifts to align the values for correct computation [49]

(Figure 2.1b) or uses the library implementations, such as `ap_fixed` format for Xilinx Vivado [59].

Implementation Structure Daisy has a modular structure, it reads a set of flags and parameters from the command line and builds the execution pipeline from the corresponding components (called `Phases` in Daisy). For instance, to optimize a program `example.scala` with Daisy's mixed-precision tuning and rewriting one has to run the following command in a terminal:

```
./daisy example.scala --mixed-tuning --rewrite
```

Such a structure makes extending the tool straight-forward, one needs to implement a `Phase` that processes the AST (with analysis or transformation) and add it to the pipeline.

When processing inputs, Daisy internally uses Scala compiler's front-end for parsing and type-checking. The domain-specific language for real-valued operations on the datatype `Real` is defined in the `Real.scala` file and imported as a library to each specification file. Another mandatory library `daisy.lang` implements extraction of the Daisy-specific abstract syntax tree (AST) and operations on it. These libraries can be reused and extended. When implementing a new domain-specific language that operates on vectors and matrices, we created new types `Vector` and `Matrix` on top of the `Real` type, and defined the operations on the new data types in `daisy.lang`.

More details on Daisy's features can be found in the corresponding tool paper [14] and in the online documentation [82].

```
import daisy.lang._
import Real._

object RigidBody {
  def rigidBody2(x1: Real, x2: Real, x3: Real): Real = {
    require(-15.0 <= x1 && x1 <= 15 && -15.0 <= x2 && x2 <= 15.0 &&
      -15.0 <= x3 && x3 <= 15)
    2*(x1*x2*x3) + (3*x3*x3) - x2*(x1*x2*x3) + (3*x3*x3) - x2
  } ensuring(res => res +/- 1e-2)

  def anotherAlgorithm(...): Real = {
    ...
  }
}
```

(a) Input specification

```
#include <math.h>

long rigidBody2(long x1, long x2, long x3) {
 long _tmp13 = ((x1 * x2) >> 31);
 long _tmp14 = ((_tmp13 * x3) >> 31);
 long _tmp16 = ((1073741824 * _tmp14) >> 30);
 long _tmp15 = ((1610612736 * x3) >> 31);
 long _tmp17 = ((_tmp15 * x3) >> 31);
 long _tmp20 = (((_tmp16 << 3) + _tmp17) >> 3);
 long _tmp18 = ((x1 * x2) >> 31);
 long _tmp19 = ((_tmp18 * x3) >> 31);
 long _tmp21 = ((x2 * _tmp19) >> 31);
 long _tmp23 = ((_tmp20 - (_tmp21 << 3)) >> 3);
 long _tmp22 = ((1610612736 * x3) >> 31);
 long _tmp24 = ((_tmp22 * x3) >> 31);
 long _tmp25 = (((_tmp23 << 6) + _tmp24) >> 6);
 return (((_tmp25 << 12) - x2) >> 12);
} // [-58740.0, 58740.0] +/- 0.0003096703
```

(b) Output C code for uniform 32-bit fixed-point precision with bit shifts

```
import scala.annotation.strictfp

@strictfp
object RigidBody {
 def rigidBody2_32_05(x1: Double, x2: Double,
                      x3: Float): Double = {
   val _const0: Float = 2f
   val _const1: Double = 3
   val _const2: Double = 3
   val _tmp13: Double = (x1 * x2)
   val _tmp14: Double = (_tmp13 * x3)
   val _tmp16: Double = (_const0 * _tmp14)
   val _tmp15: Double = (_const1 * x3)
   val _tmp17: Double = (_tmp15 * x3)
   val _tmp20: Double = (_tmp16 + _tmp17)
   val _tmp18: Double = (x1 * x2)
   val _tmp19: Double = (_tmp18 * x3)
   val _tmp21: Double = (x2 * _tmp19)
   val _tmp23: Double = (_tmp20 - _tmp21)
   val _tmp22: Double = (_const2 * x3)
   val _tmp24: Double = (_tmp22 * x3)
   val _tmp25: Double = (_tmp23 + _tmp24)
   (_tmp25 - x2)
 }} // [-58740.0, 58740.0] +/- 0.0019097329
```

(c) Output Scala code with mixed floating-point precision

Figure 2.1.: Daisy's input/output for the `rigidBody` algorithm.

Part I.

Analysis of Programs With Loops

3. Large Bounded Loops

Estimation of rounding error bounds on programs with bounded loops is currently limited by the size of the loop. Today's tools successfully handle loops with reasonably small bounds by unrolling them, but the same method does not work when a bounded loop has more than a few hundred iterations. Such loops, however, are common in large neural networks [7] and control systems that sample sensor values frequently and store them before computing the actual control signal.

Recent efforts to address scalability of analyzers involve various abstractions. The tool Satire [57] builds an incremental abstraction of numerical expressions in a form of a directed acyclic graph, and uses symbolic Taylor terms [71] to compute tight rounding error bounds. Similarly to FPTaylor [71], that introduced symbolic Taylor approximation terms, Satire uses these terms to formulate error bound search as an optimization task. While Satire handles larger programs than other state-of-the-art tools, it requires a user to unroll all loops manually, which is a tedious an error-prone process.

Another state-of-the-art static analyzer Fluctuat [39] does not impose such strict constraints on the input program, it handles general-form imperative programs and takes care of loops automatically. Fluctuat provides several possibilities to handle loops: compute a merge-over-all-paths (MOP) solution, which is essentially unrolling the loop, or apply widening—in principle, an abstraction—, and a combination of both (widen only after a certain fixed number of unrolled iterations). Fluctuat's base analysis for straight-line code is extremely fast, which allows it to handle larger programs with unrolling as well, but also up to some limit. Widening allows the analysis to converge to a fixpoint solution quickly, however, it is too imprecise for estimating rounding errors and often returns a trivial bound of $[-\infty, \infty]$.

Therefore, there is a need for automated tools that handle large numerical loops and do not require users to specify prohibitively long inputs. While handling loops of a general form is still challenging, there are sub-classes of loops, where scalability of static analysis can be improved with abstractions.

Many numerical programs operate on values stored in data structures such as N-dimensional arrays. Numerical computations are then performed in some form of a loop over such array. Loops of this form occur frequently in various application domains, for instance, in statistical computations in data analyses, image and other signal processing, Fourier and stencil transformations in embedded systems, computations in neural networks, etc. State-of-the-art tools handle these loops by assigning an individual variable to each array element, which is exactly the same as unrolling. An alternative approach of abstracting the whole array as one unit that is applied in various static analyses [83] does not work well for rounding error analysis. While this abstraction speeds up the analysis, the inherent over-approximation is in general too coarse for analysis where values should be computed as tightly as possible, and may lead to unusable results. This chapter presents the first rounding error analysis with *explicit* support for operations and loops over array-like data structures (i.e. vectors or lists and matrices). To facilitate this analysis we design a *functional* domain-specific input language (DSL) with operations over lists and matrices that allows to express many commonly used patterns in numerical computing and that serves as the input to our tool.

The benefit of a functional input language is two-fold. First, it allows users to succinctly express their computations and reduces the possibility of common (off-by-one) indexing errors. More importantly, however, a functional language carries *semantic information* that can be *leveraged* by the analysis, removing the need to unroll many operations. For example, loops applying a function to each value in a list (functional `map(`$\lambda x.f(x)$`)`) do not propagate errors between iterations, and a rounding error analysis only has to analyze the loop body once. An unrolling of the loop would lose that high-level information and effectively re-compute the analysis for each loop iteration. For operations that do require unrolling, we show how to use the semantic information to avoid recomputing analysis information that can be effectively over-approximated, further reducing the burden on the analysis. Our abstraction is designed for rounding errors and accounts for different variables' ranges and thus provides a viable tradeoff between analysis accuracy and performance.

We design our input DSL based on a new set of numerical benchmarks that we collected from a variety of domains. We implement our rounding error analysis for this DSL in a tool called DS2L and show that compared to a baseline analysis that unrolls all operations, it can substantially reduce analysis time with little impact on analysis accuracy.

Our focus in this work is on *scalability* and we thus compare DS2L against the two most scalable (available) rounding error analysis tools Fluctuat [39] and Satire [57]. For completeness, we create benchmarks with different data structure sizes. Our evaluation shows that for smaller ones unrolling such as done by Fluctuat and Satire is preferable, but for larger ones (the focus of this chapter), DS2L scales significantly better. While for benchmarks with smaller data structure sizes DS2L has comparable running time than Fluctuat [39], it can analyze 46% more *large* benchmarks (has fewer timeouts, overflows and infinite error bounds), and is median 24x faster. Compared to Satire [57], DS2L can handle 59% more *large* benchmarks and is median 260x faster.

While we evaluate DS2L only on floating-point code to permit a comparison with existing tools, our analysis is general and extends to fixed-point arithmetic as well.

Contributions. In summary, this chapter makes the following contributions:

- a new finite-precision benchmark set to be released as open-source;
- a fully automated, sound rounding error analysis for programs written in a functional-style DSL (section 3.3);

- the open-source implementation of this analysis is available at `https://github.com/malyzajko/daisy/tree/ds2l` (section 3.4);
- an evaluation against state-of-the-art analysis tools in terms of accuracy and time (section 3.5).

3.1. Baseline Rounding Error Analysis

Our approach builds on top of the existing rounding error analysis tools that work for straight-line code with arithmetic operations on *scalar* values. Our data-structure-oriented analysis explained in section 3.3 reuses the straight-line code analysis as a baseline for evaluating scalar expressions, and we use it also for comparison in the evaluation (section 3.5).

We choose the baseline analysis to be of the dataflow type as implemented in Fluctuat [39], Rosa [13] and Daisy [14]. The alternative analysis phrases the computation of the rounding error as a global non-linear real-valued optimization problem [57,71,73]. We specifically choose a dataflow approach as our base analysis for several reasons. First, it is unclear how to generate global error constraints in the presence of data structures. Additionally, we identified optimization opportunities when the range information is available separately from the errors. Finally, even though in this paper we focus on floating-point arithmetic for simplicity, dataflow analysis is immediately applicable to fixed-point arithmetic as well, making our analysis more widely applicable. Global optimization-based rounding error analysis, as it appears in state-of-the-art tools FPTaylor [71] and Satire [57], analyzes floating-point programs only.

3.2. DSL for List-like Data Structures

Before designing our functional domain-specific language for numerical computations (subsection 3.2.2), we collected a new set of benchmarks that informed the design of our DSL, and specifically the set of supported operations (subsection 3.2.3).

3.2.1. Benchmark Set

Rounding error analysis on programs that contain operations on data structures such as arrays and loops over them is an open challenge, and correspondingly there is no standard benchmark set yet. The existing FPBench benchmarks [84] cover only straight-line code and a few while-loops but no data structures. We therefore create a new benchmark set that covers different domains where numerical computations are frequent:

- statistical computations: *avg, stdDeviation, variance*
- linear and non-linear digital filters: *roux1, goubault, harmonic* and *nonlin{1-3}* [46]

- differential equations: *lorenz, pendulum* [13,57]
- signal processing: *alphaBlending* (image mask), *fftvector, fftmatrix* (two versions of forward Fourier transform)
- stencil computations: *convolve2d_size3, sobel3, heat1d* [13,57]
- neural networks: *lyapunov, controllerTora* [7]

Some of the benchmarks from FPBench contain loop bodies of control loops, which we rephrase as loops over arrays of sensor data. Other benchmarks have been collected from scientific publications [7, 13, 46, 57] as well as open-source implementations in different programming languages.

The benchmarks are available in the appendix, section A.1.

3.2.2. A Functional DSL

Many verification techniques face the dilemma of either adapting the verification techniques to work on legacy code and (possibly) giving up some precision, or requiring to rewrite the code with verification in mind and being able to reason about a program in more detail. In this work, we choose the second option, and note that our domain-specific language uses Scala syntax and is similar to other existing functional languages and we thus expect it to be largely familiar to developers.

The goal of our DSL is to allow a convenient way to 1) write programs that perform operations on array-like data structures and 2) to analyze them. Our main insight is that a functional style of programming covers both aspects: it allows for a more succinct representation of programs and it retains high-level semantic information of the operations that can be leveraged by the analysis.

heat1d Example We illustrate the succinctness of our DSL on one of the benchmarks that we collected from related work [57]. Figure 3.1 shows the function *heat1d* in the input formats of two different tools. The *heat1d* function takes as input a temperature distribution and computes the temperature at a coordinate `x0` after 32 units of time. The computation requires temperature values for neighboring coordinates which must be repeatedly recomputed, which is essentially a stencil.

The original straight-line version of the *heat1d* benchmark comes from Satire analyzer [57] and includes *1094 lines of code*, 67 of which specify input ranges of (individual) variables, the rest are unrolled loops. Its representative parts are shown in Figure 3.1a. Unrolled computations are not only lengthy, but also error-prone and unnatural for a user to write. A more natural choice when implementing the same algorithm in an imperative style is to use two nested loops. Figure 3.1b shows the same algorithm written in C formatted for the tool Fluctuat [39]. A loop representation is more succinct—14 lines of code with computations, however it requires loop bounds to be set manually and may lead to index-out-of-bounds errors.

```
INPUTS {
     xm1_0   fl64 : (1,2) ;
     xm2_0   fl64 : (1,2) ;
     ... // repeat until xm32_0
     xm32_0   fl64 : (1,2) ;
     x0_0     fl64 : (1,2) ;
     xp1_0   fl64 : (1,2) ;
     xp2_0   fl64 : (1,2) ;
     ... // repeat until xp32_0
     xp32_0   fl64 : (1,2) ;
     }
OUTPUTS { x0_32; }

EXPRS {
  xm31_1 rnd64 = (0.25*xm32_0 + 0.5*xm31_0 + 0.25*xm30_0);
  xm30_1 rnd64 = (0.25*xm31_0 + 0.5*xm30_0 + 0.25*xm29_0);
  ... // repeat until xm1_1
  x0_1 rnd64 = (0.25*xm1_0 + 0.5*x0_0 + 0.25*xp1_0);
  xp1_1 rnd64 = (0.25*x0_0 + 0.5*xp1_0 + 0.25*xp2_0);
  ... // repeat until xp31_1
  xm31_2 rnd64 = (0.25*xm32_1 + 0.5*xm31_1 + 0.25*xm30_1);
  xm30_2 rnd64 = (0.25*xm31_1 + 0.5*xm30_1 + 0.25*xm29_1);
  ... // repeat until xm1_2
  x0_2 rnd64 = (0.25*xm1_1 + 0.5*x0_1 + 0.25*xp1_1);
  xp1_2 rnd64 = (0.25*x0_1 + 0.5*xp1_1 + 0.25*xp2_1);
  ... // repeat until xp31_2
  // repeat until x0_31 is reached
  x0_31 rnd64 = (0.25*xm1_30 + 0.5*x0_30 + 0.25*xp1_30);
  xp1_31 rnd64 = (0.25*x0_30 + 0.5*xp1_30 + 0.25*xp2_30);
  x0_32 rnd64 = (0.25*xm1_31 + 0.5*x0_31 + 0.25*xp1_31);
}
```

(a) Unrolled loop (Satire's input)

```
#include <fluctuat_math.h>
#define N 33
// computations
double heat1d(double (*xm)[N],
              double (*xp)[N], double* x0) {
for(int j=1;j<N; j++) {
  for(int i=2; i<(N-j); i++) {
    xm[j][i] = 0.25*xm[j-1][i+1] +
               0.5*xm[j-1][i] + 0.25*xm[j-1][i-1];
    xp[j][i] = 0.25*xp[j-1][i-1] +
               0.5*xp[j-1][i]+0.25*xp[j-1][i+1];
  }
  xm[j][0]=0.25*xm[j-1][1]+0.5*xm[j-1][0]+0.25*x0[j-1];
  xp[j][0]=0.25*xp[j-1][1]+0.5*xp[j-1][0]+0.25*x0[j-1];
  x0[j]=0.25*xm[0][j-1]+0.5*x0[j-1]+0.25*xp[0][j-1];
}
return x0[N-1]; }
int main() {
  int i,j;
  double x0[N];
  double xm[N][N];
  double xp[N][N];
  // specify input ranges
  for(i=0; i<N; i++){
    x0[i] = DBETWEEN(1.0, 2.0);
    for(j=0; j<N; j++){
      xm[i][j] = DBETWEEN(1.0, 2.0);
      xp[i][j] = DBETWEEN(1.0, 2.0);
  }}
  heat1d(xm, xp, x0);
  return 0; }
```

(b) Imperative loop (Fluctuat's input)

```
def heat1d(ax: Vector): Real = {
  require(1.0 <= ax && ax <= 2.0 && ax.size(33))
  if (ax.length() <= 1) {
    ax.head
  } else {
    val coef = Vector(List(0.25, 0.5, 0.25))
    val updCoefs: Vector = ax.slideReduce(3,1)(v => (coef*v).sum())
    heat1d(updCoefs)
  }
}
```

(c) Functional style (our DSL)

Figure 3.1.: heat1d benchmark in input formats for different tools

We show the same function *heat1d* written in our functional DSL in Figure 3.1c. It uses a sliding window over a list (`slideReduce` operation, explained in more detail in subsection 3.2.3) and passes the new values into a recursive call. In contrast to alternative implementations, a functional style program is much shorter—6 lines of code—and eliminates index-out-of-bounds errors as it does not require users to explicitly write elements' indices.

DSL Design Our DSL is designed for writing numerical algorithms on array-like data structures and was inspired by the popular libraries Lift [85] and TensorFlow [86]. It includes commonly occurring operations on vectors and matrices from the collected benchmarks. When naming DSL functions, we have re-used the names used by Lift and TensorFlow whenever possible and attempted to make other functions' names self-explanatory. We do not expect our current DSL to exhaustively cover all possible numerical programs; rather it serves as a starting point already covering a variety of operations that can and should be extended in the future.

Data Types Following previous work in rounding error analysis, all values and operations in our DSL are real-valued (as opposed to finite precision), i.e. they have a `Real` type. Real-valued algorithms are more intuitive for a user to write, and easier to analyze as they provide a clear reference semantics. Our DSL provides two data types: a `Vector` is an indexed sequence of `Real` scalar values, and a `Matrix` corresponds to a sequence of vectors of the same length. In the following, we refer to lists (`Vectors`) as vectors, and vectors and matrices as data structures (DSs), for simplicity. Our DSL is purely functional, and as such all data structures are immutable.

Input Ranges Any rounding error analysis requires information on ranges of input variables. Both scalar and DS input ranges can be specified using the `require` clause. The specification should ideally be as precise as possible and provide tight ranges that can be different for some DS elements. We therefore allow two ways to specify input ranges for DSs. If all elements have the same input range, it is enough to specify the range once for the whole DS (`1.0 <= ax && ax <= 2.0`). Additionally, it is possible to specify individual input ranges for subsets of DS elements. For vector elements these ranges are specified as a tuple $((loInd, hiInd), range)$, where *loInd* and *hiInd* are the smallest and the largest index of consecutive elements with the input range *range*. For example, to specify that the first and the second element of `ax` in *heat1d* have the input range $[0.0, 0.5]$, we would write `ax.range(0, 1)(0.0, 0.5)`. We also allow individual range specifications on matrices, however, specifying a lower and upper bound of an index range is ambiguous for a matrix. Therefore, we choose a more natural way for specifying special input ranges on matrices: a user has to list the indices of elements for that range. For example, to convey that the first elements in the first and second row of a matrix `m` should have the range $[-0.5, 0.5]$, we write `m.specM(Set(Set((0,0),(1,0)),(-0.5,0.5)))`.[1]

DS Size To analyze operations that traverse a DS, the analysis also needs to know the number of elements in the DS. Our DSL allows to specify the expected *maximum size* of an input data structure—length of a vector, number of rows and columns for a matrix. Having the upper bound on the number of elements in the DS allows us to compute

[1]Admittedly, the `Set()` notation is not the most user-friendly way of input for small specifications. We use it for simplicity of implementation; the notation can be improved with extensions to our parser.

```
object Vector {
  def zeroVector(i:Int): Vector
  def zip(v1: Vector, v2: Vector): Matrix
}
case class Vector(data: List[Real]) {
  // uncertainty on the vector
  def +/-(x: Real): Boolean
  // specify one range for the whole vector
  def <=(x: Real): Boolean // also >=
  // specify range for a subset of elements
  def specV(ranges: Set[((Int, Int), (Real, Real))]):
    Boolean
  def size(i: Int): Boolean
  // element-wise operations
  def +(v: Vector): Vector // also -,*,/
  // element-wise elementary functions
  def log(): Vector // sin(), cos(), tan(), ctan(), etc.
  // cross-product
  def x(v: Vector): Vector
  // operations with constants
  def *(c: Real): Vector // also +, /
  // non-arithmetic operations
  def length(): Int
  def at(i: Int): Real
  def slice(i: Int, j: Int): Vector
  def everyNth(i: Int, from: Int): Vector
  // standard functions
  def map(fnc: (Real) => Real): Vector
  def fold(init: Real)(fnc: (Real,Real) => Real): Real
  def filter(fnc: (Real) => Boolean): Vector
  // sliding window
  def slideReduce(size:Int, step: Int)(
    fnc:(Vector) => Real): Vector
  def enumSlideFlatMap(n: Int)(
    fnc: (Int, Vector) => Vector): Vector
  // add zeros padding around the vector
  def pad(i: Int): Vector
  def max(): Real // also min()
  def sum(): Real // same as fold(0.0)(λa,x.a+x)
  // concatenate and add elements
  def ++(v: Vector): Vector
  // also append :+(_), prepend +:(_)
}
object Matrix {
  def zeroMatrix(i:Int, j:Int): Matrix
}
case class Matrix(data: List[List[Real]]) {
  // < same as in Vector >
  // element-wise operations and elem. functions
  // operations with constants
  // non-arithmetic operations
  // basic functional ops
  // < different from Vector >
  // input spec for range and size
  def specM(ranges: Set[(Set[(Int, Int)],
                    (Real, Real))]): Boolean
  def size(i: Int, j: Int): Boolean
  // +non-arithmetic operations
  def row(i: Int): Vector
  def slice(fromI:Int, fromJ:Int)
                   (toI:Int, toJ:Int): Matrix
  def at(i:Int, j: Int): Real
  def numRows(): Int
  def numCols(): Int
  // flip elements upside down
  def flipud(): Matrix // also fliplr() left to right
  def enumRowsMap(fnc: (Int, Vector) => Vector): Matrix
  // operations on individual elements
  def mapElements(fnc: (Real) => Real): Matrix
  def foldElements(init: Real)(
        fnc: (Real,Real) => Real): Real   }
```

Figure 3.2.: DSL for numerical programs on data structures

sound results: reported ranges and rounding errors subsume the ranges and errors of programs with input DSs smaller than the specified size.

3.2.3. DSL Functions

Our DSL uses Scala syntax, precisely it is an extension of the real-valued specification language of Daisy. We show a representative subset of the DSL functions in Figure 3.2; semantically we can roughly split its functions into four groups:

1. *element-wise functions*, such as arithmetic operations and transcendental functions applied to individual elements of a DS;
2. *standard higher-order functions*, such as `map`, `fold` and `filter`;
3. *domain-specific* functions, e.g., stencil-like filters, matrix multiplication;

4. *non-numerical* operations, e.g., appending or flipping elements in DS.

Additionally, our DSL supports recursive calls with specific conditional statements. To avoid rounding errors in conditional expressions, we currently limit them to (integer) DS size comparisons, such as `v.length <= c` or `m.numRows <= c`. Next, we explain the concrete semantics of the DSL functions using pseudocode that makes indices explicit (while they are typically implicit in our DSL). We choose to present the semantics with pseudocode (and not sets of rules), because it is more concise and because it expresses how the operators are ultimately evaluated, which is important for the rounding error analysis.

Note that the semantics of most of our DSL operators are standard. Additionally, our analysis does *not* depend on exactly this syntax and semantics of the DSL. We therefore expect our analysis to be *applicable to other (intermediate) representations or languages* with similar semantics. Such representation must (only) be purely functional (immutable variables and DS, no side-effects) and provide a syntactic distinction between different iterators, precisely, the functionality of an iterator must be unambiguous without an additional analysis of the iterator's body.

Element-wise Functions They cover arithmetic operations applied to a single DS or a pair of DS, for instance $v1 + v2$, where $v1, v2$ are vectors. Semantically these operations are the same as arithmetic operations on scalar numbers. The only difference is that for binary operations on two DS, the operands must have the same dimensions. Element-wise operations are defined for both vectors and matrices: the operation is applied to the elements in the operand DSs with the same indices. We also define element-wise operations with constants.

For all unary (`uop`) and binary (`bop`) arithmetic operations the semantics is:

```
a bop b = [a[i] bop b[i] | ∀i∈Indices(a), #Indices(a) == #Indices(b)]
uop(a) = [uop(a[i]) | ∀i∈Indices(a)]
```

In our example function *heat1d* in Figure 3.1c (line 8) the expression `coef*v` is an element-wise multiplication of vectors `coef` and `v`. It will multiply each *i*-th element of `coef` with the *i*-th element of `v` and put the result in the *i*-th element of an output vector.

Standard Higher-Order Functions Classic higher-order functions `map`, `fold`, `filter` preserve their semantics. `map` and `fold` are defined on vector elements, and for a matrix on both rows and elements. We add a function `ds.sum()` as syntactic sugar for `fold` with an addition operator to compute a sum of DS elements. We also extend the map on matrix rows to support indexed iterations with `enumRowsMap(λi,x.f(i,x))`. The function maps over rows of the matrix and applies f to both row's index and elements:

```
m.enumRowsMap(f) = [f(i, m[i,j]) | ∀i∈Rows(m), (i,j)∈Indices(m)]
```

`filter` is defined to apply the conditional to vector elements, and to matrix rows. We do not allow a `filter` on individual matrix elements, as it may result in modified and uneven matrix dimensions.

Domain-Specific Functions Our DSL defines operations required for implementing neural networks (i.e. matrix multiplication), stencils and image processing filters. We describe the most interesting operations below.

Stencil operations usually require a more complex transformation than `map` or `fold` can provide. The transformations involve an outlook of several elements before and after the current element of a DS, as opposed to accessing a single element in one iteration of `map` and `fold`. Such an outlook is commonly called a sliding window. Our DSL defines it on vectors and matrices with `ds.slideReduce(size, step)(`λ`x.`f`(x))`, where a window of size `size` shifts by `step` indices at every iteration. For vectors a window is a subset of consecutive elements of length `size`, for matrices a window is a matrix with dimensions `size`$\times$`size`. A user-supplied function $f(x)$ is then applied to the created window, it returns a scalar value that is saved at the corresponding index of the newly created DS. Intuitively, it is similar to applying a `fold` to a sliding window.

Our example benchmark *heat1d* in Figure 3.1c creates a sliding window of 3 vector elements and shifts the window by 1 index at every iteration; the resulting vector `updCoefs` contains results of the `sum()` operation. The pseudocode below explains `ax.slideReduce(3,1)(f)` using explicit indices of the vector `ax`:

```
k=0
∀ i∈ {1..size(ax)-2}:
   v = [ ax[i-1], ax[i], ax[i+1] ]
   // f(v) = (coef*v).sum()
   updCoefs[k] = coef[0]*v[0] + coef[1]*v[1] + coef[2]*v[2]
   k++
```

Note that the pseudocode contains two different indices: i is the index of elements in the original DS `ax`, and k is the index of a sliding window over `ax` and the output vector `updCoefs`.

Our DSL also allows a combination of a sliding window and a `map`, which is useful for implementing signal filters such as the fast Fourier transform. The function `enumSlideFlatMap(n)(`λ`i,x.`f`(i,x))`, defined on vectors, creates a sliding window of size n that shifts by n indices every iteration. The resulting windows are enumerated and a function $f(i,x)$ transforms every element in the window and saves the results into a new vector. In the FFT implementation in Figure 3.3, the sliding window includes 2 elements of the vector `evens` and computes vectors `resleft` and `resright` of the same size as `evens`. The window index `k` is used for accessing elements of the vector `odds` and for computing the filtered values (lines 16 and 22). The pseudocode below explains with explicit indices how `evens.enumSlideFlatMap(2)(f(k,xv))` iterates over the vector `evens`:

```
k=0
∀ i∈ {0,2,4,...,size(evens)-2}:
   xv = [ evens[i], evens[i+1] ]
```

```
    def fftvector(vr: Vector, vi: Vector): Vector = {
 2   // v: (real part of signal / Fourier coeff.,
     // imaginary part of signal / Fourier coeff.)
 4   require(vr >= 68.9 && vr <= 160.43 && vr.size(128) &&
             vi >= -133.21 && vi <= 723.11 && vi.size(128))
 6    if (vr.length() == 1)
        Vector(List(vr.head, vi.head))
 8    else {
        val scalar: Real = 1; val Pi: Real = 3.1415926
10      val n:Int = vr.length(); val direction:Vector = Vector(List(0.0, -2.0))
        val evens: Vector = fftvector(vr.everyNth(2, 0), vi.everyNth(2, 0))
12      val odds:  Vector = fftvector(vr.everyNth(2, 1), vi.everyNth(2, 1))
        val resleft: Vector = evens.enumSlideFlatMap(2)((k, xv) => {
14        val base: Vector = xv / scalar
          val oddV: Vector = odds.slice(2 * k, 2 * k + 1)
16        val expV: Vector = (direction.*(Pi * k / n)).exp()
          val offset: Vector = (oddV x expV) / scalar
18        base + offset })
        val resright: Vector = evens.enumSlideFlatMap(2)((k, xv) => {
20        val base: Vector = xv / scalar
          val oddV: Vector = odds.slice(2 * k, 2 * k + 1)
22        val expV: Vector = (direction.*(Pi * k / n)).exp()
          val offset: Vector = (oddV x expV) / scalar
24        base - offset })
        resleft ++ resright  })
```

Figure 3.3.: Fast Fourier transform filter implemented in our DSL

```
tmp = f(k, xv) // where tmp is a vector
res[i] = tmp[0]; res[i+1] = tmp[1]
k++
```

Non-Numerical Such operations include obtaining a subset of elements (`v.slice(i,j)`), reordering (`m.flipud()`, `m.fliplr()`), appending and prepending elements and rows (`v.+:(elt)`, `m :+ v`). Additionally, our DSL allows to add a zero-padding around a vector or a matrix, and obtain smallest and largest elements of a DS. A special variant of a subset operation `ds.everyNth(n, fromInd)` creates a new DS by taking every n-th element of a vector (or row of a matrix) starting from the index $fromInd$. Our FFT benchmark in Figure 3.3 uses the `everyNth` function to obtain subsets of signal values at even and odd indices (lines 11 and 12).

3.3. Data-Structure Guided Analysis

While a baseline range and error analysis for straight-line code can handle unrolled iterators, it does not make use of implicit additional information that is present in a high-level specification. In an unrolled program each iteration makes up independent expressions to be evaluated, regardless of whether values in consecutive iterations depend on one another. This may result in redundant computations; for instance, a `map` performs the same computation over all elements in a vector and when all those elements have the same specified input range, we only need to analyze the rounding error of the computation once. The same holds for matrix multiplication: each element of the resulting matrix is computed with the same arithmetic expression, but it would appear as a new independent computation if unrolled. When sets of involved elements have the same ranges, it is sufficient to analyze the rounding error of the resulting matrix element once.

We observe that while concrete DS inputs will in general not be the same, a specification of a function to be analyzed will typically provide ranges that in practice often tend to be identical for many inputs. We leverage this in our analysis and compute the ranges and error bounds as rarely as possible. Even though it is not possible to directly apply this approach to iterators where iteration values have dependencies, like `fold`, the analysis can be optimized based on groups of elements with the same specification by introducing suitable over-approximations (see subsection 3.3.4).

We first introduce our DS-based concrete and abstract domains before explaining how expressions are analyzed and their analysis is optimized.

3.3.1. DS-based Concrete Domain

The goal of our analysis is to collect information about ranges and rounding error bounds for groups of elements. To do so, our concrete domain tracks a tuple (r, f) for each value in a program, where r is the ideal value if a program would be executed with a real numbers semantics, and f is the same value if the program is executed with the finite-precision semantics.

We denote all valid indices of data structures as $Inds^{(n)} = \mathbb{N}^n$, where $n \geq 0$ is the dimension of the DS: $n = 1$ for vectors and $n = 2$ for matrices. For scalar values the set of indices is empty, $n = 0$. Using the indices we define elements of a DS as $\mathbb{V}^{(n)} = Inds^{(n)} \mapsto (\mathbb{R}, \mathbb{F})$, where $(\mathbb{R}, \mathbb{F})$ denote sets of pairs of real and their corresponding finite-precision values (r, f). Given a set of elements' values $\mathbb{V}^{(n)}$ we define our concrete domain as $\mathbb{C}^{(n)} = 2^{\mathbb{V}^{(n)}}$, for each dimension of data structures n.

3.3.2. DS-based Abstract Domain

We then abstract each tuple (r, f) using a pair of intervals: $\alpha((r, f)) = (I_R \times I_R)$, where the first interval denotes a range of *real* values that contains r, and the second tightly bounds the *real-valued* difference between r and f. Here the difference between a real number r and a finite-precision number f represents the rounding error.

Lifted to the DS with dimension n we obtain abstract element's values: $\mathbb{D}^{(n)} = Inds^{(n)} \hookrightarrow (\mathbb{I}_R \times \mathbb{I}_R)$. Note that we are only interested in abstract values of elements with valid indices (as opposed to all possible indices), and use a partial mapping $\hookrightarrow$ to express it in our domain. For invalid indices the mapping is undefined. The abstract domain for our analysis combines all $\mathbb{D}^{(n)}$ with for scalar values, vectors and matrices: $\mathbb{D} = (\mathbb{D}^{(n)})_{n \geq 0}$. Join and meet operators use standard definitions of join and meet on intervals, and are lifted to all valid indices point-wise.

An abstract state $D^{(n)}$ soundly describes a concrete state $C^{(n)}$, that is: $C^{(n)} \subseteq \gamma(D^{(n)})$, where concretization function is defined as follows. Given a set of indices S and a set of mappings from these indices $D^{(n)} = \{i \mapsto (I_i, E_i) | i \in S\}$:

$$\gamma(D^{(n)}) = \{\{i \mapsto (r_j, f_j) | i \in S\} | \forall j. r_j \in I_i, |r_j - f_j| \in E_i\} \tag{3.1}$$

Each transformation of the abstract state is parametrized with an expression to be evaluated, a mapping of variables' values and computes a new abstract state:

$$[\![\cdot]\!]^\sharp = Expr^{(n)} \to (Vars \overset{\text{n}}{\mapsto} \mathbb{D}^*) \to \mathbb{D}^{(n)}, \tag{3.2}$$

where $\overset{\text{n}}{\mapsto}$ is a type-preserving mapping that assigns $D^{(0)}$ values to scalar variables, and $D^{(1)}$, $D^{(2)}$ values to vector and matrix literals respectively.

Theorem 1. ***Soundness.*** *Given an abstract state $D \in \mathbb{D}^{(n)}, \{i \mapsto (R, E)\} \in D$ there exists no concrete state $C \in \mathbb{C}^{(n)}$ such that $C \subseteq \gamma(D)$, $\{i \mapsto (r, f)\} \in C$ and $r \notin R \vee |r - f| \notin E$. Moreover, if $D \in \alpha(C)$, $[\![e]\!]C = C'$, and $[\![e]\!]^\sharp D = D'$, then $D' \in \alpha(C')$.*

Proof. (sketch) The theorem states that there is no unsound abstract state in our analysis, and given a sound starting state, our abstract transformations result in a sound end state. The first part follows directly from the definition of interval abstraction and concretization.

The transformations $[\![.]\!]^\sharp$ on data structures are defined for each individual element, which reduces them to transformations on basic blocks. The conditional expressions allowed in the language do not introduce instabilities or discontinuity errors [13,16], all iterators are ultimately reduced to straight-line code (with abstraction or unrolling) and thus do not require special treatment (explained in more detail later). Therefore, soundness of our analysis follows from the soundness of the underlying baseline analysis for straight-line code. $\square$

Our functional DSL defines all DS to be immutable, therefore each element of a DS is only assigned once. Our abstract domain does not require updates to individual element's ranges, and all recursive calls are unrolled. Since our analysis handles *only bounded loops* by design, we can unroll all operations, if needed, which is why we do not provide an additional widening operator. While widening in general allows the analysis to terminate quickly, for rounding error analysis the performance/accuracy trade-off is too costly. As our experiments with Fluctuat show (Section 3.5.1), for rounding error bounds, precision lost with widening cannot be recovered, hence an analyser that uses widening in the vast majority of cases reports infinite error bounds, which is sound but not especially meaningful.

3.3.3. DS Analysis

Both concrete and abstract domains partition DS elements in groups based on their real value and value range respectively. Our implementation describes a group of elements using a *set of indices*. The indices in one group need not be consecutive, the only condition is that they correspond to unique and valid indices of DS elements. Thus when analyzing an operator such as `map`, adding or multiplying by a constant, we only need to run the analysis once per group.

The initial grouping of elements is defined by user range specifications on the input DS. For intermediate variables in the computations, numerical indices for an abstraction of DS elements are inferred during the analysis. Note that the grouping does not change the semantics of functions and operators. As our DSL operates on real numbers, for commutative operations on DS elements their order does not matter. Whenever the analysis encounters an operation where the order of elements does matter, e.g. when computing an accumulator value in `fold`, we sort and split the groups to only contain consecutive elements' indices.

Whenever the expression under analysis contains only scalar values and operations, our analysis re-uses the baseline dataflow rounding error analysis, described in section 3.1. We next describe how our analysis handles different kinds of DS operations.

Example We illustrate our abstraction using the running example program in Figure 3.4. This contrived example is not part of our benchmark set, we use it here purely for demonstrating the relevant DSL details in a succinct way. Function `fun` takes two input vectors `x` and `y`, both of size 5. An abstraction for vector `x` keeps track of separate ranges for the first two elements and the remaining ones (with indices 2,3,4), i.e. $D_x^{(1)} = \{\{0,1\} \mapsto [0.5,1.5], \{2,3,4\} \mapsto [0,10]\}$. For the input vector `y` the abstraction also has two groups, but indices in the first group are not consecutive: $D_y^{(1)} = \{\{0,4\} \mapsto [-1,2], \{1,2,3\} \mapsto [0,1.5]\}$.

Map and Element-Wise Operations Our domains group elements that have the same real range by their indices, such that we can perform range evaluation once for each group. The most prominent example where such evaluation makes a difference for performance is the `map` function, such as on line 4 in Figure 3.4. The program multiplies all the elements of the list resulting from `x + y` by 2.0 and adds 1.5. The individual multiplications and additions are independent of each other, i.e. they do not propagate through iterations. For DS elements in one group we thus evaluate the range and error of `i*2.0 + 1.5` only once.

We use a similar approach for element-wise arithmetic operations between two vectors (or two matrices), such as `x + y` in Figure 3.4. In contrast to `map`, element-wise operations are binary and we need to take into account *pairs* of ranges. For each unique pair of ranges of operands we compute the range (and error) once. In our example program, `x + y` is performed on 4 pairs of ranges:

```
def fun(x: Vector, y: Vector): Real = {
 require(x>=0.0 && x<=10.0 && x.size(5) && x.range(0,1)(0.5,1.5) &&
        y>= -1.0 && y<=2.0 && y.size(5) && y.range(1,3)(0.0,1.5))
   val z = (x + y).map(i => i*2.0 + 1.5)
   val r = z.fold(1.0)((acc: Real, i: Real) => acc * sqrt(i))
   r / (x.length()) }
```

Figure 3.4.: Example program in our DSL

`x + y` indices	`x` range	`y` range
0	$[0.5, 1.5]$	$[-1.0, 2.0]$
1	$[0.5, 1.5]$	$[0.0, 1.5]$
2, 3	$[0.0, 10.0]$	$[0.0, 1.5]$
4	$[0.0, 10.0]$	$[-1.0, 2.0]$

Matrix Multiplication Evaluation of matrix multiplication is similar to the element-wise operations, where we compute pairs of ranges. Except, for matrix multiplication the elements, for which we need to know the ranges are located at the left-hand-side matrix row and the right-hand-side matrix column. We construct an expression for computing the resulting matrix elements internally. For each unique pair of ranges we only evaluate this expression once.

Filter Filter also takes advantage of the element grouping; our analysis evaluates the condition on each group of DS elements only once. However, `filter` is different from the rest of the functions in our DSL, because its abstract semantics do not exactly mirror the concrete. In the concrete semantics, `ds.filter(`$\lambda x.f(x)$`)` partitions the DS `ds` into two disjoint sets: elements that satisfy `f(x)`, and that satisfy its negation. In the abstract semantics these sets are not necessarily disjoint. Our evaluation `eval` returns an over-approximation of a set of elements from `ds`: the elements that *may* satisfy the condition `f(x)`. Currently we limit expressions in `f(x)` to simple comparisons $x \leq c$ and $x \geq c$, where `x` is the DS element and `c` is a scalar variable or a constant. More complex arithmetic operations are likely to introduce rounding error inside the condition itself, which may lead to a discontinuity error—elements that would have satisfied `f(x)` in a real-valued expression, do not satisfy it under floating-point semantics (or vice versa). We note that complementary techniques for bounding this discontinuity error [13,16] exist that may be integrated into our analysis.

Unrolled Operations Naturally, not all operations can benefit from a grouping of DS elements alone. The "once-per-range" evaluation cannot be applied on operations that propagate values through multiple iterations (`fold, slideReduce`) or use fresh values at each iteration (for example, loop counters in `enumSlideFlatMap, enumRowsMap`). For these functions, the abstraction-guided analysis falls back to the baseline version. It unrolls the iterators and performs range and error evaluation once for each iteration, we then

join the ranges (for values and, separately, for errors) to ensure that our results subsume all evaluated iterations. Our analysis handles recursive calls in the same way and unrolls each call as one iteration. Note that for our analysis to terminate, a recursive function must contain an exit condition that uses the (decreasing) length of a DS.
In our running example the analysis unrolls `z.fold` and evaluates ranges and errors of the unrolled expression:

```
1.0*sqrt(z.at(0))*sqrt(z.at(1))*sqrt(z.at(2))*sqrt(z.at(3))*sqrt(z.at(4)).
```

Non-Numerical Operations Operations that do not involve arithmetic computations do not introduce new errors, however, they do affect our abstraction. For example, a prepend operation `x.+:(8.0)` will add an element with index 0 and range $[8,8]$ to the abstraction and shift all indices of `x` by one. If we apply `x.+:(8.0)` to the `x` in the running example, the resulting abstraction will become $D_x^{(1)} = \{\{0\} \mapsto [8,8], \{1,2\} \mapsto [0.5,1.5], \{3,4,5\} \mapsto [0,10]\}$. Similarly, the `pad` operation adds elements with range $[0,0]$ around a vector or matrix and re-scales the original elements' indices. Another interesting case of the non-numerical operations is the `x.everyNth(n,k)` function that constructs a new DS by appending every n-th vector element (or every n-th matrix row) starting from the index k and assigning new indices to them. Evaluating `x.everyNth(2,0)` on the $D_x^{(1)}$ from our running example will result in $D_{nth}^{(1)} = \{\{0\} \mapsto [0.5,1.5], \{1,2\} \mapsto [0,10]\}$.

3.3.4. Optimized Evaluation of `fold`

The `fold` function cannot be evaluated only once per range group, since the accumulator's value changes at every iteration. For analysis, it would thus have to be unrolled. We observed, however, that in many applications the function passed to `fold` has a rather simple structure, such as summing up all elements of the DS. For such simple iterator bodies, the explicit unrolling can be replaced with an optimized evaluation that benefits from grouping of elements.

Our optimization over-approximates the accumulator, thereby effectively eliminating the change in input values from iteration to iteration. The analysis then computes one range per group of elements using a closed-form formula. In general, it is also possible to use approximation of an accumulator and a DS element for the whole loop, not only per group of elements with the same range. However, such a computation will introduce an even larger over-approximation in the result. To keep the bounds reasonably tight, we choose to apply over-approximations rarely.

We have implemented this optimization for the most common special cases of lambda functions `f()` that follow next.

Linear loop In a linear loop, i.e. $f(ac, el) = a \cdot el + b \cdot ac + c$, if f is executed on a group of elements with the same range $range(el)$, then we can compute the resulting range after n iterations with:

$$range_n = a \cdot range(el) \cdot \sum_{i=0}^{n-1}(b^i) + b^n \cdot init + c \cdot \sum_{i=0}^{n-1}(b^i), \tag{3.3}$$

where *init* is the initial value of the accumulator for the current group of elements. The initial accumulator value changes from group to group: it starts with the input parameter of `fold` and for each consecutive group it is replaced with the result of the previous computation. To account for all combinations of signs of linear coefficients a,b,c, we take their ranges to be symmetrical around zero. For generic linear loops, the order of computations matters, therefore we sort and split the groups in the abstraction $D^{(n)}$, such that each group only contains elements with consecutive indices, and the computation is applied to each group in the natural order: starting with the group containing index 0.

There is no simple closed-form equation to compute the rounding errors for linear loops. We therefore unroll the loop for error computations, but we use the over-approximated range of `acc`, pre-computed using Equation 3.3. Note that such an evaluation is faster than the full unrolling, since we pre-compute the ranges necessary for error computations.

Sum A sum of all elements in a vector or matrix is a special case of a linear loop, but in the absence of linear coefficients the range computations are much simpler. For a function $f(acc, el) = acc + el$, we compute one range per group of elements in $D^{(n)}$ abstraction using the formula: $n \cdot range(el) + init$, where n is the number of elements in the group, and *init* is the initial value of the accumulator for the current group. Note that here the order of groups does not matter, as our DSL specifies a program over real numbers and real-valued sum is associative.

The error computation is performed similar to linear loops: we over-approximate the value of `acc` and use the range to compute the error on the unrolled `fold`.

3.4. Implementation

We implement our analysis in a tool called DS2L as an extension of Daisy [14] written in the Scala programming language. For performance reasons, we implement all internal computations using intervals with arbitrary-precision bounds (with outwards rounding for soundness), using the MPFR library [11] with 128 bits of precision. We use the intervals for both range and error computation, and sacrifice some of the error accuracy compared to affine arithmetic that is used by most state-of-the-art analyzers.

We choose to implement the partitioning using sets of indices, among other alternative representations: linear inequalities [87,88], difference-bound matrices [87], and sets of other simple symbolic expressions [89]. We choose a set representation because it does not depend on patterns to group the elements. We have empirically confirmed that on our benchmarks the set representation of index groups performs better than symbolic ranges of consecutive indices. This is because our range evaluation often needs to obtain the range of a DS element with a given index[2], which is a simple inclusion check for sets, but requires additional computation of numerical bounds from symbolic expressions in other representations.

In this chapter, we consider only the natural order of evaluation (left-to-right with call-by-value), exactly as it syntactically appears in the program under analysis (modulo operators precedence). For this natural order, DS2L generates executable Scala code and for that code the analysis is sound. Our analysis can also be adapted to other, more efficient, evaluation orders, but determining that order is an orthogonal issue.

3.5. Experimental Evaluation

We evaluate our DS-based analysis in DS2L in terms of performance and accuracy, focusing on the following research questions:

RQ1 How does DS2L compare to state-of-the-art tools (on large programs)?

RQ2 How does DS-based abstraction affect the accuracy/performance tradeoff?

RQ3 Are error bounds reported by DS2L adequate?

Benchmarks We evaluate DS2L on the new benchmark set we collected (subsection 3.2.1). The original codes were written in different programming languages. We have translated them into our functional-style DSL for the purpose of our evaluation and validated our translation with testing. Table 3.1 displays in more detail which elements of our DSL were used in which benchmarks. Many of the benchmarks operating on vectors have been repurposed from controller loops used in previous work [84] and therefore have similar structure. As an artifact of this translation, our vector-based benchmarks use `fold` frequently.

For each benchmark, we create 12 variants by varying two parameters: size of the input DS and the specification granularity.

Size of the input DS. Input vectors are assigned 100 (small), 1k (medium), or 10k (large) elements. Input matrix sizes are 10x10 (small), 100x100 (medium) and 500x500 (large). For benchmarks where the size of a DS is predefined by the algorithm, we take the sizes closest to 10, 100 and 500 (for example, the input matrix for *fftmatrix* has 8x2, 128x2 and 512x2 elements for the small, medium and large setting, respectively). The benchmark input DS size influences the number of operations to be evaluated by the analysis. To give an unambiguous measure of complexity of the programs under

[2]For instance, taking a single element's range or a range of a group of elements when unrolling an iterator.

Benchmark	DSL usage						max #ops	Benchmark sizes		
	map	fold	slideRed.	enum*	rec	matMul	in line	small	medium	large
vector benchmarks										
avg		✓					1	101	1001	10001
variance		✓					3	202	2002	20002
stdDev.		✓					3	202	2002	20002
roux		✓					3	100	1k	10k
goubalt		✓					3	100	1k	10k
harmonic		✓					3	200	2k	20k
nonlin1		✓					7	200	2k	20k
nonlin2		✓					8	200	2k	20k
nonlin3		✓					6	200	2k	20k
heat1d	✓		✓		✓		5	257	1025	65537
fftvector	✓	✓		✓	✓	✓	4	96	9596	48636
matrix benchmarks										
pendulum		✓					4	404	4004	40004
alphaBlend.	✓						4	100	1k	250k
fftmatrix	✓	✓		✓	✓		8	64	6012	30204
conv.2d_sz3			✓				1	162	1458	118098
sobel3			✓				3	972	8748	708588
lorentz		✓					6	141	211	281
lyapunov	✓					✓	(20,200,1000)†	11	101	501
control.Tora	✓					✓	(20,200,1000)†	31	301	1501

Table 3.1.: Benchmarks description: usage of DSL functions and unrolled program sizes for different DS size configurations (in lines of code)

† The benchmark contains matrix multiplication, the maximum number of arithmetic operations in one line of code depends on the size of multiplied matrices. Reported values are for (small,medium,large) input DSs.

analysis, we report the sizes of unrolled programs in Table 3.1. The reported numbers are lines of code if all operations on DSs would be unrolled to scalar operations, i.e. the number of iterations times number of lines of code computing a scalar value inside each iterator. Since in the absence of DSs there would be no need for non-numerical functions as concatenation of vectors or changing the order of elements in a matrix, we only count lines of code with numerical operations and let-statements. Such unrolled programs could, for example, be used by state-of-the-art rounding error analyzers that operate on straight-line code. Additionally we report the maximum number of arithmetic operations in one line of unrolled code.

Our goal is to efficiently analyze *large* benchmarks. We include small and medium sizes for completeness and to demonstrate scalability, but do not consider DS2L to be necessarily the analysis tool of choice for these.

Range specification granularity. We vary the amount of individually specified ranges per DS. The input ranges are specified with either one, i.e. the same, interval for all elements (*AllSame*), different intervals for all elements (*AllDiff*), or for some. When specifying individual ranges for subsets of elements we vary the amount of new range specifications to be 10% and 30% of the input DS size (*Diff10P* and *Diff30P*). For instance, if an input vector has 100 elements *Diff10P* configuration will have 10 additional range specifications, each with an arbitrary amount of elements in it, and the *Diff30P* will have 30 additional range specifications. To avoid any bias by using input ranges that are easier for the analyzer to compute with, we generate all input ranges randomly. Similarly, the amount of elements in one group with special ranges is determined randomly. The smaller ranges of more refined specifications are subsumed by the ranges in AllSame specification.

Experimental Setup To answer our research questions we evaluate differences in accuracy and performance between a baseline analysis, our new DS abstraction-guided analysis and state-of-the-art tools. To do so, we normalize the reported worst-case rounding error and the running time of the analysis (separately) with respect to a baseline (different for each comparison). Such a normalization is necessary since the running times and error magnitudes vary widely between different benchmarks due to their diverse complexity. We then evaluate the normalized worst-case errors and analysis times.

As running time, we use the reported analysis time of each tool. This is a subset of the total wall-clock running time and excludes, for instance, parsing of the input programs. Since the formats of the input programs differ widely, we consider the analysis time a more meaningful measure for a comparison. We report analysis time averaged over 3 runs. We consider that a tool failed on a benchmark if it either timed out with 30 minutes, reported an infinite error bound, or encountered some other error. Timeouts were always consistent across all runs on each configuration. Note that the timeout applies to the total running time, including parsing, pre- and post-processing of the results.

As accuracy measure, we use reported absolute worst-case rounding error bounds of each tool for double floating-point precision. For the 13 benchmarks where the return type is a vector or a matrix we take the maximum error of all output DS elements.

All experiments were run on an Intel Xeon machine with 8 CPUs @ 3.50GHz, 32G of RAM under the OS Ubuntu 22.04. We run both DS2L and a baseline straight-line code analysis in a JVM with 2G memory and 1G stack space.

3.5.1. State-of-the-Art Tools

We compare DS2L against the state-of-the-art rounding error analyzers Fluctuat [39] and Satire [57]. We choose these two tools specifically, because they are the only tools that natively support data structures and loops over them (Fluctuat), or that analyze straight-line code, but whose abstractions were designed specifically for large program sizes (Satire). In these two dimensions that are relevant for our comparison, Fluctuat and Satire are the state-of-the-art. Satire does include approximations such as not considering higher-order terms that technically affect its soundness, but we ignore this here. DS2L and Fluctuat are 'fully sound'.

We note that an entirely fair comparison is not possible due to the different input formats, as well as different implementation choices such as programming language in which the tools themselves are implemented. Each of our high-level benchmarks written in our functional DSL can be translated to Fluctuat's and Satire's imperative formats in different ways that each may or may not affect the results (no guidelines exist). We manually translate our benchmarks into the tool's input formats by choosing the way that we consider to be natural for a programmer, and so a regular user of the tools would choose, and validate the translation with testing.

In our comparison, we use relative performance and accuracy as a measure of success. DS2L and Fluctuat are deterministic and always report the same error bounds. On some benchmarks Satire reported slightly different errors, we take the largest reported error across the runs. Note that the differences were on the order of 10^{-12}, and taking the average or the smallest error across the runs does not affect the qualitative results.

Fluctuat

Fluctuat can both unroll loops internally and abstract the loop behavior by applying widening. We use the latest available version of Fluctuat provided to us in October 2022.

Fluctuat takes C-programs as input and is itself implemented in C. When translating our benchmarks, we tried to preserve as much functional-style semantics as possible, but had to give up the DS immutability and replace all recursive calls by loops. Furthermore, Fluctuat's library did not support a `max()` function required for implementing the ReLU function in the neural network benchmarks *lyapunov* and *contr.Tora*. We replaced the call to `max()` with an explicit if-then-else statement. Fluctuat does not have a dedicated way of specifying input ranges for data structures, only for scalar values. We therefore assign a range to each element separately, and use loops to assign repeating ranges for the *AllSame* specification. Each benchmark is implemented in a separate function that is called from `main`. We compare DS2L with Fluctuat on all 19 benchmarks.

We run Fluctuat with several different settings:

1. loop iterations are evaluated separately, results joined (merge over all paths—MOP—solution)

2. loops are unrolled until 50k iterations. The largest number of iterations in our benchmarks is 62.5k, however, Fluctuat's setting did not allow us to set the unroll limit higher than 50k.

3. loops are abstracted by widening, nothing is unrolled

4. automatic setting, where Fluctuat finds a suitable number of loop unrollings before applying joins and widening.

Out of all configurations the overall best results were achieved with MOP (which is effectively unrolling) and the explicit unrolling configuration. Fluctuat with MOP and unrolling has timed out less often than other configurations and whenever Flucutat computed non-trivial error bounds, they were exactly the same for all settings. Surprisingly, the automatic configuration of Fluctuat had the highest timeout rate: it failed to produce results within 30 minutes on 33% of specifications. The pure widening configuration performed better with only 16% rate of timeouts.

Since all other settings provided worse or the same results, we compare DS2L's results only to the MOP setting of Fluctuat.

Satire

We use the latest version of Satire available in the open-source GitHub repository in April 2023[3]. Satire's open-source benchmark set contains pre-processed large unrolled loops, but no original programs that were unrolled. Unfortunately, the original programs with loops were not available (upon request). We have therefore reverse-engineered the loops over data structures from their unrolled versions for two benchmarks *lorenz*, and *heat1d*. Additionally, we translated some of our benchmarks into Satire's input format, which is an imperative DSL that specifies floating-point precision for each variable assignment. We only compare the results on a subset of benchmarks, since we are required to manually unroll the loops, and translate functional operators into imperative code. This translation process is non-trivial, tedious and error prone, especially for complex functions.

Overall, we translated 9 benchmarks that contain a `fold` over an input vector. For these 9 benchmarks we used the same variations in configurations, described above: small, medium, large input DS sizes, and *AllSame, Diff10P, Diff30P, AllDiff* specification granularities. We took Satire's original benchmarks as is: *heat1d* had only one version, that corresponds to our input specification with small input DS and one input range for all elements. The *lorentz* benchmark was available in three different sizes of input DS (20, 30 and 40), and all of them had the same input range for all elements of DS (*AllSame*). In total, we have compared our results on 112 benchmark variations.

We ran Satire with its default parameters and both with and without abstraction. The version with abstraction predictably produced results faster and had fewer timeouts. We therefore compare to the version of Satire with abstraction enabled.

3.5.2. RQ1: Comparison to State-of-the-Art Tools

We compare relative performance and accuracy of state-of-the-art tools normalized against DS2L's results and provide cumulative values in Table 3.2. The values greater than 1 denote individual benchmarks where DS2L was faster (respectively, more accurate) than the state-of-the-art tool. For instance, value 24.41 means that DS2L is median 24.41 faster than Fluctuat. As it is ambiguous to compute the relative value if one of the tools did not report results, we do not include these cases into the minimum, median and maximum values. Instead we report the number of failures per tool (timeouts, infinite error bounds, overflows). We mark in bold the smaller number of fails per comparison, and median values where DS2L did better than competitors. Note that

[3]To be precise, we use the version with the commit hash 8a4816aac6fad4fb86c2af8dc8e634bf02912b90.

Benchmark size	Accuracy min	Accuracy median	Accuracy max	Performance min	Performance median	Performance max	# fails	# fails DS2L	total # of bench.
				Fluctuat					
Small	2.11e-07	0.557	3.55	0.07	0.36	7.22	**2**	10	76
Medium	2.83e-04	0.639	2.91	0.23	**1.98**	4636.52	22	**12**	76
Large	3.60e-02	0.555	2.91	0.66	**24.41**	339.55	60	**25**	76
				Satire					
Small	2.98e-07	0.737	3.54	6.33	**24.68**	449.33	6	6	38
Medium	2.07e-10	0.153	3.54	6.32	**39.90**	507.45	12	**8**	37
Large	3.94e-02	0.953	1.34	8.95	**259.64**	767.13	32	**10**	37

Table 3.2.: Relative accuracy/performance of state-of-the-art tools compared to DS2L with DS abstraction.

we provide comparison on small and medium benchmarks for completeness, while our focus lays on large benchmarks.

In addition to normalized values, we present absolute values of our experiments on large benchmarks in Table 3.3. 'TO' denotes timeouts, other times are reported in seconds. We additionally mark the benchmarks, for which a tool reported overflow or an infinite error bound. For the original Satire benchmark *lorentz*, the missing configurations *Diff10P, Diff30P, AllDiff* with individual ranges for input DS elements are marked with 'na' (non-applicable). Another original benchmark *heat1d* is only defined for a small size of input DS. We provide absolute experimental values for small and medium benchmarks in the appendix, section A.2.

Accuracy

As expected, state-of-the-art tools often computed tighter error bounds on small and medium benchmarks. However, DS2L was consistently more accurate on the *stdDeviation* benchmark, and the larger (among the two in our set) neural network *controllerTora*. Additionally, Fluctuat reports infinite errors on all medium and large-sized variations of the FFT filter (*fftvector*, *fftmatrix*), while DS2L successfully computes rounding error bounds. Both Fluctuat and DS2L implement—in principle—the same analysis *on the unrolled programs*, and the DS abstractions alone do not affect accuracy (see subsection 3.5.3). The differences in accuracy come from 1) the optimized evaluation of `folds`; 2) DS2L's use of intervals instead of affine arithmetic; and 3) internal implementation differences that for the closed-source Fluctuat are not evident. We note that both Fluctuat's and DS2L's reported errors are itself small, and thus practically useful.

Satire reported more accurate results for non-linear benchmarks. On two configurations where DS2L reported overflow for the small input DS size (*AllSame, Diff10P* for *nonlin1* and *Diff10P, Diff30P* for *nonlin2*), Satire successfully reported rounding errors. Predictably, on benchmarks where DS2L used over-approximation of `folds` Satire's

	AllSame		*Diff10P*		*Diff30P*		*AllDiff*	
Benchmark	error	time	error	time	error	time	error	time
				DS2L				
avg	5.82e-11	**1.90**	2.86e-11	**5.48**	1.87e-11	**19.73**	**1.51e-11**	**157.63**
variance	7.39e-05	**119.28**	1.92e-05	**248.64**	9.68e-06	**412.39**	6.37e-06	**1144.74**
stdDev.	9.01e+03	**118.98**	9.35e-06	**244.51**	4.86e-07	**410.50**	2.70e-07	**1141.30**
roux1	**7.21e-14**	**3.86**	2.46e-13	**10.78**	2.64e-13	**29.88**	2.32e-13	**184.44**
goubault	7.46e-14	**3.93**	8.39e-14	**9.92**	8.39e-14	**27.98**	8.39e-14	**172.90**
harmonic	3.64e-08	**6.55**	1.42e-08	**28.40**	1.13e-08	**100.32**	1.12e-08	**713.85**
nonlin1	*overflow*	-	*overflow*	-	*overflow*	-	*overflow*	-
nonlin2	*overflow*	-	*overflow*	-	*overflow*	-	*overflow*	-
nonlin3	1.08e+74	**766.73**	2.66e+73	**1446.47**	-	TO	-	TO
pendulum	4.69e+81	**1583.05**	-	TO	-	TO	-	TO
heat1d	1.14e-13	**222.82**	7.26e-14	**830.00**	7.14e-14	**874.28**	7.13e-14	**857.08**
conv.2d_size3	3.15e-10	**63.29**	2.88e-10	**111.15**	2.62e-10	**158.94**	-	TO
sobel3	*DivByZero*	-	*DivByZero*	-	*DivByZero*	-	*DivByZero*	-
fftmatrix	**4.39e-08**	325.93	**4.24e-08**	387.76	**3.95e-08**	386.29	**2.97e-08**	399.04
fftvector	**2.02e-08**	262.99	**1.62e-08**	266.15	**1.13e-08**	269.89	**9.84e-09**	278.46
lorentz	3.42e-12	2.33	3.27e-12	2.32	3.27e-12	2.21	1.25e-12	2.27
alphaBlend.	3.14e-13	**1.83**	3.14e-13	**47.81**	3.14e-13	**225.50**	-	TO
contr.Tora	2.61e-04	**386.38**	-	TO	-	TO	-	TO
lyapunov	7.02e-08	**104.21**	-	TO	-	TO	-	TO
				Fluctuat				
avg	**2.57e-11**	516.00	**1.83e-11**	475.50	**1.63e-11**	462.00	**1.51e-11**	490.50
variance	-	TO	-	TO	-	TO	-	TO
stdDev.	-	TO	-	TO	-	TO	-	TO
roux1	2.10e-13	1310.00	**2.10e-13**	1302.00	**7.52e-14**	1212.50	**1.09e-13**	1295.50
goubault	**6.50e-14**	695.50	**6.45e-14**	726.50	**6.45e-14**	711.00	**1.87e-14**	716.50
harmonic	-	TO	-	TO	-	TO	-	TO
nonlin1	-	TO	-	TO	-	TO	-	TO
nonlin2	-	TO	-	TO	-	TO	-	TO
nonlin3	-	TO	-	TO	-	TO	-	TO
pendulum	-	TO	-	TO	-	TO	-	TO
heat1d	-	TO	-	TO	-	TO	-	TO
conv.2d_size3	-	TO	-	TO	-	TO	-	TO
sobel3	-	TO	-	TO	-	TO	-	TO
fftmatrix	∞	71.33	∞	71.33	∞	71.67	∞	71.00
fftvector	∞	35.67	∞	37.67	∞	116.00	∞	107.33
lorentz	**1.23e-13**	**2.00**	**1.21e-13**	**2.00**	**1.21e-13**	**2.00**	**1.02e-13**	**1.50**
alphaBlend.	-	TO	-	TO	-	TO	-	TO
contr.Tora	-	TO	-	TO	-	TO	-	TO
lyapunov	-	TO	-	TO	-	TO	-	TO
				Satire				
avg	3.47e-11	1456.27	2.72e-11	1421.63	2.22e-11	1419.89	2.02e-11	1410.32
variance	-	TO	-	TO	-	TO	-	TO
stdDev.	-	TO	-	TO	-	TO	-	TO
roux1	-	TO	-	TO	-	TO	-	TO
goubault	-	TO	-	TO	-	TO	-	TO
harmonic	-	TO	-	TO	-	TO	-	TO
nonlin1	-	TO	-	TO	-	TO	-	TO
nonlin2	-	TO	-	TO	-	TO	-	TO
nonlin3	-	TO	-	TO	-	TO	-	TO
lorentz	1.35e-13	1327.65	*na*	*na*	*na*	*na*	*na*	*na*

Table 3.3.: Experimental results on large benchmarks. Reported error bounds are rounded to two digits after decimal point, time is in seconds. “TO” denotes a timeout, “na” stands for non-applicable. **Bold** marks ‘winning’ values.

reported errors were also smaller. However, on all linear benchmarks except *harmonic* DS2L's accuracy could be recovered by using a non-optimized evaluation of `fold` (while still being faster than Satire, but by a smaller factor). Despite the over-approximation, DS2L was consistently more accurate on the linear *goubault*. Interestingly, DS2L was 3x more accurate than Satire on its original benchmark *heat1d* [4].

Performance

The performance comparison shows that DS2L scales better to larger programs: it reports results on 46% more *large* benchmarks than Fluctuat and on 59% more than Satire. Additionally, DS2L is faster than Fluctuat on most large and medium-sized benchmarks with a median speedup factor of 25x and 2x respectively. A notable outlier is *alphaBlending*, where DS2L is **4636x** faster than Fluctuat. This is due to the benchmark's internal structure: it contains element-wise operations on matrices, where DS2L's abstraction is particularly efficient.

Satire timed out more often than DS2L on all sizes of benchmarks, and particularly on large benchmarks where it failed to report results on all benchmarks except *avg* and *lorentz* (see Table 3.3). Moreover, Satire was slower than DS2L by at least **6x** and median **36x** across different sizes of benchmarks including its original benchmarks *heat1d* and *lorentz*.

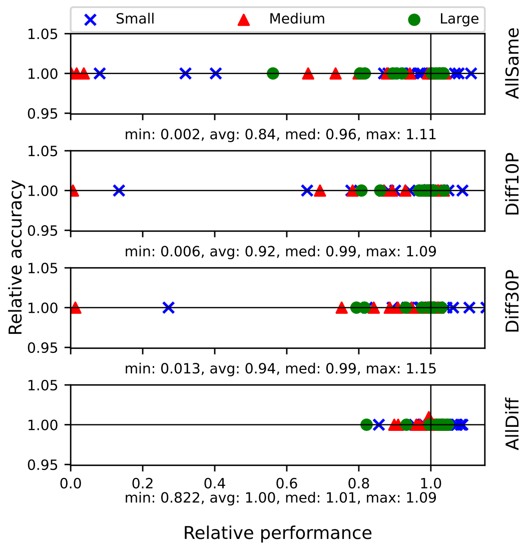
RQ1 Conclusion: Based on our experimental data, we conclude that DS2L is *significantly faster* than Satire and specifically *scales better* than Fluctuat and Satire to larger programs and is consequently able to report an error for more and larger benchmarks. While DS2L is often less accurate than Fluctuat and Satire, it still produces meaningful accuracy bounds.

3.5.3. RQ2: Accuracy/Performance Tradeoff with DS-based Abstraction

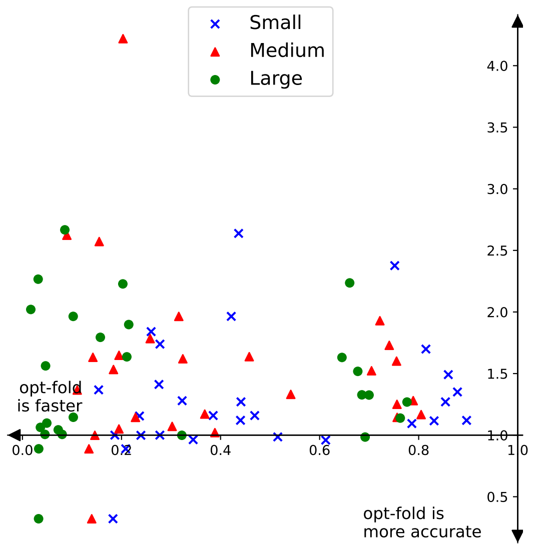
Our analysis differs from the analysis of the unrolled programs in two main points: it leverages the DS abstraction, and optimizes the evaluation of `folds` (subsection 3.3.4). We evaluate the effect of these differences on both accuracy and performance. We split this evaluation into two parts: first, we check the effect of the DS abstraction alone, then we examine the benefits of the optimized `folds`.

DS Abstraction. First, we compare the DS abstraction-based analysis of DS2L to a baseline analysis that works on unrolled code. To avoid confounding factors such as programming language choice, analysis type etc., we do this comparison on a baseline analysis that we implement within DS2L itself and that shares exactly its analysis for straight-line code. We denote this baseline analysis by BASE. BASE internally unrolls all operations, and thus just like DS2L does not explicitly construct an AST for the entire program, as this may be unnecessarily costly and bias the results. Thus, when comparing DS2L and BASE, the only difference consists in using the corresponding DS

[4]This comparison is with respect to the original version of *heat1d* that corresponds to *AllSame* configuration and small size of benchmarks. The absolute values are available in Table A.2.



(a) DS2L with abstraction vs. the baseline analysis. Smaller values along X-axis are better.



(b) DS2L: Optimized fold vs. unrolled

Figure 3.5.: Relative performance/accuracy of DS2L in various configurations

abstractions during the analysis. For the purpose of DS abstraction evaluation we use the version of DS2L without over-approximation on `folds`.

Specifically, we compare normalized analysis time and normalized computed worst-case absolute rounding errors per benchmark for each of its 12 variants. Figure 3.5a summarizes the results, smaller values on both axes are better. The x-axis shows relative analysis time of the DS abstraction analysis to the baseline, values with $x < 1$ denote benchmarks, on which DS2L was faster than BASE. The y-axis represents relative accuracy, values with $y = 1$ show that the worst-case rounding errors reported by DS2L were exactly the same as for BASE. We provide average, median, minimum and maximum relative analysis times for each specification.

For most benchmarks applying the DS abstraction has improved the analysis performance. Predictably, the performance boost was stronger for coarser specifications and close to none on the *AllDiff* specification that assigns each DS element an individual input range. We manually checked the cases where DS2L was slower than BASE. For these cases the absolute time difference is under 0.3 seconds on small and medium configurations (up to 15% of analysis time), and under 72 seconds on large configurations (at most 5% of the analysis time). We attribute this to the normal variation in running times and do not see it as a systematic problem.

The computed errors were the same for DS2L and BASE on all benchmarks. This result confirms our expectation that the DS abstractions (without `fold` optimizations) do not change the semantics and therefore do not affect computed rounding errors.

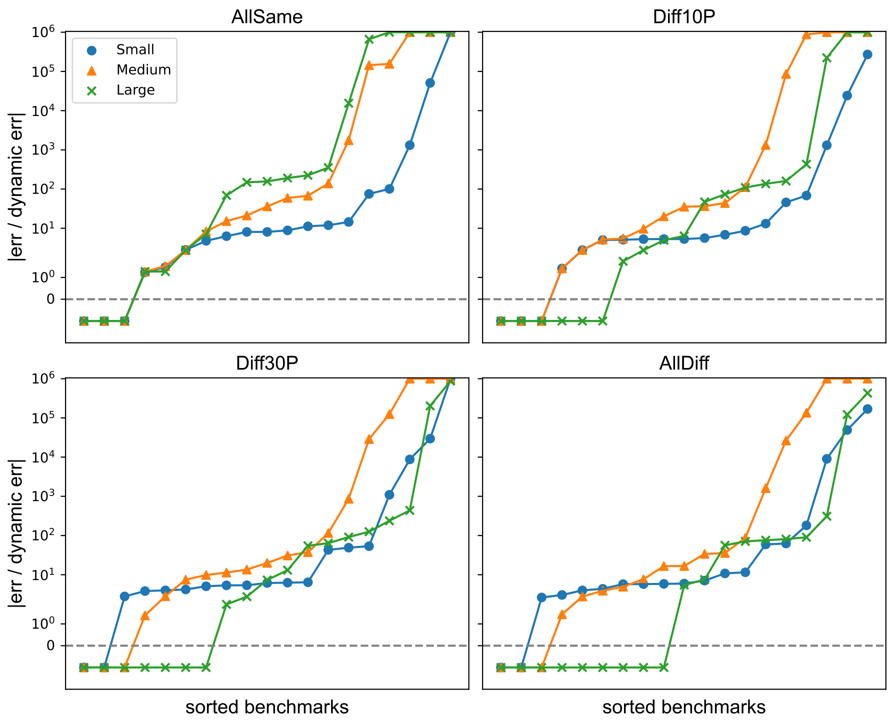


Figure 3.6.: Relative difference between worst-case errors by DS2L and dynamic errors

Optimized folds. We evaluate the effect of our `fold` optimization on top of DS abstraction improvements in Figure 3.5b. We compare the relative accuracy and performance on benchmarks with `fold` with and without the optimization. As expected, the optimized `fold` evaluation is faster and less accurate on most benchmarks, these are the points above the x-axis and to the left of the y-axis. The effect is more pronounced on the large benchmarks. Interestingly, in some cases the optimized evaluation reported smaller error bounds despite introducing an over-approximation of ranges. Upon closer inspection we note that some of the randomly generated input range bounds cannot be exactly represented in floating points, hence performing an unrolled error computation on such ranges will include the bounds' rounding error and magnify it (artificially) in subsequent iterations. The accuracy can thus improve in cases where the over-approximated ranges were exactly representable in floats, while corresponding element's input ranges were not.

RQ2 Conclusion: The DS abstraction alone improves the analysis' performance while having no effect on the accuracy. A user may further improve the performance by providing a coarser specification or enabling the optimized evaluation of `folds`, which trades off accuracy for performance.

3.5.4. RQ3: Adequacy of DS2L's Error Bounds

We evaluate the adequacy of the error bounds reported by DS2L with respect to the actual errors occurring in our benchmarks. To obtain an estimate of realistic errors we have generated Scala code and ran it on all benchmarks in two variations: with double floating-point precision and with 300-bit MPFR numbers, and computed the difference between them. For every benchmark, we collected these dynamic errors on 10^5 inputs sampled from the individually specified ranges in each specification granularity (*AllSame*,*AllDiff*,*Diff10P*, and *Diff30P*) and took the maximum value. Such dynamically obtained errors provide a lower bound on the worst-case error and they are a good measure for the *magnitude* of errors actually occurring in the program.

We plot the results of our comparisons on cactus plots, the results are sorted across all benchmarks and two points with equal values on the X-axis do not necessarily correspond to the same benchmark. Figure 3.6 shows sorted differences between dynamic absolute errors and the errors reported by DS2L. Due to the variety in complexity of our benchmarks, errors have different magnitudes, to have a better overview we show the relative differences computed as $|\text{DS2L}'s\_error/dynamic\_err|$. Whenever DS2L did not report an error, a comparison was not possible and we set the value to a dummy $y = -1$.

Among the 181 benchmarks for which DS2L reported error bounds (excluding 47 timeouts and overflows), 89% of reported errors were within six orders of magnitude from the under-approximated dynamic errors (below the $y = 10^6$). As expected, the difference was larger for large benchmarks, since over-approximation generally accumulates with every iteration. We also noticed a clear trend that the over-approximation reduces for more fine-grained input specifications.

We have manually inspected the benchmarks where the differences between worst-case errors reported by DS2L and their dynamic under-approximations were more than six orders of magnitude. Expectedly, they were non-linear benchmarks: *nonlin3*, *pendulum*, *controllerTora*. This is due to large over-approximations committed by the baseline analysis of DS2L and is not specific to our method. These benchmarks contain folds with non-linear bodies that are ultimately unrolled into a huge straight-line non-linear program. Those are known to be challenging for state-of-the-art rounding error analyses [13]. Another outlier *stdDeviation* contains a square root operator applied to a range starting from zero, for which today's rounding error analyses also report pessimistic bounds.

RQ3 Conclusion: 89% of the errors reported by DS2L are reasonably close to the under-approximated dynamic errors. Overall, we conclude that DS2L reports meaningful and adequate error bounds.

3.6. Related Work

Static Analysis Besides Fluctuat [39] and Satire [57], several other tools exist for computing guaranteed upper bounds on rounding errors; Gappa [90], Daisy [14], FPTaylor [71], Real2Float [17], Rosa [13] and PRECiSA [73]. These either implement a dataflow analysis based approach very similar to Fluctuat's or an optimization-based approach similar to Satire. Most of the research has focused on analyzing straight-line numerical expressions as accurately as possible, i.e. computing error bounds as close to the actual errors as possible. Of these, Satire has been shown to be most scalable [57].

A few of these tools can also handle limited programs beyond straight-line expressions. As already discussed, Fluctuat [39] can handle loops via unrolling or with widening, but as we observed widening has limited success with a complex analysis such as the one used to analyze floating-point rounding errors. Rosa [13] provides a more efficient way to bound rounding errors in bounded loops than complete unrolling for a specific type of while loops, but requires invariants about the variable's ranges to be given. Rosa [13], Fluctuat [37] and PRECiSA [73] also support (simple) conditional branches where they also compute the error due to diverging executions between then- and else-branches, in addition to rounding errors of each individual branch. Alternative techniques exist to detect how often a finite-precision computation takes the wrong path [38]. Such techniques are complementary to DS2L's handling of data structures.

Dynamic Analysis In contrast to sound analysis tools, dynamic analysis tools for floating-point programs have fewer restrictions on the input programs and generally handle whole programs, including loops, conditional branches and data structures. Additionally, dynamic methods can be applied both on source code [33, 91] and on already compiled binaries [28]. Typically, they execute a program on particular floating-point inputs side-by-side with a shadow execution in a higher precision [28,30,32], for instance, implemented using arbitrary-precision arithmetic that serves as an approximation of the ideal real-valued execution. Shadow executions in high precision may, however, incur a high overhead and alternative approaches were proposed to use as an oracle for the ideal computation [29, 31, 33]. By their nature, dynamic analyses cannot compute guaranteed bounds on errors, only an estimate of the errors for inputs tried. Several tools use dynamic analysis to identify inputs that result in particularly large rounding errors [30, 32–35]; they employ various techniques to minimize the likelihood of missing interesting inputs [30, 32, 92]. Symbolic execution has also been used to find inputs that cause overflow or large precision loss in floating-point programs [35, 93–95]. Recent work also combines dynamic and static analysis for identifying, or showing conditional absence of large rounding errors in larger floating-point programs [96].

Other Finite-Precision Analyses Naturally, quantifying the rounding errors is not the only possible objective for analysis of finite-precision programs. Many of the methods are concerned with detecting the presence of special floating-point values in the program, for instance, a deductive verifier KeY [97]. Abstract interpretation based analyzers such as the industrial-strength Astrée [27], or implementations of different numerical domains with varying performance characteristics [98] such as Apron [99] and ELINA [100] can prove safety of floating-point programs, i.e. the absence of overflows, division-by-zero or out-of-bounds errors by bounding the ranges of variables. They do not, however, report rounding error bounds.

Functional Properties Apart from quantifying errors and detecting exceptions, one can also verify functional properties of programs with loops over arrays. Such properties state, for instance, that sorting algorithms indeed output monotone arrays [101, 102], or assert particular relations between array cells [103]. Other tools generate symbolic invariants to prove safety properties about ranges of array elements [104, 105] and verify memory access permissions [106]. The abovementioned methods, however, are not directly comparable with our analysis as they operate on integer programs, do not take into account rounding errors and require that the target property is specified by users in the form of an assertion.

Array Languages In this chapter, we proposed one possible input language with functional iterators, however, one may consider alternative inputs. Several languages have been created specifically for operations on arrays: APL [107] and its dialects [108, 109], BQN [110]. They specialize on the efficient execution of array operations, though typically do not support multiple precisions and use an entirely different way of writing the operations that may be unfamiliar to an average programmer with imperative and functional languages background.

3.7. Conclusion and Future Work

We have shown that computing rounding errors over a functional representation of floating-point list programs can be beneficial for analysis performance, by leveraging implicit semantic information present in the high-level representation. Conceptually, our idea appears simple — "just" use a functional input language — and yet, it has not been pursued before. We view this simplicity as a strength, but also note that an effective realization of this idea required a careful design of the DSL and the analysis, as well as substantial implementation effort. Our analysis can generally handle more, and especially larger benchmarks, though some of this performance benefit comes at a trade-off with analysis accuracy.

Optimized Evaluation We could further improve scalability of our analysis by applying optimized evaluations of `fold`-like iterators more often. At the moment we only optimize evaluation of `folds` if the loop body is a linear function. For a more general case, we would have to instrument DS2L with an additional analysis computing a closed-form equation for the loop body, some form of a summary [111]. Computing useful summaries for functions is a challenging task and is likely to work only for some restricted subset of functions. Whenever the summary analysis would succeed and generate an equation, we could use it to over-approximate iterations on elements with the same ranges.

Extended DSL Our functional DSL serves as a starting point for further development, a natural direction of future work would be to extend the DSL with more operators. One more possible extension would be to include data structures of higher dimensions. Our analysis should be applicable to the N-dimensional immutable arrays (tensors) as-is on most operators. Supporting additional operators requires engineering effort and an extension of the analysis' abstract transfer functions.

Alternative Input Formats While functional DSL is convenient for specifying new algorithms, it may be difficult to translate legacy imperative code into it. A possible solution may define parsers for alternative input formats. The hard constraints to use our analysis are:

1. all variables and data structures must be immutable. This can be (partially) achieved by bringing the code into a static single assignment (SSA) form [112].
2. Semantic information about iterators must be available, such as how data flows between iterations and how it and its shape is modified. This information can potentially be obtained by applying semantic analysis to the original code (for instance, with a polyhedral model of a loop [113]) and then mapping the results to the intermediate representation language of DS2L.

Note that the translation of imperative programs into functional is known to be challenging [114].

Code Generation One more direction of future work could be to optimize code generated by DS2L after the analysis. At the moment, DS2L only generates *functional* Scala code, which it performs by directly translating the DSL operators and inlining every occurrence of every operator's implementation. As a result, some of the generated code is unnecessarily repetitive. A potential improvement would be to perform optimizing transformations (for instance, loop fusion [115]) before generating the final code. Ideally, we would like to generate efficient code for both functional and imperative programs, as C frequently appears in embedded systems projects. A translation of functional code (as in DS2L's intermediate representation) into imperative is non-trivial; some functional operations do not have a direct and unambiguous equivalent in C.

An important thing to keep in mind when optimizing the generated code (for both functional and imperative programs) is that the order of evaluation in finite-precision operations matters. To keep the guarantees on error bounds as reported by the analysis, the order of computations has to match exactly the order in which operations were evaluated by the analysis. Existing work on efficient code generation from functional specifications successfully deals with redundant intermediate data structures and operations, however, it requires manual annotation [116–119] and potentially rearranges the operations [120]. The Stainless verification system [121], in contrast, focuses on preserving the properties that it proved; it has been used to translate Scala code into imperative, but only for integer programs with mutable data structures [122]. While these techniques and tools are not directly applicable to DS2L, they present an interesting starting point for further research.

4. Inductive Invariants for Unbounded Loops

Providing a *tight and sound* estimate for rounding error bounds in the presence of loops is challenging, because errors may grow indefinitely, and for a general case the only sound bound is a trivial range of $[-\infty, \infty]$. Various applications of numerical loops involve a priori unbounded loops, for instance, when modeling and simulating a continuous process, in scientific computing and embedded control systems. Since infinite loops cannot be fully unrolled, they have to be abstracted, for example, with inductive invariants [123]. An invariant is *inductive* if it holds before entering the loop and after every iteration. Such invariants are needed to reason about a program's safety and they are a necessary ingredient for bounding rounding errors (on loops) with some techniques [13].

Given an invariant, it is (relatively) easy to prove (or disprove) that it is inductive using, for instance, existing SMT solvers [124, 125]. Generating an invariant, on the other hand, is a challenging task, both manually and with automated tools [40–43]. Moreover, finding a *finite-precision invariant* is even harder (compared to real- or integer-valued) because it has to account for overflows, rounding errors and special values (such as infinity and not-a-number in floating points). Additionally, to be able to prove the inductiveness of a finite-precision invariant and for it to be useful in a subsequent rounding error estimation, the invariant should be tight, i.e. closely cover the actual values appearing in the loop execution.

Many tools generate inductive invariants for programs over integers to prove a program's safety [43, 46, 126–128]. The safety property holds if user-specified unsafe states (ranges of variables) are proven to be unreachable. As a by-product of this proof, tools compute inductive invariants that over-approximate reachable ranges of variables, but they necessarily use the unsafe states specification to guide the invariant search. However, for finite-precision loops where the goal is to compute as tight invariants as possible, specifying unsafe states essentially amounts to finding the invariant itself.

Alternatively, one could use static analysis with abstract interpretation that generally does not require target ranges. However, some existing abstract interpretation tools still require an approximate target range to compute *tight* invariants [46] and others do not generalize well: they are limited to linear loops due to the underlying techniques [47], do not always produce invariants that satisfy the precondition [48], or rely on user input, such as invariant templates [44].

In this chapter, we propose a practical approach to synthesize inductive invariants based on a loop's simulation. Our invariant synthesis method handles both linear and non-linear loops and requires minimal input from the user: only a loop body and input ranges for variables. This generality, naturally, comes at a certain cost. Unlike other approaches [46–48], our simulation-based approach does not provide completeness guarantees, but instead it has wider applicability. Despite the absence of provable completeness, we empirically observed our algorithm to be remarkably effective.

Our algorithm employs a combination of simulation and the counterexample-guided synthesis approach (CEGIS). First, it randomly samples points from input ranges and simulates several loop iterations. Given the simulated values, we use curve fitting to propose the first invariant candidate. This candidate is checked with an off-the-shelf SMT solver that either accepts the invariant or provides a counterexample. We cannot directly query the solver for the invariant (or its coefficients), because that would require using existential quantifiers, which today's SMT solvers do not handle well for finite-precision theories.

If a solver generates a counterexample, our synthesis algorithm incorporates the counterexample into the next invariant candidate. Our algorithm generalizes the counterexample and produces some additional points, repeats the simulation starting from these counterexample points and fits the polynomial curve again. This process it repeated until either the invariant is confirmed, or the algorithm times out (with a custom value, set to 20 minutes in our experiments).

The key idea behind our algorithm is that *numerically stable* loops can tolerate a certain amount of noise. Because of this tolerance, there is no single "ground truth" invariant, but rather multiple invariants of a similar form. We leverage this idea in the random sampling and simulation, but also how we query SMT solver to check the invariant.

Modern SMT solvers are able to handle some floating-point queries and with some preprocessing also fixed-point arithmetic (encoded as bit-vectors). However, such queries are inefficient, especially when non-linear operations are involved. We therefore do not use bit-vector or floating-point theories in the SMT queries. Instead, we first search for a real-valued invariant, then add bounded non-deterministic noise that represents rounding errors, and verify whether the invariant still holds with additional noise. This approach accomplishes two things: 1) it allows us to use the reals theory for non-linear computations in queries to SMT [129] that is generally more efficient than the finite-precision theories, and 2) it decouples the loop under analysis from its implementation details, i.e. the exact choice of finite precision. We determine the ranges for non-deterministic terms by applying rounding error analysis to each loop statement. Our algorithm is independent from the exact instance of the analysis; in the prototype implementation we use the analysis from Daisy [14].

To compare with invariants generated by state-of-the-art tools, for the remainder of this chapter we will focus on floating-point loops. Note that for synthesizing a fixed-point invariant the required steps are exactly the same.

Following previous work, we generate inductive invariants that cover a wide range of numerical loops—in a form of an ellipsoid. Our invariants have a form of a polynomial

```
x ∈ [0.0, 0.1]
y ∈ [0.0, 0.1]

while (true) {
  x := x + 0.01 * (-2*x - 3*y + x*x)
  y := y + 0.01 * (x + y)
}
```

(a) Example benchmark

$$-0.03x - 0.1y + 0.44x^2 + xy + 0.86y^2 \leq 0.02$$

$$x \in [-0.5, 0.3]$$
$$y \in [-0.2, 0.4]$$

(b) Generated invariant

Figure 4.1.: Running example

inequality $\mathcal{P}(x) \leq 0$ in conjunction with intervals $\mathcal{R}_i$ for each individual variable x_i:

$$\mathcal{P}(x_1, ..., x_n) \leq 0 \land x_1 \in \mathcal{R}_1 \land \ldots \land x_n \in \mathcal{R}_n$$

We implemented our synthesis algorithm in a Python library Pine and evaluated it on 30 numerical loops from various domains. Compared to the most closely related state-of-the-art tools SMT-AI [48] and Pilat [46], Pine was able to produce tighter invariants on 70% of linear benchmarks and generated invariants for 6 non-linear benchmarks where other tools failed.

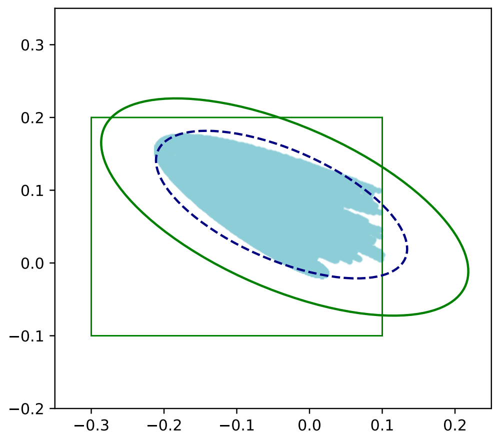
Contributions To summarize, this chapter describes the following contributions:

- the first general synthesis algorithm for inductive invariants that handles linear and non-linear, floating-point and fixed-point unbounded loops
- the open-source implementation of the algorithm in the Python library Pine available at `https://github.com/izycheva/pine`
- an extensive experimental evaluation of Pine and its comparison to state of the art.

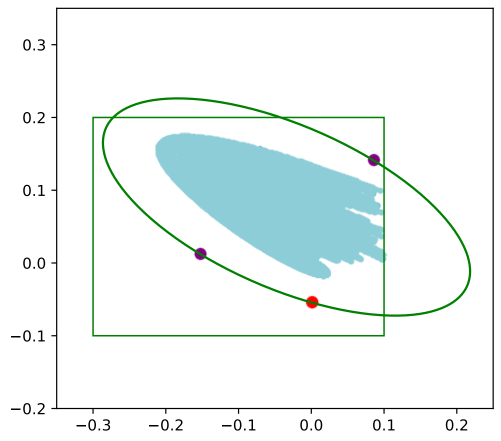
4.1. Overview

Before explaining our invariant synthesis algorithm in detail, we illustrate it at a high-level on an example. Figure 4.1a shows our example loop that simulates a dynamical system together with the precondition on the loop variables.

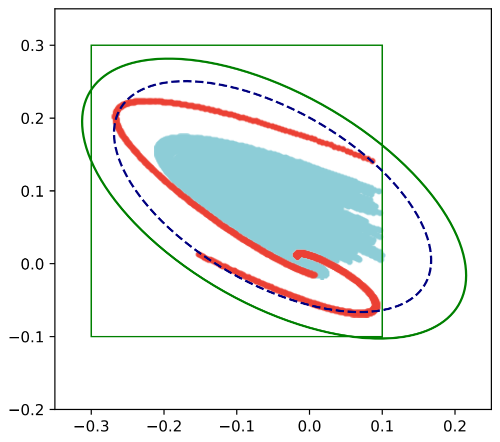
Pine starts by simulating the loop to collect a set of concrete points that an inductive invariant definitely has to include. For this, Pine samples $m = 100$ random values from the input ranges $x \in [0, 0.1]$ and $y \in [0, 0.1]$, and executes the loop $n = 1000$ times for each point. Sampled points are shown in light blue in figures 4.2a-4.2c. Since we are looking for a convex invariant, Pine next computes the convex hull of the sampled points. This reduces the number of points to consider and gives us an initial estimate of the shape of the invariant.



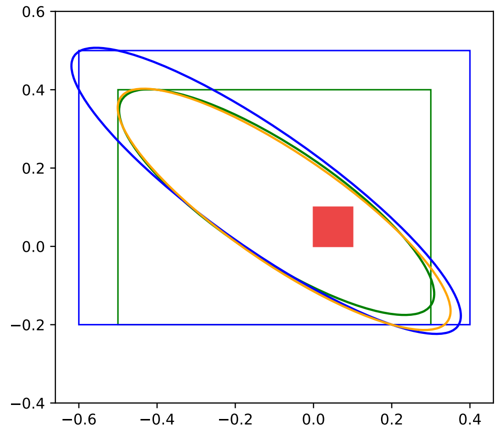
(a) First candidate invariant



(b) Counterexample and symmetric points



(c) Second candidate invariant



(d) Multiple invariants

Figure 4.2.: Non-linear benchmark candidate invariants

We consider invariants that include variable ranges and a shape enclosing all values expressed as an ellipsoid, i.e. a second degree polynomial inequality. We obtain the polynomial coefficients by computing the minimum volume ellipsoid enclosing the convex hull, and the variable bounds from the minimum and maximum values seen in the sampled points:

$$-0.0009x - 0.004y + 0.0103x^2 + 0.021xy + 0.0298y^2 \leq 5.4 \cdot 10^{-5} \wedge$$

$$x \in [-0.2098, 0.0976], y \in [-0.0159, 0.1723]$$

The computed ellipsoid is depicted in Figure 4.2a by blue dashed ellipse. We observe that this candidate invariant is noisy, for instance, some sampled points are not included. To remove (a part of) this noise, we scale and round the (normalized) polynomial coefficients and the range bounds (the latter is rounded outwards). Obtained ellipsoid and ranges form the first candidate invariant (marked green in Figure 4.2a):

$$-0.03x - 0.13y + 0.35x^2 + 0.7xy + y^2 \leq 0.01 \wedge x \in [-0.3, 0.1], y \in [-0.1, 0.2]$$

Pine uses an off-the-shelf SMT solver (Z3) to check whether this candidate invariant is inductive. For our candidate invariant the check fails and the solver returns a counterexample $C_1 : (x = 0.0, y = -0.0542)$ (red dot in Figure 4.2b). By counterexample, we mean a point that itself satisfies the candidate invariant, but after one loop iteration results in a point for which the invariant no longer holds. In our example C_1 satisfies the candidate invariant, but after one iteration we obtain $C_1' : (x = 0.001626, y = -0.054742)$ that violates the invariant.

Pine uses this counterexample to refine the candidate invariant. However, instead of recomputing the convex hull and ellipsoid shape immediately, we generate additional counterexamples in order to not bias the shape in a single direction that is (randomly) determined by the solver's counterexample. In particular, Pine computes counterexamples that are symmetric to C_1 along the symmetry axes of the ellipsoid and satisfy the candidate invariant. Figure 4.2b shows the counterexamples generated for our running example (purple dots).

Pine then uses another round of simulation, starting from the set of counterexamples, to obtain a new set of points that need to be included in an invariant (by transitivity, if a counterexample point is included after one loop iteration, then the points after additional iterations also have to be included). The new set of points is then used to generate the next candidate invariant. Figure 4.2c shows simulated points in red, and the new candidate invariant in green. Note that Figure 4.2c contains three simulation traces - one for each counterexample, and the traces originated from the bottom left counterexample and C_1 coincide.

Pine repeats this iterative process until either an invariant is found, or a maximum number of refinement iterations is reached. For our example, Pine finds an inductive invariant (shown green in Figure 4.2d) after 6 iterations.

The invariant so found holds for a real-valued loop, i.e. when the loop body is evaluated under real arithmetic. The last step of Pine's algorithm is to verify that the invariant also holds under a floating-point loop semantics. To do this, Pine uses an off-the-shelf analysis tool to get the worst-case rounding error bound for each expression in the loop body. The errors are then added as nondeterministic noise terms to the loop, and the invariant is re-checked by the SMT solver. For our running example, this check succeeds, and the following invariant is confirmed:

$$-0.03x - 0.1y + 0.44x^2 + xy + 0.86y^2 \leq 0.02 \ \wedge x \in [-0.5, 0.3], y \in [-0.2, 0.4]$$

Figure 4.2d shows several invariants generated by Pine for our example, for different parameters of its algorithm. Note that these invariants are similar, but differ slightly in shape and volume. The range component of the invariant is shown by the green and blue boxes; the red box denotes the input ranges.

4.2. Problem Definition

The input to our algorithm is a loop body together with a precondition. We consider simple non-nested loops given by the following grammar:

```
L ::= while(true){ B }
B ::= if (G) S else S | S
S ::= ϵ | xᵢ := p(x₁, ..., xₙ) + uⱼ; S
G ::= * | p ≤ 0
```

In each iteration, the loop updates a set of variables $x_i \in \mathcal{X}$. Note that this update is atomic for the whole loop body, i.e. the update for the value of x_i will only take effect in the next loop iteration. The right-hand-side of each assignment consists of polynomial expressions p in the loop variables together with an (optional) nondeterministic noise term u_j, which is bounded in magnitude. Note that u_j in the input program does not reflect rounding errors, it denotes any additional noise, e.g. input error from sensor values. We model rounding errors in a similar fashion later in the algorithm, but a user does not have to state them manually. The loop body can include a top-level conditional statement, which can also be used to express the loop exit condition. The conditions of the if-statement can either be nondeterministic choice or a polynomial inequality. We note that adding support for more complex conditions as well as nested and chained if-statements would only affect the way we parse the loop and encode it in the SMT query and is not a fundamental limitation of our algorithm.

The precondition specifies the initial ranges for all variables x_i, as well as bounds on the nondeterministic noise variables: $x_i \in [a_i, b_i]$, $u_j \in [c_j, d_j]$. The loop and noise variables take values in the set $\mathbb{F}$ of floating-point values. Then the semantics of a loop body b is given by the transformation $[\![b]\!] :: (\mathcal{X} \to \mathbb{F}) \to 2^{(\mathcal{X}\to\mathbb{F})}$, which is defined by

$$\begin{array}{lcl}
[\![\epsilon]\!]\,\rho & = & \{\rho\} \\
[\![x_i := p + u_j; s]\!]\,\rho & = & \bigcup\{[\![s]\!](\rho \oplus \{x_i \mapsto p(\rho) + u\}) \mid u \in [c_j, d_j]\} \\
[\![\mathbf{if}\,(*)\,s_1\ \mathbf{else}\ s_2]\!]\,\rho & = & [\![s_1]\!]\,\rho \cup [\![s_2]\!]\,\rho \\
[\![\mathbf{if}\ \ (p \le 0)\,s_1\ \mathbf{else}\ s_2]\!]\,\rho & = & \{\rho_1 \in [\![s_1]\!]\,\rho \mid p(\rho) \le 0\} \cup \\
 & & \{\rho_2 \in [\![s_2]\!]\,\rho \mid p(\rho) > 0\}
\end{array}$$

Here, $p(\rho)$ denotes the value of the polynomial p for the variable assignment ρ under the floating-point arithmetic semantics specified by the IEEE 754 standard [64]. The set of initial program states is given by

$$\mathsf{Init} = \{\rho : \mathcal{X} \to \mathbb{R} \mid \forall x_i \in \mathcal{X}.\, \rho(x_i) \in [a_i, b_i]\}$$

Our goal is to find an inductive invariant $\mathcal{I}$ such that

$$\mathsf{Init} \subseteq \mathcal{I} \quad \wedge \quad \forall \rho \in \mathcal{I}.\ [\![b]\!]\,\rho \subseteq \mathcal{I} \tag{4.1}$$

i.e., $\mathcal{I}$ subsumes the initial states and is preserved by each iteration of the loop. We consider convex invariants given by a polynomial inequality together with ranges for variables:

$$\mathcal{I} = \{\rho \mid \mathcal{P}(\rho) \leq 0, \rho(\mathsf{x}_i) \in \mathcal{R}_i = [l_i, h_i]\}$$

The goal is thus to find the coefficients of the polynomial $\mathcal{P}$ and the lower and upper bounds (l_i, h_i) for the variables of the loop. In this paper, we consider polynomials $\mathcal{P}$ of degree two, although our algorithm generalizes to higher degrees. We observe that second degree polynomials are already sufficient for a large class of loops.

Additionally, we are interested in finding as small an invariant as possible, where we measure size by the volume enclosed by an invariant. We note that the ellipsoid (the polynomial inequality), is not only needed to prove the inductiveness of many invariants, but it can also enable more accurate verification based on our inductive invariants, for instance by techniques relying on SMT solving. For this reason, we do not only measure the volume as the size of the box described by $\mathcal{R}$, but rather as the intersection between the box and the ellipsoid shape, which can be substantially smaller.

4.3. Algorithm

Figure 4.3 shows a high-level view of our invariant synthesis algorithm. The input to the algorithm is a loop together with a precondition on the loop variables, and the output is a polynomial $\mathcal{P}$ and a set of ranges $\mathcal{R}$, a range $\mathcal{R}_i$ for each program variable x_i, that define the synthesized invariant:

$$\mathcal{P}(x_1, ..., x_n) \leq 0 \wedge x_1 \in \mathcal{R}_1 \wedge \ldots \wedge x_n \in \mathcal{R}_n \tag{4.2}$$

The key component of our algorithm is the invariant synthesis, which infers the shape of the bounding polynomial and the variable ranges (lines 1-21). The algorithm first synthesizes an invariant assuming a real-valued semantics for the loop body (`withrounding == False`).

The synthesis starts by simulating the loop on a number of random inputs from the precondition, keeping track of all the seen points, i.e. tuples $(x_1, ..., x_n)$. From the obtained points, the algorithm next guesses the shape of a candidate invariant, i.e. a polynomial $\mathcal{P}$ and a set of ranges $\mathcal{R}$ (line 5-7). We check this candidate invariant using an off-the-shelf SMT solver (line 12). If the candidate is not an invariant or is not inductive, the solver returns a counterexample. The algorithm generalizes from the counterexample (lines 16-20) and uses the newly obtained points to refine the candidate invariant. We repeat the process until either an invariant is found, or we reach a maximum number of iterations (empirically, all benchmarks required less than 100 iterations).

After the real-valued invariant is generated, the algorithm checks whether it also holds for the floating-point implementation of the loop (line 29). Should this not be the case, invariant synthesis is repeated taking floating-point rounding errors into account in every refinement iteration. Since rounding errors are usually relatively small, this recomputation is seldom necessary, so that Pine first runs real-valued invariant synthesis for performance reasons.

```
def get_real_invariant(loop, init, withrounding):
  pts = simulate(loop, random.sample(init, $m$), $n$)
  // update pts iteratively
  for i in range(0, max_iters):
    pts = convexHull(pts)
    ranges = round(min(pts), max(pts), $prec_{range}$)
    coefficients = getShape(pts, $prec_{poly}$)
    inv = (coefficients, ranges)

    if withrounding:
      loop = addrounding(loop, ranges)
    cex = checkInvariant(loop, inv)
    if cex is None:
      return inv
    else:
      addCex = getAdditionalCex(loop, inv, cex, $cex\_num$, $d$)
      symPts = getSymmetricPts(cex, inv)
      nearbyPts = getNearbyPts(cex, $d$, inv)
      pts = pts ∪ cex ∪ addCex ∪ symPts ∪ nearbyPts
      pts = simulate(loop, pts, $k$)
  return None

def get_fp_invariant(loop, init):
  inv = get_real_invariant(loop, init, withrounding=False)
  if inv is None:
    return None
  else:
    loopFP = addrounding(loop, inv.ranges)
    cex = checkInvariant(loopFP, inv)
    if cex is None:
      return inv
    else:
      return get_real_invariant(loop, init, withrounding=True)
```

Figure 4.3.: High-level invariant synthesis algorithm (parameters are in cursive)

4.3.1. Simulation

The synthesis starts by simulating the loop execution. For this, PINE samples m values from the variables' input ranges Init uniformly at random, and concretely executes the loop n times for every sample. As a result, we obtain $m \times n$ points, i.e. combinations of variable values, that appear in the concrete semantics of the loop and thus have to be included in an invariant. The sampled points provide a starting point for the invariant search.

4.3.2. Candidate Invariant Conjecture

The invariant we are looking for has two parts: variable ranges $\mathcal{R}$ and a polynomial shape $\mathcal{P}(x)$ enclosing all variable values. To obtain $\mathcal{R}$ and $\mathcal{P}(x)$, Pine first reduces the number of samples by computing the convex hull of the sampled points. We consider invariant shapes that are convex, therefore the values inside the shape can be safely discarded.

The minimum and maximum values of each loop variable x_i in the convex hull vertices determine the range $\mathcal{R}_i$.

Pine infers the shape $\mathcal{P}(x)$ enclosing the convex hull vertices using two optimization methods: minimum volume enclosing ellipsoid (MVEE), and least squares curve fitting. The minimum volume enclosing ellipsoid method computes a bounding ellipsoid such that all points are inside the shape. Pine utilizes a library that computes MVEE by solving the following optimization problem:

$$\begin{gathered}\text{minimize} \log(\det(E)) \\ s.t.(x_i - c)^T E\ (x_i - c) \leq 1\end{gathered}$$

where x_i are the individual points, c is a vector containing the center of the ellipsoid and E contains the information about the ellipsoid shape [130].

While MVEE computes the desired shape, the library that we use supports only two dimensions, and it is furthermore possible that it diverges. To support higher-dimensional loops, or when MVEE fails, we resort to using least squares. With the method of least squares, we find coefficients such that the sum of the squares of the errors w.r.t. to the given points is minimized. For a degree 2 polynomial in variables x and y, Pine transforms the points into the matrix A with entries having the values of $[1, x, y, x^2, xy]$, and a vector b which consists of the values of y^2. By solving the system of equations $Az = b$ for z, we obtain the coefficients of the polynomial. By setting $b = y^2$, we set the last coefficient to 1 in order to avoid the trivial (zero) solution. Least squares computes a tight fit, but will, in general, not include all of the points *inside* the polynomial shape, so that we additionally have to enlarge the 'radius' such that it includes all points. While we do not explore this further in this work, we note that the above sketched least-squares approach also generalizes to fit polynomials of higher degree than 2, using suitable constraints to ensure convex shapes [131].

4.3.3. Reducing the Noise

Both methods used to infer a shape are approximate, i.e. they find a polynomial that is close to the actual shape up to a tolerance bound. Furthermore, they fit a set of points that is incomplete in that it only captures a (random) subset of all of the possible concrete executions. This makes the inferred polynomial shapes inherently noisy and unlikely to be an invariant. We reduce the noise by first normalizing and then rounding the polynomial coefficients to a predefined precision $prec_{poly}$, i.e. to a given (relatively small) number of digits after the decimal point. This effectively discards coefficients (rounds to zero) whose magnitude is significantly smaller than the largest coefficient found. For the remaining coefficients, it removes the—likely noisy—least significant digits.

Similarly, the lower and upper bounds of the computed ranges $\mathcal{R}$ capture only the values seen in simulation and are thus likely to be under-approximating the true ranges. We round the lower and upper bounds outwards to a predefined precision $prec_{range}$, thus including additional values.

The precisions (number of decimal digits) chosen for rounding the polynomial coefficients and the ranges should be high enough to not lead to too large over-approximations, but nonetheless small enough to discard most of the noise. We have empirically observed that the polynomial coefficients should be more precise than the range bounds by one digit, and that $prec_{poly} = 2$ and $prec_{range} = 1$ seems to be a good default choice.

4.3.4. Checking a Candidate Invariant

The obtained polynomial and variables ranges form a candidate invariant, which we check for inductiveness using an off-the-shelf SMT solver by encoding the (standard) constraint $(\mathsf{Init} \rightarrow I(x)) \wedge (I(x) \wedge L \rightarrow I(x'))$, where $I(x) = \mathcal{P}(x) \leq 0 \bigwedge_i x_i \in \mathcal{R}_i$, L is the loop body relating the variables x before the execution of the loop body to the variables x' after.

We translate conditional statements using the SMT command `ite`. Non-deterministic terms receive fresh values from the user-defined range at every loop iteration. Since their ranges do not change we add constraints on the ranges of non-deterministic terms only to I and Init. We encode the above constraint in SMT-LIB using the real-valued theory [129]. The SMT solver evaluates the query and returns a counterexample if it exists. If no counterexample is returned, a candidate invariant is confirmed to be inductive and returned.

4.3.5. Generalizing from Counterexamples

The counterexample returned by the SMT solver is added to the existing set of points that the invariant has to cover. However, this additional point is arbitrary, depending on the internal heuristics of the solver. In order to speed up invariant synthesis, and to avoid biasing the search in a single direction and thus skewing the invariant shape, we generate additional points that also have to be covered by the next invariant candidate. We consider three different generalizations: additional counterexamples, symmetric points and nearby points.

Pine obtains additional counterexamples from the solver by extending the SMT query such that the initial counterexample is blocked and the new counterexample has to be a minimum distance d away from it. Pine will iteratively generate up to *cex_num* additional counterexamples, as long as the solver returns them within a (small) timeout (*cex_num* is a parameter of the algorithm).

Our second generalization strategy leverages the fact that the candidate invariant is an ellipsoid and thus has several axes of symmetry. Pine computes points that are symmetric to the counterexample with respect to all axes of symmetry of the ellipsoid, and adds them as additional points if they satisfy I or Init (i.e. they are also valid counterexamples).

Nearby points are the points that are at a distance d to the counterexample. Pine computes these points in all directions, i.e. $x_i \pm d$, and adds them to the set of points, if they are valid counterexamples. The rationale behind this generalization is that points in the vicinity of a counterexample are often also likely counterexamples. Adding the nearby points allows us to explore an entire area, instead of just a single point.

Pine then performs a second simulation of the loop starting from the newly added set of counterexamples for k iterations. All obtained points are added to the original sampled values and we proceed to synthesize the next candidate invariant.

4.3.6. Floating-Point Invariant

We encode the SMT queries to check the inductiveness of our candidate invariants using the real-valued theory. We note that it is in principle possible to encode the queries using the floating-point theory, and thus to encode the semantics of the loop body, including rounding errors, exactly. However, despite the recent advances in floating-point decision procedures [132], we have observed that their performance is still prohibitively slow for our purpose (CVC4's state-of-the-art floating-point procedure [132] was several orders of magnitude slower than Z3's real-valued procedure [129]).

We thus use a real-valued SMT encoding and soundly over-approximate the rounding errors in the loop body. We compute a worst-case rounding error bound rnd for each expression in the loop body using an off-the-shelf rounding error analysis tool. We use Daisy's dataflow rounding error analysis [14] on the loop body. Daisy computes rounding error bounds for loop-free code, which is sufficient for our purpose, since we only need to verify that $I(x) \wedge L \rightarrow I(x')$, i.e. the executions of the (loop-free) loop body remain within the bounds given by I.

The computed rounding error bound is added to the expression as a non-deterministic noise term bounded by $[-rnd, rnd]$. Note that unlike in existing work [47] that derives one general error bound for all programs assuming a large enough number of arithmetic operations, our rounding error is computed on-demand for each particular candidate invariant. The magnitude of rounding errors depends on ranges of inputs, and so by computing the rounding error only for the invariant's ranges, we are able to add only as little noise as is necessary.

Our algorithm first finds a real-valued invariant and then verifies whether it also holds under floating-point loop semantics. If not, we restart the invariant synthesis and take rounding errors into account for each candidate invariant, recomputing a new tight rounding error in each iteration of our algorithm (line 11). We do not include rounding errors in the first run of the synthesis for better performance, since in practice, we rarely need to recompute the invariant.

Except for the rounding error analysis, our algorithm is agnostic to the finite precision used for the implementation of the loop. By choosing to compute rounding errors w.r.t. different precisions, it thus supports in particular both single and double floating-point precision, but also fixed-point arithmetic of different bit lengths [14], which is particularly relevant for embedded platforms that do not have a floating-point unit.

4.3.7. Implementation

We have implemented the algorithm from Figure 4.3 in the tool Pine as a Python library in roughly 1600 lines of code, relying on the following main libraries and tools: the Qhull library for computing the convex hull[1], a library for computing the minimum volume ellipsoid[2], the least-squares function from scipy (`scipy.linalg.lstsq`), the Python API for the Z3 SMT solver version 4.8.7, and the Daisy tool [14] for computing rounding errors. Simulations of the loop are performed in 64-bit floating-point arithmetic.

4.4. Experimental Evaluation

We evaluate Pine on a set of benchmarks from scientific computing and control theory domains. We aim to answer the following research questions:

RQ1: How does Pine compare with state-of-the-art tools?

RQ2: How quickly does Pine generate invariants?

RQ3: How sensitive is Pine's algorithm to parameter changes?

4.4.1. State-of-the-Art Techniques

We compare the invariants synthesized by Pine to those generated by two state-of-the-art tools: Pilat [48] and SMT-AI [47]. These two tools are the only ones that compute polynomial inequality invariants for floating-point loops without requiring a target condition to be given.

Pilat reduces the generation of invariants of a loop body f to computing the eigenvector ϕ of f that is associated to the eigenvalue 1, i.e. $f(\phi) = \phi$ and ϕ is thus an invariant. Pilat can, in principle, handle non-linear loops by introducing a new variable for each non-linear term and thus effectively linearizing it. This transformation is similar to how we use least-squares to fit a polynomial (subsection 4.3.2). Pilat handles floating-point rounding errors by (manually) including nondeterministic noise for each floating-point operation that captures the rounding error: $(x \circ y) \cdot \delta$, where $\circ \in \{+, -, \times, /\}$ and $|\delta| \leq \epsilon$ is bounded by the machine epsilon. For simplicity, we ignore errors due to subnormal numbers.

[1]`www.qhull.org`

[2]`https://github.com/minillinim/ellipsoid`

SMT-AI [47] and Adje et al. [44] implement policy iteration using the ellipsoid abstract domain. The approach by Adje et al. requires the ellipsoid template to be provided, while SMT-AI generates templates automatically. For our comparison we therefore consider the more general approach of SMT-AI. SMT-AI generates the ellipsoid templates from Lyapunov functions [133], which are functions known from control theory for proving that equilibrium points of dynamical systems are stable. These functions prove that a loop is bounded and thus the shape effectively serves as an invariant. It is known that for linear loops one can generate the polynomial shapes automatically using semi-definite programming. Since such an automated method does not exist for non-linear functions, SMT-AI is limited to linear loops. Semi-definite programming can compute different polynomial shapes, and SMT-AI selects shapes to be tight using a binary search. SMT-AI first computes a real-valued invariant, like Pine, and then verifies that it also satisfies a floating-point loop. Unlike Pine, SMT-AI derives one generic rounding error bound for all (reasonably-sized) loops, and does not recompute the invariant if the floating-point verification fails. We were unfortunately not able to install SMT-AI, so that we perform our comparison on the benchmarks used by SMT-AI, comparing to the (detailed) results reported in the paper [47,134].

Interproc [135] is a static analyzer based on abstract interpretation. It infers numerical invariants using boxes, octagons, linear congruences and convex polyhedra. A user can choose between two libraries that implement these domains: APRON [99] and Parma Polyhedra Library [136]. We tried Interproc on our set of benchmarks, and on 2 benchmarks it produced some bounds for a subset of the program variables. However, the invariants were not convex, and we could not compute their volume. We therefore exclude Interproc from the comparison.

Another potential competitor is an approach by Mine et al. [46] that combines interval and octagon abstract domains with constraint solving. The invariants discovered are effectively ellipsoids, i.e. second-degree polynomial inequalities. However, their approach fundamentally requires target bounds. Since the goal of Pine is to find such tight bounds, and not only prove that they are inductive, we do not compare with Mine et al. [46].

4.4.2. Experimental Setup

Our set of benchmarks contains both linear and non-linear loops and is available open-source[3]. Each benchmark consists of a loop body which iterates an infinite number of times. The linear benchmarks *filter_goubault*, *filter_mine**, *arrow_hurwicz*, *harmonic*, *symplectic* are taken from related work [44,46] and implement linear filters and oscillators. Benchmarks *ex** are taken from the evaluation of SMT-AI [134] and comprise linear controllers, found for instance in embedded systems.

We additionally include the non-linear benchmark *pendulum**, that simulates a simple pendulum and *rotation**, which repeatedly rotates a 2D vector by an (small) angle that is nondeterministically picked in each iteration. Both benchmarks use the sine function,

[3]`https://github.com/izycheva/pine/tree/master/benchmarks`

which we approximate using a Taylor approximation. The *nonlin_example** are non-linear dynamical systems collected from textbook examples on Lyapunov functions.

Three of our benchmarks contain operations on nondeterministic noise terms. Most benchmarks are 2-dimensional, except for *ex4**, which has 3 variables, *ex2** and *ex5**, which have 4 variables, and *ex6** that has 5 variables.

We run our evaluation on a MacBook Pro with an 3.1 GHz Intel Core i5 CPU, 16 GB RAM, and macOS Catalina 10.15.3.

4.4.3. RQ1: Comparison with State-of-the-Art

Each tool generates an invariant with an elliptic shape, and Pine and SMT-AI provide additionally ranges for variables. We compare the inductive invariants generated by each tool based on their volume. The volume of an invariant is given by the set of points satisfying $\mathcal{P}(x) \leq 0 \wedge \bigwedge_i x_i \in \mathcal{R}_i$, where the variable ranges may intersect with the ellipsoid. We compute this intersection (approximately) using a Monte-Carlo simulation with $3 \cdot 10^6$ samples, by comparing how many samples are within the invariant to how many are inside the variable ranges (for the latter we know the volume exactly). Our volume estimates are accurate to two decimal digits.

We run Pine with a default set of parameters, that we determined empirically (see subsection 4.4.5). In order to compare with other tools that only support single floating-point precision, Pine computes rounding errors (and invariants) for 32-bit floats.

Columns 2-4 in Table 4.1 show the volumes of the invariants generated by SMT-AI, Pilat, and Pine. '-' denotes the cases where a tool did not generate an invariant. Benchmarks for which we did not have data for SMT-AI are marked as 'undef'. 'PF' denotes cases where an invariant was generated, but it did not satisfy the given precondition. 'TO' marks cases when a tool took longer than 20 minutes to generate an invariant. Here, smaller volume is better, the best volumes are marked bold.

Due to the inherent randomness in its algorithm, we run Pine 4 times and compute the average volume and running time across the runs. The last column shows variations in volume with respect to the average (i.e. (max - min)/average).

We observe that Pine produces the tightest invariants on 17/24 (70%) of the linear benchmarks. Additionally, Pine generates invariants for all non-linear benchmarks in our set, whereas Pilat was not able to generate invariants for any of them. Pine produces invariants that are in the best case on average 20x tighter than the ones by SMT-AI, and 2.7x tighter than the ones by Pilat (compared on the 6 benchmarks, for which it was able to generate an invariant). In the worst case (observed over our 4 runs), the factors decrease to 13.8x and 1.8x respectively. Only for the benchmarks *ex6-butterworth* and *filter-mine2-nondet*, the worst-case volumes computed by Pine become 1.9x and 1.6x larger than the ones computed by SMT-AI and Pilat, respectively, and are thus still of the same order of magnitude.

RQ1 Conclusion: Based on our experimental results, we conclude that Pine generates much tighter invariants than its competitors. Moreover, unlike other state-of-the-art tools, Pine is applicable to both linear and non-linear loops.

	Benchmark	SMT-AI	Pilat	Pine	Pine avg time, s	Volume variation
Non-linear	pendulum-approx	undef	-	**12.92**	21.09	30.03%
	rot.nondet-small	undef	-	**5.97**	30.13	16.25%
	rot.nondet-large	undef	-	**6.67**	33.78	10.87%
	nonlin-ex1	undef	-	**0.23**	14.43	18.51%
	nonlin-ex2	undef	-	**0.56**	7.32	5.23%
	nonlin-ex3	undef	-	**7.07**	12.45	3.35%
Linear	arrow-hurwitz	undef	-	**4.40**	4.75	7.00%
	harmonic	undef	18.41	**3.52**	10.81	9.70%
	symplectic	undef	PF	**2.32**	7.71	12.11%
	filter-goubault	undef	PF	**1.84**	4.94	1.31%
	filter-mine1	undef	PF	**6.32**	7.18	1.58%
	filter-mine2	undef	1.16	**0.49**	4.48	71.92%
	filter-mine2-nondet	undef	4.92	**4.45**	10.70	66.38%
	pendulum-small	undef	12.53	**9.10**	7.11	7.51%
	ex1- filter	**475.06**	498.37	-	43.61	-
	ex1-reset-filter	**475.98**	-	-	45.95	-
	ex2-2order	17.37	**1.07**	4.92	7.45	46.73%
	ex2-reset-2order	17.36	-	**3.08**	6.28	6.65%
	ex3-leadlag	-	-	-	46.68	-
	ex3-reset-leadlag	-	-	-	44.56	-
	ex4-gaussian	0.61	-	**0.22**	16.93	46.16%
	ex4-reset-gaussian	17.05	-	**1.45**	23.10	137.47%
	ex5-coupled-mass	5,538.47	TO	**100.61**	8.63	9.48%
	ex5-reset-coupled-mass	5,538.34	-	**81.02**	8.44	27.54%
	ex6-butterworth	65.25	-	**25.43**	16.34	272.89%
	ex6-reset-butterworth	700.06	-	**10.30**	219.34	0.00%
	ex7-dampened	**12.17**	-	18.68	19.96	15.71%
	ex7-reset-dampened	**12.17**	-	-	39.70	-
	ex8-harmonic	5.75	-	**2.32**	6.99	9.77%
	ex8-reset-harmonic	5.75	-	**2.85**	7.15	28.08%
	ex5+6	**6,927.12**	TO	TO	TO	-

Table 4.1.: Volumes of invariants generated by Pine, Pilat and SMT-AI, Pine's average running time and variation in invariant volumes across 4 runs

4.4.4. RQ2: Pine's Efficiency

Pine generates invariants in on average 25, and at most 220 seconds; the largest running time is also the benchmark with the largest number of variables. Pine was able to confirm the real-valued invariant also for the floating-point semantics for all but two *rotation** benchmarks, for which it had to recompute the invariants two out of four times. We consider the running times to be acceptably low such that it is feasible to re-run Pine several times for an input loop, in order to obtain a smaller invariant, if needed.

RQ2 Conclusion: Pine finds and confirms invariants quickly, therefore can be applied multiple times until a desirably small invariant is generated. Its running time depends on the benchmark's complexity; even on larger benchmarks Pine finishes in under a few minutes.

m	n	cex_num	d	k	symPts	nearbyPts	volume
100	1000	0	0.5	500	✓		2.283
100	1000	0	0.5	100		✓	2.297
100	10000	5	0.25	100	✓		2.311
100	1000	2	0.25	100		✓	2.314
100	10000	1	0.5	500		✓	2.335

Table 4.2.: Top-5 minimum volume configurations

4.4.5. RQ3: Parameter Sensitivity

We now evaluate the influence of different parameter settings on the performance of our proposed algorithm in terms of its ability to find tight inductive invariants. For this, we explored the parameter space of our algorithm on 13 of our benchmarks that include (non-)linear infinite loops without branching. We evaluate the different combinations of varying the following parameters:

- whether or not symmetric points are used
- whether or not nearby points are used
- number of random inputs and loop iterations for initial simulation (algorithm parameter m-n): 100-1k, 1k-1k, 100-10k
- number of loop iterations for counterexamples simulation (k): 0, 100, 500
- number of additional counterexamples (cex_num): 0, 1, 2, 5 (when $cex\_num = 0$, no additional counterexample is generated)
- distance to nearby points (in % of the range) (d): 10%, 25%, 50%
- three different precisions for rounding: $(prec_{poly} = 1, prec_{range} = 0)$, $(prec_{poly} = 2, prec_{range} = 1)$, $(prec_{poly} = 3, prec_{range} = 2)$, where $prec_{poly}, prec_{range}$ give the number of decimal digits for the polynomial coefficients and the variable ranges, respectively.

In total, we obtain 1296 configurations. We run Pine with each of them once.

Default Configuration 185 parameter configurations were successful on all of the 13 benchmarks. From these, we select the configuration that generates invariants with the smallest average volume across the benchmarks as our default configuration: $prec_{poly} = 2, prec_{range} = 1, m = 100, n = 1000, k = 500$. To generalize from counterexamples the default configuration uses only symmetric points.

Table 4.2 shows the 5 best configurations, according to average volume (we normalized the volume across benchmarks). We note that the differences between volumes for successful configurations are small, so that we could have chosen any of these configurations as the default.

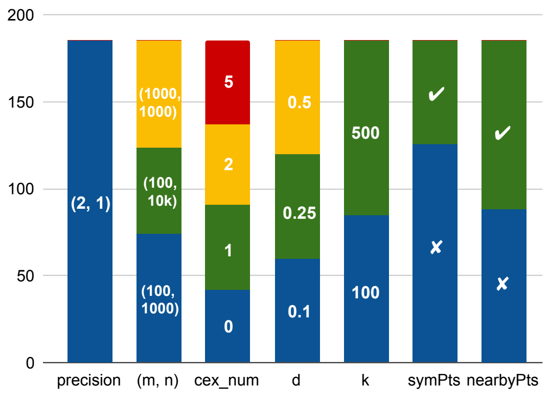


Figure 4.4.: Proportion of parameters appearing in successful configurations

Benchmark	min	avg	max
pendulum-approx	9.88	12.03	20.21
rot.nondet-small	4.76	5.78	8.07
rot.nondet-large	6.27	6.92	11.43
nonlin-ex1	0.06	0.20	0.83
nonlin-ex2	0.55	0.56	0.59
nonlin-ex3	6.84	7.04	7.27
harmonic	3.28	3.84	4.46
symplectic	2.12	2.21	2.68
filter-goubault	1.82	1.83	1.85
filter-mine1	6.29	6.40	8.72
filter-mine2	0.28	1.03	3.83
filter-mine2-nondet	1.74	2.44	11.37
pendulum-small	8.61	10.14	24.55

Figure 4.5.: Volumes of invariants with successful configurations

Successful Configurations We study the 185 successful configurations to see which parameter values appear the most frequently, and thus seem most successful in finding invariants. Figure 4.4 shows the distribution of the different parameters in the set of successful configurations. For instance, nearby points are included in the generalization in roughly half of the configurations, while the precisions $(prec_{poly} = 1, prec_{range} = 0)$ and $(prec_{poly} = 3, prec_{range} = 2)$ do not appear at all in the successful configurations, i.e. only $(prec_{poly} = 2, prec_{range} = 1)$ was able to find invariants for all benchmarks.

From Figure 4.4, we conclude that simulating the loop starting from counterexamples (line 21 in Figure 4.3) is crucial in finding an invariant - none of the configurations without this additional simulation worked on all benchmarks. On the other hand, whether this simulation runs 100 or 500 loop iterations seems to make less of a difference.

For the remaining parameters, we do not observe a strong significance; they are roughly equally distributed among the successful configurations. From this we conclude that our algorithm is not sensitive to particular parameter settings, and will find invariants successfully for many different parameter configurations.

The choice of parameters does, however, influence the size of the invariants generated, at least for certain benchmarks. Figure 4.5 shows the minimum, maximum and average volumes for each benchmark across successful configurations. While for some benchmarks, the variation is small, for others the best configuration produces invariants that are half the size from the worst one.

Across the 1296 configurations, we observe that if a real-valued invariant is found, it is also confirmed in 89% of cases, and thus has to be re-computed in only 11% of cases. The only outlier that needs recomputation more than once is *rotation-nondet-large*, which rotates a vector by a larger angle, and therefore is understandably more sensitive to enlarging the coordinates with some noise.

Last but not least, we changed the input precision parameter to 16-bit fixed points to assess whether Pine can be applied to other finite precisions. We ran Pine with its default configuration on all our 30 benchmarks (including *ex**). The smaller bit length results in larger rounding errors, so that PINE had to recompute an invariant for 5 additional benchmarks (i.e. where the real-valued invariant was not confirmed), but was able to find an inductive invariant for as many loops as with floating-point implementation.

RQ3 Conclusion: PINE's success in finding and confirming finite-precision inductive invariants is independent from a particular parameter configuration. The essential point is to use *simulation* and *some generalization* from counterexamples.

4.5. Related Work

Invariants with Abstract Interpretation Many tools and libraries [99] infer invariants over program variables using abstract interpretation. The abstract domains range from efficient and imprecise intervals [137], over octagons [138], to more expensive and expressive polyhedra [136, 139]. For programs with elliptic invariants most linear abstract domains are insufficient to express an invariant [47].

Ellipsoid domains have been defined to work for specific types of programs, e.g. digital filters [140] and programs where variables grow linearly with respect to the enclosing loop counters [141]. Performing abstract interpretation using policy iterations instead of widening allows the use of the ellipsoid abstract domain more generally [44, 45]. This approach requires templates of the ellipsoids to be given, however. Recent works [47, 48] are able to discover ellipsoid inductive loop invariants without the need for templates, but being based on semidefinite programming and linear algebra, respectively, are fundamentally limited to linear loops only. Alternatively, Bagnara et.al. [142] have explored an abstract domain that approximates polynomial inequalities by convex polyhedra and leverages the operations, including widening, of polyhedra. Sankaranarayanan et.al. [143] show how to generate polynomial equality invariants by reducing the problem to a constraint satisfaction problem.

Invariants for Integer Loops Our algorithm builds on several ideas that have been explored in loop invariant synthesis previously, including the use of concrete executions to derive polynomial templates and counterexample-based refinement. Floating-point loops and in particular the uncertainties introduced due to rounding errors pose unique challenges that existing techniques cannot handle, as we discuss next.

Several works have explored the use of machine-learning in teacher-learner frameworks [43, 128]: the learner guesses a candidate invariant from a set of examples, and the teacher checks whether the invariant is inductive. If it is not, the teacher provides feedback to the learner in form of additional (counter)examples. These approaches rely on a target property to be given (to provide negative examples) and are thus not immediately applicable to synthesizing floating-point inequality invariants. The framework C2I [127] employs a learner-teacher framework, but where the learner uses a randomized search to generate candidate invariants. While surprisingly effective, the approach is, however, limited to a fixed search, e.g. linear inequalities with a finite set of given constants as coefficients. Sharma et.al. [126] present a learning-based algorithm to generate invariants that are arbitrary boolean combinations of polynomial inequalities, but require a set of good and bad states and thus an assertion to be given.

The tool InvGen [144] generates integer linear invariants from linear templates, using concrete program executions to derive constraints on the template parameters. The tool NumInv [145] and the Guess-And-Check algorithm [146] generate polynomial equality invariants using a similar approach. For integer programs and in particular equality constraints, this approach is exact. In our setting with floating-point programs and inequalities such constraints cannot be solved exactly and thus require a different, approximate, approach. NumInv and Guess-And-Check furthermore employ counterexamples returned by the solver for refinement of the invariant. These counterexamples are program inputs, however, due to the complexity of the floating-point or real-arithmetic decision procedures, this technique does not scale to our target numerical programs. We are thus restricted to counterexamples to the invariant property.

Abductive inference in the tool Hola [41] and enumerative synthesis in FreqHorn [104, 105] are two further techniques that have been used to generate invariants for numerical programs, but are unfortunately not applicable to generate the invariants we are looking for. Hola relies on quantifier elimination which solvers do not support (well) for floating-points and reals; FreqHorn generates the invariant grammar from the program's source code, but for our invariants their components do not explicitly appear in the program itself.

Chakraborty et al. [147] introduced an induction technique for finding an invariant for full programs without explicitly generating invariants for each loop. This technique performs non-trivial transformations that potentially reorder computations and is therefore not directly applicable to floating-point code where the order of computation affects the rounding errors.

Allamigeon et. al. [148] extend ellipsoidal analyses to generate disjunctive and non-convex invariants for switched linear systems. We do not consider disjunctive invariants in this work and leave their exploration to future work.

Recurrence-based techniques [149, 150] generate loop invariants that exactly capture the behavior of a numerical integer loop. While these techniques work for arbitrary conditional branches, imperative code and nested loops, they generate invariants of a different form, i.e. in general not polynomial inequalities and are thus orthogonal to our approach.

Preconditions Besides inductive loop invariants, one can also infer other interesting properties for programs, for instance, floating-point preconditions [151] to reason about valid inputs to a function. Most of the work applicable to numerical loops, however, focuses on integer preconditions [152, 153] and does not take into account uncertainties introduced by floats; or uses sampling and therefore does not provide sound guarantees [154, 155]. Precondition inference is not directly comparable to our work as it reasons about a program in a backward way and therefore usually requires a target end-state to be given by users.

4.6. Conclusion and Future Work

We presented a novel algorithm for synthesizing polynomial inequality invariants for finite-precision loops. For this, we show how to extend the well-know technique of counterexample-guided invariant synthesis to handle the uncertainties arising from finite-precision arithmetic. The key insight to make our iterative refinement work is that a single counterexample is not sufficient and the algorithm has to explore the space of counterexamples more evenly in order to successfully generalize. While the resulting algorithm is heuristic in nature, it proved to be remarkably effective on existing benchmarks as well as on handling benchmarks out of reach of existing tools.

Alternative Shapes of Invariants One possible extension of our synthesis algorithm is to include other shapes of invariants beyond ellipsoids. The hard constraint is only that it must be possible to encode in an SMT query. In this chapter, we focused on convex shapes but given a suitable curve fitting function, in principle, it is also possible to generate non-convex invariants.

Another possibility would be to allow several disconnected shapes in the invariant, for instance, a disjunction of two separate ellipsoids (in addition to variables' ranges). The candidate invariant could be proposed, for instance, by first applying a clustering algorithm to the sampled points and then fitting the individual shapes. However, one has to carefully design SMT queries that check the invariants for inductiveness, as queries containing a disjunction operator are potentially more expensive to evaluate, which may affect PINE's performance.

Bounded Loops In this chapter, we focused on unbounded numerically stable loops. Though our input program grammar allows specifying bounded loops, we currently do not provide an explicit support for them. Using the grammar defined in section 4.2 bounded loops could be encoded as:

```
if (i <= c) {
        x_i := expression
        i := i+1
} else {
        x_i:= x_i
        i := i
}

x_i\in [lo, up]
i\in [0.0, 0.0]
```

where the if-branch describes the loop body and the else-branch denotes the stable state, where the values no longer change after the loop counter `i` has reached its bound c.

The current implementation of Pine treats the counter `i` as if it was another real-valued variable. The solver is unaware that the counter cannot take arbitrary values in the range $[0, c]$, but can only be one of the values in $\{0, 1, ..., c\}$ and therefore finds fractional counterexamples that cannot actually occur in the program.

To mitigate this issue we could introduce explicit support for bounded loops and instrument our SMT queries to include additional information about counter variables.

Use the Invariants Another meaningful direction of future work is to use the synthesized invariants to compute rounding errors for loops. For instance, we could use the closed-form equation from Rosa [13] to derive rounding error after k iterations of the loop:

$$\left| f^k(x) - \tilde{f}^k(\tilde{x}) \right| \leq K^k \lambda ((I - K)^{-1}(I - K^k))\sigma$$

Here, K is the matrix of propagation coefficients, I is the identity matrix, λ is the vector of initial errors and σ is the vector of rounding errors committed in one iteration. To compute the matrix K and the vector σ Rosa's method requires a real-valued invariant for the variables in the loop and a finite-precision invariant, which we could provide with Pine.

Part II.

Optimization of Numerical Kernels

5. Optimizing Kernels with Elementary Functions

Many numerical applications can tolerate a certain amount of noise and still compute useful results [156]. When the results do not need to be computed exactly, one may intentionally introduce noise in the implementation thus trading off a controlled portion of accuracy for performance (or another optimization objective). In this chapter, we explore this trade-off through approximations.

Approximations can have different flavors. When using reals [157] is prohibitively expensive, fixed-point- and IEEE-754 floating-point numbers [64] provide a convenient and practical approximation: finite precision enables an efficient execution, at the expense of rounding errors and thus reduced accuracy. This kind of approximation is leveraged, for instance, by mixed-precision tuning.

A more aggressive flavor of approximations is replacing a (sub-)expression with a syntactically different expression that is easier or faster to evaluate. A prominent example of such approximations are polynomials of a variable degree, Taylor series [158], Chebyshev polynomials [159], etc. Polynomials are particularly useful when approximating *elementary functions* (trigonometric functions, exponent, logarithm) that by definition can be represented as a finite sequence of operations and functions [160]. By varying the degree and the coefficients of the polynomials, one can influence how closely they model elementary functions.

Finding appropriate approximations that improve performance is challenging. The space of possible approximations is huge: every function can be approximated with multiple polynomials, and each of these polynomials can be implemented in different finite precision, affecting the magnitude of the introduced error. Additionally, to be able to provide guarantees on the overall error of the optimized program, we need to verify the program together with all proposed approximations before we can decide whether the approximations are suitable.

In this chapter, we present the first sound optimization method that automatically synthesizes *efficient* numerical kernels with *approximate elementary functions* that have guaranteed accuracy. The user specifies an ideal, real-valued program together with a maximum error bound. Our technique generates an efficient finite-precision implementation with polynomial approximations of elementary functions.

The numerical kernels we handle cover widely used applications in various domains, to name a few: embedded control to compute rotations of robotic components, scientific computing simulations to determine the state of a periodic event, and machine learning models with sigmoid activation functions. By default, programmers implement the elementary functions in these kernels using library implementations. While these are convenient and optimized, they usually provide only a limited set of accuracies, for example, only for 32 and 64-bit precisions, which may lead to using unnecessarily accurate versions of the function call. Moreover, library functions are designed to produce accurate results on all valid inputs, many of which may never occur in the program.

When our algorithm replaces a function call with a (piecewise-) polynomial approximation it has a fine-grained control over the resulting accuracy. First, our polynomial approximations are tailored to ranges of elementary function arguments occurring in the program. This allows the optimization to use polynomials of (relatively) low degrees and find fast implementations. Secondly, when implementing the polynomials our algorithm can choose a suitably low or high precision to match the desired accuracy.

Our algorithm is not specific to one type of finite precision, however, the potential of approximations can be better explored in combination with fixed points. Fixed-point implementations allow arbitrary bit-widths for individual arithmetic operations and provide more options in the accuracy/performance tradeoff. On the other hand, higher flexibility means a significantly larger search space compared to floating-point precisions, which makes finding a solution to the optimization problem more challenging.

Our algorithm distributes the maximum allowed error specified by the user among different sources of errors—rounding and approximation—and generates an efficient and sound finite-precision implementation with polynomial approximations. The key observation behind our algorithm is that errors of a different nature behave similarly: the approximation and rounding errors will accumulate and propagate together. This observation allows us to repurpose a finite precision assignment procedure as an error budget distribution algorithm. Given this distribution, we then generate approximations with a combination of existing polynomial approximation technique [161] and rounding error analysis [14].

We implemented our algorithm inside the tool Daisy and evaluate it on several embedded, scientific computing and machine learning kernels. Compared to implementations using default library functions, our synthesized programs take on average **2.23x** less machine cycles to execute on FPGAs.

Contributions To summarize, this chapter describes the following contributions:

- the first sound performance optimization that replaces elementary function calls with fixed-point polynomial approximations,
- an experimental evaluation using 13 benchmarks from various application domains,
- a prototype implementation of the synthesis algorithm, which we released as open-source: `https://github.com/malyzajko/daisy/tree/approx`.

5.1. Background

Before describing our algorithm, we briefly review important details about elementary functions and polynomial approximations. For information on fixed-point arithmetic and rounding error analysis we refer the reader to chapter 2.

Elementary Functions Elementary functions are defined as mathematical functions that can be represented by a finite composition of constants, arithmetic operations, algebraic, exponential and logarithmic functions [160]. They include all trigonometric functions, exponent and logarithm, and are implemented by standard mathematical libraries for both floating-point and fixed-point arithmetic. Here, we focus on the implementation provided and used by Xilinx Vivado compiler [59], which we use in our experiments and which is widely used for compiling code for accelerators. We note, however, that our approach is not tied to a particular choice of fixed-point compiler.

Xilinx Vivado supports 32-bit fixed-point implementations for sine, cosine and logarithm, and 8 or 16-bit versions of the exponential function. The compiler further supports automated conversion to floating-point arithmetic, such that floating-point implementations of elementary function calls can be used within fixed-point arithmetic programs. These are provided for precisions 16, 32, and 64 bit. Thus, while some support for elementary functions is provided, it is only available for a small variety of precisions, effectively limiting optimization options.

Polynomial Approximation in Metalibm

Polynomials are a common choice for approximating complex functions. The approximation accuracy largely depends on the degree of the polynomial, larger degrees being more accurate, but incurring a higher execution cost. State-of-the-art tool Metalibm [161] finds a polynomial of a suitable degree fully automatically. To generate the approximation, Metalibm requires a specification that consists of an elementary function, an input domain and a target error which the approximation has to satisfy. It employs Remez' algorithm [162], which guarantees the best possible polynomial approximation. It additionally performs domain splitting [163], which allows different polynomials and degrees to be used on different parts of the input domain, and supports a number of further features such as generation of tables for table lookup and range reduction.

Metalibm currently generates double floating-point C implementations which can outperform highly optimized library implementations [164]. It can be applied to individual or compound elementary functions as long as they are univariate (Remez' algorithm only supports univariate functions), though it usually times out after several hours on more complex compound functions. To summarize, Metalibm generates efficient individual floating-point approximations, but cannot be applied to entire programs and it does not support fixed-point arithmetic.

```
def xu1(x1: Real, x2: Real): Real = {
  require(0.01 <= x1 && x1 <= 0.75 && 0.01 <= x2 && x2 <= 1.5)
   2 * sin(x1) + 0.8 * cos(2 * x1) + 7 * sin(x2) - x1
} ensuring(res => res +/- 4.24e-06)
```

Figure 5.1.: Example input program with elementary function calls

5.2. Our Optimization Algorithm

We illustrate the optimization process with an example program `xu1` shown in Figure 5.1, which is taken from the benchmark set of the CORPIN project [165]. The input specification for the optimization consists of: an ideal real-valued algorithm with three elementary function calls `sin(x1)`, `cos(2*x1)` and `sin(x2)`, input ranges for variables in the `require` clause, and the maximum tolerated absolute error for the program in the `ensuring` clause: 4.24e-6.

Given this specification, our goal is to automatically find an optimized program which approximates the expensive elementary function calls and implements the arithmetic operations in a suitable fixed-point precision, while respecting the specified error bound.

The specified maximum tolerated error for the program can be seen as a budget, which has to be distributed between all the different sources of errors in the program, namely the elementary function approximations as well as the finite-precision arithmetic. Note that the approximation polynomials themselves have to be implemented in fixed-point arithmetic as well. In our example we thus need to assign a roundoff error budget to the four multiplications, two additions and one subtraction of the top-level program, as well as to the yet unknown polynomial approximations of `sin(x1)`, `cos(2*x1)` and `sin(x2)`.

Thus, in order to synthesize an approximate program which satisfies the specified error bound, we need to:

1. distribute the error budget, specified for the whole program, between arithmetic operations in the top-level function, potentially multiple elementary function calls, and the finite-precision implementation of the polynomials,

2. find a (piecewise-) polynomial approximation for every elementary function which stays within limits of the assigned approximation error budget, and

3. assign a finite precision to each arithmetic operation of the top-level function, as well as the polynomial approximations.

Each of the above challenges involves finding a solution in a large search space, and the search is furthermore complicated by the fact that individual errors interact in non-linear and discrete ways. Every error introduced at one point in the program gets propagated through the remaining part of the computation, in the course of which it may be magnified, or diminished.

Because we are explicitly aiming to synthesize a more efficient program, we furthermore have to keep in mind the accuracy-efficiency trade off. If we assign a significant portion of the error budget to elementary function calls, we might need to use a higher, and thus more expensive, precision for the rest of the operations in order to satisfy the error budget for the whole program. Thus, performance gained by approximation might be negated by the need for high finite precision. This is a multiple-objective optimization task, which is known to be difficult in general.

Previous work provides only partial solutions to some of these challenges, which only exist in isolation. While Metalibm generates polynomial solutions with guaranteed bounds, it requires the user to provide range bounds and target errors *at the call site* and thus does not consider the full program and error propagation. Additionally, Metalibm only generates double-precision floating-point implementations. State-of-the-art rounding error analyzers can assign uniform or mixed fixed-point precisions to arithmetic computations and function calls but do not consider their approximations.

5.2.1. High-level Algorithm

In this chapter, we provide a complete solution for the above-mentioned challenges and propose a push-button optimization that takes into account the interactions between different errors and synthesizes efficient numerical kernels, which are guaranteed to be accurate up to a specified total error bound. [1]

We distinguish two error budgets. The *global* budget covers errors of elementary function calls and roundoffs of arithmetic operations in the original program. The *local* budget covers the approximation error of individual elementary function calls and roundoff errors introduced by their polynomial approximations.

Figure 5.2 shows our high-level algorithm. The algorithm operates top-down. It first distributes the global error budget (subsection 5.2.2), which assigns local error budgets to individual elementary function calls. The local budget is distributed itself in a feedback loop between approximation and implementation errors (subsection 5.2.3). The approximation error, as well as other information obtained using static analysis is used to call Metalibm to generate polynomial approximations (subsection 5.2.4). Finally, the implementation error budget is used to assign fixed-point precisions to the approximation polynomials (subsection 5.2.5). We discuss alternatives to this top-down approach in subsection 5.2.6.

5.2.2. Distributing the Global Error Budget

Given the global error budget ϵ_g we first distribute it to local budgets for each arithmetic operation, variable and elementary function call, taking into account error propagation. Our *key observation* for this distribution is that the accuracy of the elementary function calls is unlikely to be very different from the other arithmetic operations, otherwise, the

[1] We optimize for running time, but our algorithm is also applicable to other objectives such as energy, with an appropriate cost function. We note that running time often correlates with energy.

```
Input: $\mathbb{S}$ - all variables, arithmetic operations and elementary function calls;
$s_{ef}$ - elementary function calls; $\epsilon_g$ - global error budget

  1. $\forall s \in \mathbb{S}$ assign precision $p_s$ wrt. cost of $s$ and $\epsilon_g$

  2. Based on $p_s$ assign local budget $\epsilon_i$ to all $s_{ef}$

  3. $\forall s_{ef}$ and $k.0 \leq k \leq 5$ REPEAT:
       • Split $\epsilon_i$ into $\epsilon_{i\_approx}$ and $\epsilon_{i\_fp}$, for $k = 0$ $\epsilon_{i\_approx} = \epsilon_{i\_fp}$, $k \geq 1$: $\epsilon_{i\_fp} =$
         $\epsilon_{i\_fp} \circ \delta$, where $\delta = \epsilon_i / 2^{k+1}, \circ \in \{+, -\}$
       • Call Metalibm to generate a polynomial approximation wrt. $\epsilon_{i\_approx}$
       • Generate a finite-precision implementation, such that $e_{fp} \leq \epsilon_{i\_fp}$
       • Compute cost $c_k$ of the obtained finite-precision implementation $i_k$
       • Consider following cases:
           – $c_k > c_{k-1}$: if $k = 1$ choose the opposite $\circ \in \{+, -\}$, else RETURN $i_{k-1}$
           – $c_k = c_{k-1}$: if $k = 1$ REPEAT, else RETURN $i_k$
           – $c_k < c_{k-1}$: if $k < 5$ REPEAT, else RETURN $i_k$
```

Figure 5.2.: High-level synthesis algorithm

errors they introduce would dominate the overall error. Based on this observation, we *treat the approximation errors introduced by elementary functions as a kind of roundoff error* of a given finite precision. With this assumption, we can leverage a precision assignment algorithm to distribute the global error budget.

In particular, we use the two assignment strategies implemented in the tool Daisy, which provide a uniform- or mixed-precision assignment. They assign a fixed-point precision to every arithmetic operation and elementary function call. For elementary functions, we interpret the associated roundoff error with this fixed-point format as the local error budget.

Daisy's uniform precision assignment performs a linear search and selects the smallest uniform precision which satisfies the provided overall error bound. Mixed-precision tuning is more involved, as it introduces cast operations which incur a certain cost. Unlike uniform precision assignment, mixed-precision tuning thus requires a cost function to choose between efficient programs. However, at this point, we do not know the actual implementation of the elementary function approximations. Furthermore, the performance of fixed-precision implementations on an accelerator depends on the compilation algorithm, which is a highly complex, and generally unknown function (e.g. the commercial Xilinx Vivado compiler). Thus the cost function has to *estimate* the cost of elementary function calls and arithmetic operations as well as possible.

We extend Daisy's mixed-precision tuning to be parametric in the cost function, which allows us to explore different options. We consider three cost functions:

1) an area-based [166] one used by Daisy previously, that predicts the area size of a chip that would be able to execute the program,

2) one obtained with machine learning, and

3) an equally weighted combination of 1) and 2). The implementations of all cost functions are available under `https://github.com/malyzajko/daisy/blob/approx/src/main/scala/daisy/opt/CostFunctions.scala`.

For 2), we learned a multi-layer perceptron regressor [167] from random precision assignments on a set of benchmarks, for which we obtained actual performance data by compiling them to an FPGA with Xilinx Vivado. We furthermore extended both the area-based and the machine-learned cost function so that elementary function calls incur twice the cost of arithmetic operations. The factor 2 has been found empirically; it confirms our intuition that the error introduced by the elementary function call is comparable to errors of arithmetic operations.

We have empirically determined that the weighted combination (i.e. option 3) works best in general. We have also observed that whether uniform or mixed precision is best is highly application specific. Thus, our algorithm tries both a uniform and a mixed-precision assignment with a weighted cost function and returns a better result. For our running example, uniform precision assignment performs best overall and assigns precision `Fixed(26)` to `sin(x1)`, `cos(2*x1)` and `sin(x2)`. From this, we obtain local error budgets $\epsilon_0 = \epsilon_1 = \epsilon_2 = 5.96\text{e-}8$.

5.2.3. Distributing the Local Error Budget

Once a local error budget ϵ_i is assigned to each individual elementary function call, we have to decide how much of ϵ_i will be spent on the approximation ϵ_{i_approx} and how much on the finite-precision implementation of the approximation polynomial ϵ_{i_fp}.

To find an optimal split between the two local budgets we use a refinement loop guided by the cost function. We start with an equal split, i.e. $\epsilon_{i\_approx} = \epsilon_{i\_fp} = 0.5\epsilon_i$, synthesize a polynomial approximation respecting ϵ_{i_approx} and assign finite precision such that ϵ_{i_fp} is satisfied (see sections below). We then estimate a cost c_0 of the obtained implementation using a cost function.

Then, our algorithm increases ϵ_{i_fp} by $\delta = \epsilon_i/2^{k+1}$, where k is the number of steps taken in one direction, and decreases ϵ_{i_approx} respectively. We repeat synthesis of a polynomial and finite-precision assignment for the new values of ϵ_{i_fp} and ϵ_{i_approx} and compute the updated cost c_k. The obtained cost c_k is used to determine the fitness of the local error budget distribution. We accept an implementation found at the step $k-1$ if $c_k > c_{k-1} \wedge k > 1$. In case the cost increases at the very first step, we change the direction of the search, i.e. decrease ϵ_{i_fp}, reset k to 0 and repeat the refinement. If the cost has not changed $c_k = c_{k-1}$ at the beginning of the search ($k = 1$), we make one more refinement iteration, for $k > 1$ the k-th implementation is accepted. If after the k-th step we have $c_k < c_{k-1}$, this indicates that the performance of the implementation at the step k has improved. We then repeat the refinement until the $(k-1)$-step implementation has been accepted. To ensure termination we set the maximum number of steps to $k = 5$.

The quality of refinement depends on how accurately a cost function reflects the actual compiler behavior, i.e. how well it can predict the circuit that will be implemented. Our approach is parametric in the cost function, which allows flexibility in optimization for different objectives and hardware. Similarly to global budget distribution with mixed-precision tuning, we evaluated an area-based, machine-learned and a combined cost function, and found that an equally weighted combination of the area-based and machine-learned cost function had best performance overall.

For our running example, the refinement loop needed two iterations for `sin(x1)`, meaning that the optimal distribution found was $\epsilon_{0\_fp} = 3\epsilon_{0\_approx}$. The corresponding values are: ϵ_{0_fp} =4.47e-8 and ϵ_{0_approx}= 1.49e-8. For `cos(2*x1)` and `sin(x2)` the initial equal split already had a minimum cost, i.e. $\epsilon_{i\_fp} = \epsilon_{i\_approx} = 2.98\text{e-}8$ for $i \in \{1, 2\}$.

5.2.4. Synthesizing the Approximation Polynomial

For finding a polynomial approximation of each individual elementary function we leverage the tool Metalibm. To generate an approximation, we need to specify the folllowing parameters: a) the elementary function $f(x)$ to be approximated, b) the domain $x \in I$, on which $f(x)$ will be approximated, c) the assigned local approximation error budget ϵ_{i_approx}, and d) the maximum polynomial degree. Note that domain I is not the input domain specified by the user, but the local input domain of the function's parameter x. This domain should be computed as tightly as possible, as this may allow Metalibm to use polynomials of smaller degree or less internal domain subdivisions. In general, determining these domains is challenging to do manually. Our algorithm uses static analysis of ranges and finite-precision errors using interval and affine arithmetic to compute this information fully automatically. Whenever a program contains the same elementary function call several times, we check whether we have already synthesized an approximation for a given range and assigned local error budget ϵ_i. In this case, we reuse already generated approximation.

We have empirically found a suitable value for the maximum polynomial degree to be 7. This limit influences how Metalibm refines the search for a suitable polynomial. When an approximation found at some intermediate step is insufficiently accurate, Metalibm will either increase the polynomial degree or subdivide the domain and search for approximations in each subdomain separately. The limit of 7, therefore, does not imply that all polynomials will have this degree, but only means that the domain of the function under approximation should be reduced. We leave the remaining parameters of Metalibm to their default values.

Metalibm generates the approximation as code optimized for double floating-point precision. Since we target fixed-point precisions instead, most of the implemented optimizations for range reduction, expression decomposition and meta-splitting are not applicable to our fixed-point implementations. Our implementation thus does not reuse the generated C code. Instead, it extracts the abstract syntax tree of the generated

```
def cos_0_02to1_5_err2_9802322387695312em08(x: Real): Real = {
  require((0.02 <= x) && (x <= 1.5))
    if ((x < 1.165)) {
      c0 + (c1 + (c2 + (c4 + (c6 + (c7 + c8*x)* x)* x*x)* x)* x)* x*x
  } else {
    let t = (x - 1.33) in
       b0 + (b1 + (b2 + (b3 + (b4 + b5*t)* t)* t)* t)* t
  }
} ensuring (res => res +/- 2.9802322387695312e-08) // finite-precision budget
```

Figure 5.3.: Approximation polynomial parsed from Metalibm output.

piece-wise polynomial from the code and adds it to our top-level program as a separate function. The elementary function call is then replaced by the call to the generated function.

We currently do not support automated range reduction; for some of our benchmarks, we have reduced ranges manually during preprocessing. In general, many programs implemented in fixed-point arithmetic will not need automatic range reduction, as many kernels have by design limited ranges. For other cases, adding the automatic range reduction is possible with some engineering effort, since we already handle all necessary operations and have the ranges computed by Daisy.

Figure 5.3 shows the extracted polynomial approximating `cos (2*x1)` over the input domain $[0.02, 1.5]$ with an approximation target error of 2.98e-8 for our running example.

5.2.5. Assigning Finite Precision

Once the approximation polynomial has been generated, our algorithm assigns a finite precision to the generated polynomials. The goal is to find an assignment that satisfies the local roundoff error budget ϵ_{i_fp}, but uses as coarse precision as possible for performance reasons. The generated polynomials contain branching, but the branches are always at the top-level. For this simple structure there are no discontinuity errors, i.e. errors due to diverging control-flow between the finite-precision and real-valued execution, so that we can safely handle each branch separately.

For assigning the lowest possible finite precision to obtained polynomials, such that their roundoff error e_{fp} satisfies ϵ_{i_fp}, we again leverage the uniform or mixed-precision assignment of Daisy. Finally, we re-run the roundoff error analysis on the whole program, where elementary function errors are replaced by the sum of e_{fp} and e_{approx}. This error is potentially smaller than the originally allocated local error budget, as Metalibm or the precision assignment usually cannot exhaust the budget due to complex, discrete constraints. That is, our tool in the end reports the actually achieved error of the final implementation.

5.2.6. Alternative Algorithm Designs

Alternatively to the proposed error distribution strategy, we could have designed our algorithm bottom-up: first assign local error budgets for both approximation and roundoff errors for elementary function calls, generate their approximations, then distribute what is left of the global error budget between other operations and variables. Or, we could first assign an approximation error budget to each elementary function call, generate approximations, then use the rest of the global error budget to assign finite-precisions to the entire generated program at once.

We note that for these alternatives it is unclear how to distribute the initial error budgets. Crucial information about the actual overall error on the polynomials becomes available only later, when they are implemented. It may happen that the polynomials have used up too much of the initial error budget and there is not enough left to implement the arithmetic operations of the original program. In this case, backtracking would be necessary, which may be costly.

In our top-down approach, we still have to distribute the global error budget in the first step, but we do so on the top-level program, ensuring that all operations will have a sufficient portion of the error budget to be implemented in some finite precision (unless the overall specified error is already too small for the accurate implementation of the program).

5.3. Experimental Evaluation

We implemented our algorithm on top of the tools Daisy and Metalibm and evaluate it on several benchmarks from scientific computing, embedded and machine learning domains. In particular, we answer the following research questions:

RQ1: How do the approximations affect the accuracy/performance trade-off?

RQ2: Can kernels be optimized in reasonable time?

Benchmarks Our set of benchmarks contains programs with up to 5 elementary function calls in straight-line code (all benchmarks are available open-source[2]). The number of elementary function calls for each benchmark is shown in Table 5.2. The benchmarks *predictGaussianNB*, *predictSVC* and *predictMLPLogistic* are machine learning classifiers generated by the python scikit-learn library on the standard Iris data set. The benchmarks *forwardk2j** are taken from the Axbench approximate computing benchmark suite [168] and compute a forward kinematics expression. We have created the benchmarks *axisRotation*, rodriguesRotation*, which rotate coordinate axes and a vector respectively. The *pendulum** benchmarks come from the Rosa project for analysis of finite-precision code [13]. Finally, benchmarks *xu** and *sinxx10* are from the CORPIN project [165].

[2]`https://github.com/malyzajko/daisy/tree/approx/testcases/approx`

Experiments setup We evaluate our approach on a commonly used FPGA board (Xylinx Zync 7000 with 10ns clock period), but note that our technique is not specific to any particular hardware and believe that our results qualitatively carry over. Generation of optimized programs has been performed on a MacBook Pro with an 3.1 GHz Intel Core i5 processor and 16 GB RAM, macOS Mojave 10.14.

We perform all experiments for two different sets of target errors—small and large. To obtain these error bounds, we first run roundoff analysis on the benchmarks with uniform fixed-point precision with 32 bits. Small and large target errors are by two orders of magnitude smaller, resp. larger than these computed roundoff errors. Both target errors are reported in Table 5.1.

For performance measurements we compile our generated programs using Xilinx Vivado HLS v.2019.1, which reports the minimum and maximum number of machine cycles of the compiled design, and thus provides an exact performance measurement. We do not measure actual running time as such a measurement is necessarily noisy.

The baseline programs against which we compare correspond to the programs a user can implement with today's state of the art: by running Daisy on the input program without approximations to assign a uniform precision to all operations and then by compiling the generated code using Xilinx' elementary function library. The compiled programs can use either the fixed-point or the floating-point versions of library functions (this is decided by the compiler). For our baseline, we evaluate all valid versions (those which satisfy the overall error bound), and use the smallest number of cycles obtained.

5.3.1. RQ1: Accuracy vs Performance

The main goal of our optimization is to increase performance, therefore, we first evaluate performance improvements.

Performance Improvements

Table 5.1 compares the running time in terms of machine cycles of programs synthesized by our approach (columns 3 and 7) with the baseline implementation (columns 2 and 6) for small and large target errors. A pair '52-60' denotes minimum and maximum cycles; whenever these values coincide, we show only one number. We report the number of cycles for the fastest approximated program, obtained by distributing the global error budget using either uniform or mixed-precision assignment.

For all benchmarks, except *predictGaussianNB*, we observe a significant performance improvement when elementary function calls are replaced with piecewise-polynomial approximations. Our optimized approximate programs run on average 2.23x faster than the baseline, and up to 4.64x (4.46x) faster for small (large) target errors respectively.

For 10 out of 13 of the benchmarks, the largest speedup was achieved when using uniform precision assignment for both top-level program and polynomial approximations. For three *predict** benchmarks the best performance has been achieved using mixed-precision tuning. We believe that mixed-precision can be improved further by

Target errors	small				large			
Benchmark	baseline	approx	target	actual	baseline	approx	target	actual
axisRot.X	52-60	**24**	1.49e-10	7.52e-11	30-34	**14**	1.49e-6	5.5e-7
axisRot.Y	52-60	**24**	1.49e-10	7.52e-11	30-34	**14**	1.49e-6	5.5e-7
fwdk2jX	97-113	**23**	8.39e-11	2.98e-11	30-34	**24**	8.39e-7	2.41e-7
fwdk2jY	94-110	**22**	4.89e-11	1.49e-11	30-34	**12**	4.89e-7	1.06e-7
xu1	97-113	**43**	1.89e-10	2.47e-10	53-61	**14**	1.89e-6	1.93e-6
xu2	96-112	**44**	1.88e-10	2.3e-10	54-62	**13**	1.88e-6	1.86e-6
rodriguesRot.	52-60	**25**	1.70e-8	1.11e-8	31-35	**14**	1.70e-4	9.07e-5
sinxx10	52-60	**28**	2.51e-9	1.61e-9	31-35	**15**	2.51e-5	1.26e-5
pendulum1	33-37	**27**	4.79e-11	3.74e-11	32-36	**16**	4.79e-7	3.06e-7
pendulum2	53-61	**26**	1.07e-10	8.11e-11	32-36	**15**	1.07e-6	6.64e-7
pred.Gaus.	84	119-125	4.15e-7	4.07e-7	58	77	4.15e-3	4.08e-3
pred.SVC	22	20-28	1.46e-6	1.47e-7	21	21	1.46e-2	6.82e-4
pred.MLPLog.	195	**191**	2.15e-6	4.14e-10	143	**126**	2.15e-2	7.21e-7

Table 5.1.: Running time in machine cycles of baseline and synthesized programs, and error budgets together with the achieved accuracy

using a more accurate cost function. Disabling the refinement loop produced slower programs for 3 benchmarks and did not change results for the rest. We observed the largest speedup when using a combination of the area-based and the machine-learned cost functions.

We noticed that on the benchmark *predictGaussianNB* the baseline programs run faster than the synthesized ones. We suspect the reason is that *predictGaussianNB* repeatedly calls the `log` function on slightly different, but largely overlapping, domains. Our implementation generates a different polynomial for each call, when in this scenario reusing the code seems to be beneficial. We leave the detection of such cases to future work. We noticed that the largest improvements are observed for benchmarks with `sin`, `cos`, whereas for the `exp` function in the *predictMLPLogistic* improvements are smaller, and for *predictSVC*, our approach cannot improve the running time. We suspect this effect is due to an efficient implementation of `exp` in the Xilinx math library.

Accuracy Comparison

In Table 5.1 we also show the target errors (columns 4 and 8), as well as the errors of the best synthesized approximated programs (columns 5 and 9), for both the small and large error setting. We observe that not all of the available error budget is used up by our optimized programs. This is to be expected, as the space of precisions is not continuous. The coarser a finite precision gets, the greater becomes the difference between roundoff errors computed for two neighboring precisions, and a leftover portion of the budget might be insufficient for implementing the program with precision even 1 bit lower. This is reflected in our experimental results, where small error budgets are used up more

Benchmark	# elem. fnc calls	Small errors time	Small errors # arith. ops	Large errors time	Large errors # arith. ops
axisRot.X	2	3m 13.26s	142	41.54s	48
axisRot.Y	2	3m 1.61s	142	40.66s	48
fwdk2jX	2	5m 56.5s	222	1m 35.33s	102
fwdk2jY	2	1m 29.16s	71	24.75s	24
xu1	3	3m 50.24s	168	50.97s	61
xu2	3	6m 56.96s	212	1m 31.22s	73
rodriguesRot.	2	2m 40.15s	126	30.73s	45
sinxx10	1	1m 38.28s	71	25.8s	24
pendulum1	1	2m 18.36s	71	27.64s	22
pendulum2	1	1m 43.24s	71	23.98s	24
pred.Gaus.	5	1h 45m 27.7s	708	4m 26.231s	255
pred.SVC	1	21m 33.35s	247	1m 51.62s	95
pred.MLPLog.	1	3h 19m 48.57s	399	57m 29.185s	170

Table 5.2.: Size of the generated polynomials and the running times for optimization

than large ones: for small error budgets the average usage is 62.33%, while for large budgets it is only 41.12%.

Size of Generated Approximations

Replacing library function calls with locally defined functions naturally increases the size of the program. Table 5.2 shows the number of elementary functions and the size of the generated polynomials (sum over all elementary functions) per benchmark (for the setting with the largest performance improvement, as reported in Table 5.1). Factors that influence the reported total size are: *a)* the number of elementary function calls with distinct input ranges and local error budgets, because we generate an approximation for each of them; *b)* the local error budget and thus approximation error budget, which influences the size of each polynomial inversely, the smaller the error budget, the larger the polynomial satisfying this budget needs to be. The largest generated program has in total 708 arithmetic operations, which is still reasonable for embedded systems.

RQ1 Conclusion: Based on our experimental data, we conclude that our optimization is efficient: by using on average half of the user-defined error tolerance (budget) our optimization generated 2.23x faster kernels. Our optimization performs best on programs with transcendental functions, in particular when original programs contain multiple calls to the same function with the same arguments' ranges.

5.3.2. RQ2: Optimization Running Times

Table 5.2 shows the synthesis times of our implementation. As expected, our optimization is significantly slower on programs with small target errors than with large ones.

Smaller target errors usually require polynomial approximations with larger degrees and result in larger programs. Additionally, to satisfy smaller roundoff error bounds, finite-precision tuning has to consider higher precisions, thus searching a larger space for a suitable precision assignment.

We expect the optimization to be run once before a program is deployed onto the chip. Running time of a few hours is a reasonable price to pay for the performance boost when running the optimized programs.

RQ2 Conclusion: Our experiments confirm that the optimization running time is inversely proportional to the size of the assigned error budgets. Optimizing kernels with small target errors is on average 5x slower than with large budgets (while the smaller budgets are on average 7270x smaller than the large ones). However, optimization times for both budgets are reasonable for a one-time cost.

5.4. Related Work

Approximate Computing Our optimization trades acceptable accuracy loss for resource savings. This idea has been extensively pursued under the name of approximate computing [156, 169]. Techniques in this domain span all layers of the computing stack from approximate hardware [170] to software-based approximations such as skipping loop iterations [171], removing synchronization [172], delaying control computations for embedded systems plants [58], lossy compression of images [173], neural network quantization at the training [174] and deploying stage [175].

Most related to our work from this domain is another combination of Daisy and Metalibm [176]. However, it only considers floating-point arithmetic and, unlike our tool, uses the polynomials generated by Metalibm directly without optimizing them. A more recent tool OpTuner [177] also considers floating-point elementary function implementations with reduced accuracy. Unlike our approach and the combination of Daisy and Metalibm for floating points, OpTuner starts from a collection of different approximations and evaluates which of them are best suitable for the program under optimization. Instead of producing a single optimized implementation, OpTuner outputs a Pareto curve that depicts the accuracy/performance trade-off for several best combinations and leaves the final choice of the optimal solution to the developer. One more approximate computing tool Chisel [24] optimizes arithmetic programs by selecting which operations can be run on approximate hardware. Its error analysis is a slightly simplified version of ours in this work. While Chisel considers also probabilistic specifications, it only optimizes arithmetic operations.

Apart from Metalibm there exist other approximation generators, such as FloPoCo [178] that optimizes the implementation of mathematical operators to a specific FPGA design and then generates the VHDL code. If we were to replace Metalibm with FloPoCo in our algorithm, it would require transforming the code optimized for VHDL into our intermediate representation, which may offset some of the improvements suggested by FloPoCo.

Other work allows programmers to specify several versions of a program with different accuracy-efficiency tradeoffs, and let a specialized compiler autotune a program to a particular environment [179]. While this approach handles programs of larger size than ours, it requires the library writer to provide the different versions, together with accuracy specifications. A programming framework Green [180] focuses on energy-efficient implementations that are re-calibrated dynamically to use approximations. However, Green uses sampling to check for accuracy loss and only provides statistical (unsound) guarantees.

Approximations can be particularly efficient when run on custom hardware, such as neural processing units, for which one can learn an approximate program which mimics the original imperative one [181]. Verification is again performed only on a limited set of test inputs. STOKE is an autotuner that operates on low-level machine code and has also been applied to generate approximate floating-point programs [182]. Its scalability is limited as it considers low-level code, and furthermore it also cannot guarantee accuracy.

Finally, approximations can naturally also be applied manually, e.g. for obtaining efficient, low-resource heartbeat classifiers [183]. This particular work has approximated an exponential function by a piece-wise linear function, but due to the manual process without accuracy guarantees.

Numerical Program Analysis We reviewed sound rounding error analysis tools in subsection 2.2.2; all of them assume fixed library implementations when analyzing programs with elementary functions and do not optimize for efficiency. Consequently, mixed-precision tuning approaches guided by these analyses achieve limited performance improvements when dealing with elementary function calls, especially those that only consider floating-point precisions [23,25]. Our presented work leverages the much larger tradeoff space of fixed-point arithmetic *and* elementary function approximations and achieves significantly larger performance savings.

Mathematical Libraries While the goal of our approach is to reduce the accuracy of elementary function calls, the results are correct modulo assigned error budget. Previous work has also verified accuracy of existing library functions [184] and compared different library implementations [185]. Alternatively, libraries have been designed with correct rounding in mind for single [186] and double floating-point precision [187] and for several alternative precisions [26].

Program Synthesis Program synthesis [188] aims to automatically generate programs from (possibly declarative) specifications, and has had considerable success to generate programs from a variety of domains [189–196]. One can view optimizations as a form of synthesis, where the original program and an optimization metric form an input specification. However, the vast majority of the synthesis techniques require that the generated program satisfies the user-given specification exactly. Furthermore, most approaches do not explicitly optimize for a non-correctness metric.

A branch of program synthesis – automated repair – allows to modify parts of a program to satisfy given criteria. The tools AutoRNP [20], Herbie [19,197] and the tool by Wang et.al. [21] repair numerical programs by detecting an input subdomain that triggers high floating-point errors and rewriting expressions with approximations on this subdomain. Opposite to our approach, repair tools aim to increase accuracy and often introduce time overhead for repaired programs [20]. A recent modification of Herbie [197] combines accuracy optimization with precision tuning and improves the running time as well.

The Metasketches framework [198] searches for an optimal program with smallest cost according to a cost function. It has been used for synthesizing polynomial approximations, however, the accuracy of the generated programs is only verified based on a small set of test inputs, and thus without accuracy guarantees. In contrast, Metalibm's polynomial approximation algorithm is guaranteed to find the best polynomial approximation, and our entire approach guarantees end-to-end accuracy.

Dynamic Optimizations Apart from the already mentioned dynamic tools AutoRNP, Herbie and the work of Wang et.al, multiple other optimizers improve performance metrics with precision tuning: HiFPTuner [77], FloatSmith [10], the tool by Lam et. al [78], Precimonious [51], STOKE-Float [182], TAFFO [79]. Furthermore, precision tuning has also been applied to improve accuracy of mathematical functions with FPDebug [199]. However, these tools are guided by dynamic analysis and inherently cannot provide soundness guarantees unless they are additionally combined with sound methods [200,201].

5.5. Conclusions and Future Work

We presented a performance optimization for numerical kernels with elementary function calls. Our optimization trades off a controlled portion of accuracy (the assigned error budget) for performance and replaces elementary function calls with piece-wise polynomial approximations. Our main contribution is the algorithm that automatically distributes the error budget among all sources of errors in the program: rounding errors on the operations in the original program, approximations of elementary functions and rounding errors in the approximating polynomials. The key observation that inspired our algorithm is that errors of different origins behave similarly and therefore can be distributed together.

Using Up Error Budget Another observation we made was that due to the discrete nature of errors in finite precision implementations, it is difficult to use up the whole error budget. It is more challenging for larger budgets because the difference in accuracy between neighboring precisions is larger when the precision is already coarse.

However, there are ways to improve the usage of the error budget. For instance, when a polynomial is already fixed and the approximation error is known, if it is smaller than the assigned budget, we could add the leftover part to the finite-precision budget.

Secondly, we could improve the polynomial approximation generation by implementing range reduction [163, 177] for fixed-point approximations. The range reduction in Metalibm is designed specifically for floating points and cannot be directly applied to fixed points. Moreover, since we target numerical kernels to be executed on FPGAs, the optimal approximation for fast executions may look different from those targetting regular CPUs and floats [202].

Equivalent Function Calls Another way of improving approximations is to scale back on how much we customize the polynomials for each call. Our experimental evaluation revealed that our method can be improved on benchmarks with multiple calls of the same function. Currently, we only reuse the approximating polynomials for repeated function calls if both local error budget and the range of the arguments match exactly for the call occurrences. For the benchmark *predictGaussianNB* this strategy does not bring much, because the repeated calls to the `log` function have slightly different argument ranges. This can be improved by reusing the cached approximations more often, for instance, by introducing a margin around arguments' ranges to make several function call occurrences virtually equivalent. However, one has to choose the margin magnitude carefully, since too small of a margin will keep similar calls apart, and a too-large margin may significantly increase the input domain for the approximation and require complex and expensive-to-evaluate polynomials.

Handling Complex Control-Flow Another direction of future work is to extend our optimization method beyond straight-line kernels. Currently, we only optimize the kernels because the error budget distribution algorithm requires an appropriate finite-precision assignment method that is not available for complex control-flow statements. Even a uniform precision assignment is challenging in the presence of loops and conditionals since it depends on the rounding error analysis, which is a difficult problem in itself.

The problem is even more complex for mixed precision as different iterations may require different precision assignments. One possible solution would be to assign the precision to stay constant across iterations, but it may be suboptimal, because values' magnitudes may differ significantly across iterations. Another possibility would be to detect the iterations that require extraordinarily fine (respectively, coarse) precision and move them out of the loop. This, however, may lead to splitting a single original loop into several loops and make the end code less readable.

6. Meta-Optimization: Regime Inference

Existing sound optimizations of finite-precision programs target different sides of the accuracy-performance trade-off spectrum. For instance, our approximation synthesis method, described in the previous chapter, gives up a controlled amount of accuracy for better performance. A similar principle is the basis of mixed-precision tuning [22, 23, 51] that reduces accuracy by using low precision on a subset of operations and keeps the overall error bound below some user-specified value. On a different side of the spectrum are optimizations that target accuracy, for instance, by rewriting them [19–22, 49] using real-valued identities or approximations.

Since rounding errors depend on the magnitude of an expression's variables, such optimizations are necessarily specialized for a particular user-defined input domain. However, most current tools consider the specified input domain as a whole, that is, they generate one optimized expression for the entire domain. This often leads to suboptimal results, because rounding errors typically vary across the input domain, and so the optimized expression may not be the ideal choice for a large part of the input domain. Recall, for instance, the program `carthesianToPolar_radius` and its error profile (for uniform double precision) that we presented earlier and depict again in Figure 6.1. The plot shows the absolute errors of the function's results for different inputs; darker color depicts larger rounding errors. The plot indicates that inputs from the top right corner induce higher rounding errors, and thus sub-domains closer to the top-right corner will require a higher (mixed) precision assignment than bottom-left parts of the domain, and may require a different rewriting than other parts of the domain.

A few recent tools [19–21] propose to generate *regimes*—a partition of the input domain into sub-domains, each with a different optimized version of the program. These tools apply rewrites in order to repair high rounding errors in certain parts of the domain. While they can successfully improve programs that suffer from large numerical issues, they estimate errors using a dynamic analysis (sampling) and thus do not provide accuracy *guarantees*. Furthermore, they are not immediately applicable for optimizing numerically stable code, i.e. *without* particularly large rounding errors.

In this chapter, we present the first regime inference for *sound* floating-point optimizations[1], i.e. for optimizations whose accuracy analysis computes guaranteed worst-case error bounds for all possible (specified) inputs. Our approach partitions the input domain and optimizes each part separately with an existing sound mixed-precision tuning or rewriting optimization routine, improving performance or accuracy, respectively. By doing so, we provide a significant benefit also for numerically stable code.

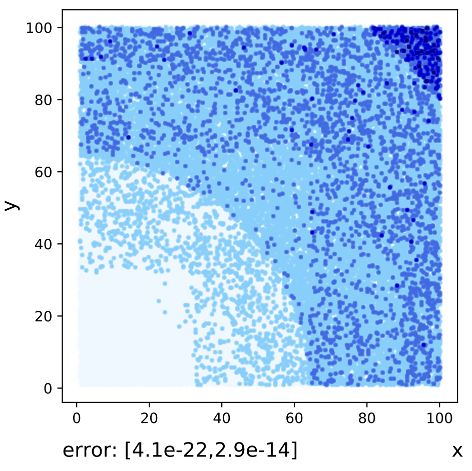
[1]We discuss extensions to other formats later in section 6.6.

```
def carthesianToPolar_radius(x: Real, y: Real): Real = {
  require(((1 <= x && x <= 100) && (1 <= y && y <= 100)))
  sqrt(((x * x) + (y * y)));
} ensuring(res => (res +/- 2.51e-14))
```

(a) Source code of `carthesianToPolar_radius`. The `require` clause specifies the input domain.

Error magnitude legend:

- ● [4.1e-22, 5.8e-15]
- ● [5.8e-15, 1.2e-14]
- ● [1.2e-14, 1.7e-14]
- ● [1.7e-14, 2.3e-14]
- ● [2.3e-14, 2.9e-14]



(b) Error profile for `carthesianToPolar_radius`

Figure 6.1.: Example program `carthesianToPolar_radius` that computes polar 'radius' from Cartesian coordinates

Inferring *effective* regimes for sound optimizations is nontrivial. In particular, we cannot simply take partitions derived by a dynamic analysis from one of the existing tools [19–21]. While the error profile in Figure 6.1b, which was also obtained with a dynamic analysis, appears to suggest certain partitions, these do not necessarily lead to partitions for which sound optimizations will provide any improvement. This is because we need to find regimes for which a sound optimization routine can *prove* that a particular error bound holds. Since sound error analyses necessarily abstract rounding errors, they commit over-approximations that are hard to predict a priori, and that make it difficult to guess a regime by sampling. Furthermore, the space of possible optimizations is highly discontinuous. For instance, mixed-precision tuning considers only a small number of distinct precisions for each variable, and changing the precision of only a single variable can have a disproportionate impact on the overall error (recall Figure 1.1b).

In principle, there are infinitely many possible regimes and enumerating all of them is infeasible, especially because today's sound optimizations are relatively expensive. Furthermore, each domain split results in at least one conditional statement in the final generated code, which introduces an additional cost at runtime and decreases code readability. Moreover, for multivariate programs splitting along one variable's domain can be more beneficial than splitting along the other.

Due to the inherent complexity, we do not attempt to infer optimal regimes. Instead, we focus on finding regimes with *interval sub-domains* and combine two heuristic approaches inspired by techniques that have been successful in the area of floating-point analysis. Interval sub-domains allow to keep the regimes' cost low, as lower and upper bounds of each variable can be efficiently checked for at runtime.

Our algorithm first starts generating regimes *bottom-up*. Inspired by interval subdivision [14, 39], it splits the input domains of all variables to create a fixed number of sub-domains. It optimizes each separately and then attempts to merge sub-regimes with the same optimized expressions. Next, the algorithm proceeds in a *top-down* fashion inspired by branch-and-bound techniques [15]. Starting with the sub-domains generated by the bottom-up phase, the algorithm iteratively splits some of them on-demand, in each iteration selecting the variable to split based on the optimization objective. This combined technique allows to explore the space of regimes in both breadth as well as depth and avoids getting stuck at a local optimum.

Our regime inference algorithm is generic in the optimization and the optimizer. In this chapter, we instantiate it with the currently available sound optimizations for floating-point arithmetic: rewriting and two mixed-precision tuning routines from Daisy [22] and FPTuner [23]. These tools can optimize straight-line numerical expressions consisting of arithmetic and elementary operations. Unfortunately, no sound tool exists that can directly optimize loops since even bounding rounding errors in loops is a largely unsolved orthogonal problem [13]. That said, errors in embedded control loops are often handled with control-theoretic techniques [36], so that the loop body can be optimized in isolation, and thus by our approach. For loops with a limited number of iterations, our approach can be applied to the unrolled loop, at the expense of (significantly) increased program size.

Here, we only consider IEEE-754 floating-point arithmetic, which is supported by both Daisy and FPTuner. However, our approach is also applicable to other arithmetics. Provided a suitable hardware-specific cost function, our algorithm can infer regimes for fixed-points programs (e.g., using Daisy, FPTuner does not support fixed-point arithmetic). For more complicated arithmetics, like mixed integer and finite-precision programs, currently there are no tools to perform sound optimizations directly (i.e. without approximating integers by floating-points), but once such tools appear, they can be immediately used within our framework (together with an appropriate cost function).

We evaluate our approach on 100 benchmarks from the standard benchmark set FPBench [84] and show that our algorithm infers regimes that on average improve performance by 65% and accuracy by 54%, compared to Daisy's whole-domain optimizations, and by 52% compared to FPTuner's mixed-precision optimizations.

Contributions In summary, this chapter describes the following contributions:

- we present the first *sound* regime inference algorithm,
- that we implement in a prototype tool called Regina, available open-source at `https://github.com/malyzajko/daisy/regimes`, and
- extensively evaluate it using mixed-precision tuning and rewriting optimizations and show that regime inference is highly beneficial for sound floating-point optimizations.

```
   def azimuth(lat1: Real,lat2: Real,lon1: Real,lon2: Real):Real = {
2    require(((0.0 <= lat1) && (lat1 <= 0.4) &&
      (0.5 <= lat2) && (lat2 <= 1.0) &&
4     (0.0 <= lon1) && (lon1 <= 3.142) &&
      (-3.142 <= lon2) && (lon2 <= -0.5)))
6
     val dLon: Real = (lon2 - lon1)
8    val slat1: Real = sin(lat1)
     val clat1: Real = cos(lat1)
10   val slat2: Real = sin(lat2)
     val clat2: Real = cos(lat2)
12   val sdLon: Real = sin(dLon)
     val cdLon: Real = cos(dLon)
14   atan((clat2 * sdLon) / ((clat1 * slat2) - ((slat1 * clat2) * cdLon)))
    } ensuring((res) => (res +/- 4.57e-14)) // 0.5x double error
16 // ensuring((res) => (res +/- 9.15e-15)) // 0.1x double error
```

Figure 6.2.: Source code of the `azimuth` benchmark

6.1. Example

Before explaining our regime inference algorithm in detail, we provide a high-level overview. Consider the function `azimuth` in Figure 6.2 that computes the angle between an observer and a point of interest with a reference plane. Similar computations frequently appear in domains such as cyber-physical systems or robotics, where they are executed often, so should run fast, but where they may also need high accuracy. The example is specified as a real-valued function, together with a precondition (`require` clause on line 2) that bounds the possible input values, as well as a postcondition (the `ensuring` clause in line 15) that specifies a maximum absolute error on the result. In practice, the input domain would be, for instance, determined from valid ranges of sensors, and the maximum allowed error from the sensitivity of actuators, or stability proofs in the case of controllers [36].

Regina's goal is to find an effective partition of the program's input domain such that the program can be soundly optimized on each sub-domain separately, leading to an overall reduction in a given cost metric. In this paper, we consider two optimizations, mixed-precision tuning and rewriting, that consider running time and accuracy of the generated code as the cost metric, respectively. We call an optimized expression together with the sub-domain it has been optimized on a *sub-regime*. A set of non-overlapping sub-regimes covering the whole input domain is called a *regime*, and the *size of regime* denotes the number of sub-regimes.

Mixed-Precision Tuning Recall mixed-precision tuning optimization [22, 23, 51] that assigns (potentially different) finite-precision types to each variable and arithmetic operation to increase performance while satisfying a user-defined error bound. Suppose that a user has specified that the worst-case absolute error of the function `azimuth` should be 4.57e-14. The sound roundoff error analysis tool Daisy [14] determines a worst-case error of 9.15e-14 when all operations are in uniform double floating-point precision, which is not enough to meet the target error bound. Since the error is close to the target error (less than an order of magnitude), it might be enough to increase the precision for only a subset of variables. Unfortunately, when Daisy's mixed-precision tuning algorithm [22] optimizes the program on the whole specified input domain, it assigns quad precision [2] to all but the input variables. The resulting program is as slow as the uniform quad precision implementation.

Our regime inference tool Regina, parametric in the optimization and optimizer, can be used on top of Daisy's mixed-precision tuning to find a faster program than Daisy alone. First, Regina subdivides the input domain into 24 sub-domains, and runs Daisy's mixed-precision tuning on each of them separately. This optimization results in the same precision assignment on several sub-domains, which are subsequently (partially) merged. For the target error of 4.57e-14, the 24 original sub-domains are merged into 5. In a second step, our top-down phase starts from these 5 sub-domains and attempts to find additional splits that reduce the (abstract) cost. In our example, Regina finds one more beneficial split and returns 6 sub-domains. In fact, Regina finds that only a single sub-domain (out of the resulting 6) actually requires mixed-precision:

$$\text{lat1} \in [\mathbf{0.2}, \mathbf{0.4}], \text{lat2} \in [\mathbf{0.5}, \mathbf{0.625}], \text{lon1} \in [\mathbf{2.094}, \mathbf{3.142}], \text{lon2} \in [-\mathbf{3.142}, -\mathbf{1.821}]$$

Regina encodes the sub-domains using if-then-else statements. We show the structure of the resulting generated program in Figure 6.3a, and the code generated for the one mixed-precision sub-domain in Figure 6.3b. The remaining 5 sub-domains use uniform `double` precision. The generated C code meets the user-specified error bound and *runs 93% faster* than the mixed-precision implementation that Daisy alone generates.

For a tighter error bound of 9.15e-15 as on line 16 in Figure 6.2 (an order of magnitude smaller than the uniform double error), Regina generated 11 sub-regimes of which 5 need mixed precision and the remaining ones can still be implemented in uniform double precision. The bottom-up phase again generates 24 sub-domains initially, and merges these into 10. The top-down phase subsequently splits one of these so that 11 sub-regimes are generated overall. The generated code runs 84% faster than the uniform quad implementation that Daisy alone generates.

Rewriting We further instantiate Regina's regime inference with the rewriting optimization (also provided by Daisy) that attempts to improve the worst-case absolute error bound of an expression using real-valued equivalence rules (i.e. the cost metric is accuracy). For our example, such rewriting can be applied on the computation on line 14 in Figure 6.2. Assuming uniform double precision, Daisy's rewriting applied to the whole domain is only able to improve the maximum error by 1%.

[2]IEEE quad precision has 128 bits, but libraries, such as GCC's quadmath [203] that we use often provide slightly less precision for increased performance.

```c
if ((lat2 <= 0.75)) {
  if ((lat2 <= 0.625)) {
    if ((lon1 <= 2.0943951)) {
      // uniform double
    } else {
      if ((lon2 <= -1.820796325)) {
        if ((lat1 <= 0.2)) {
          // uniform double
        } else {
          // **mixed-precision**
        }
      } else {
        // uniform double
      }
    }
  } else {
    // uniform double
  }
} else {
  // uniform double
}
```

(a) Structure of generated code

```c
__float128 dLon = ((__float128)lon2 - (__float128)lon1);
__float128 slat1 = sinq((__float128)lat1);
__float128 clat1 = cosq((__float128)lat1);
__float128 slat2 = sinq((__float128)lat2);
__float128 clat2 = cosq((__float128)lat2);
double sdLon = (double)sinq(dLon);
double cdLon = (double)cosq(dLon);
double _tmp3 = (double)(clat2 * (__float128)sdLon);
double _tmp1 = (double)(clat1 * slat2);
double _tmp = (double)(slat1 * clat2);
double _tmp2 = (_tmp * cdLon);
double _tmp4 = (_tmp1 - _tmp2);
double _tmp5 = (_tmp3 / _tmp4);
return atan(_tmp5);
```

(b) Mixed-precision sub-regime

Figure 6.3.: Sub-regimes in C generated for `azimuth` benchmark

Regina generates 23 sub-regimes with 11 unique rewritten expressions, including, for instance:

```
sdLon * ( clat2 / ((slat2 * clat1) - (clat2 * (cdLon * slat1) )) )
(clat2 * sdLon) / ((clat1 * slat2) - (slat1 * (clat2 * cdLon)))
clat2 * (sdLon / ((clat1 * slat2) - ((slat1 * cdLon) * clat2)))
```

The inferred regime has only 11 distinct rewritings, because sub-regimes with the same rewritten expression are not necessarily neighboring each other and thus cannot be merged. Note that we did not limit the number of branches for this optimization, since we optimized for accuracy, though it is straight-forward to customize Regina to limit the number of branches, if needed. We observed the cost of our simple conditional branches to be very small, especially compared to the additional cost that rewriting may introduce due to additional operations (e.g. when applying distributivity). The generated code has a (proven) worst-case error that is 36% smaller than the expression which Daisy generates for the whole domain.

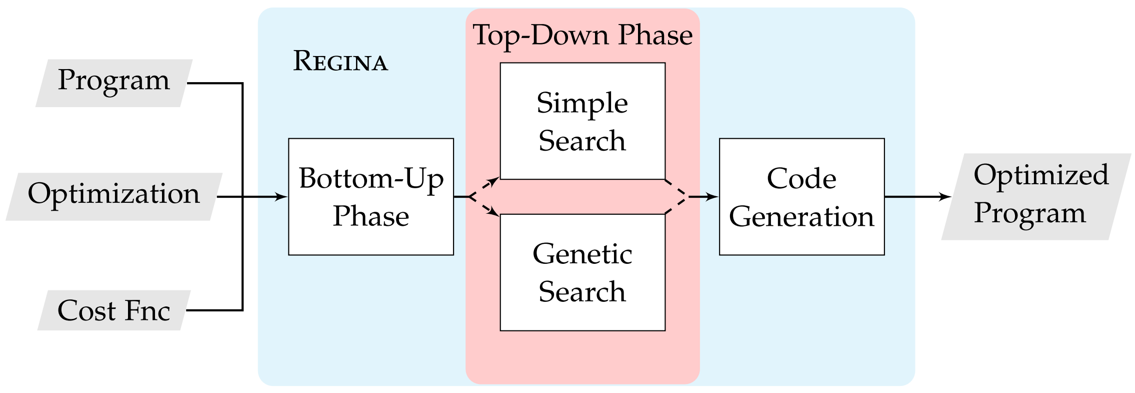


Figure 6.4.: Regime inference algorithm

6.2. Regime Inference Algorithm

We focus on sound floating-point optimizations that guarantee or optimize a worst-case rounding error bound that holds for all specified inputs. Since the magnitude of rounding errors heavily depends on the domain of an expression's variables, different domains allow for different optimizations. While it is possible to partition an input domain using complex expressions, in this work we focus on sub-domains described by intervals for two main reasons. First, at runtime a program needs to evaluate partition conditions to decide which optimized version (sub-regime) to execute. Evaluating complex expressions takes longer than a simple interval bounds check, and may offset the performance improvements of each sub-regime (when optimizing for performance with mixed-precision tuning). Lastly, complex conditional expressions may themselves introduce rounding errors, for which we would have to account; which by itself is a non-trivial task [38]. Secondly, the rounding error analysis in state-of-the-art sound numerical programs' optimizers [14,23] is fundamentally interval-based and thus does not leverage non-interval sub-domains well.

Interval sub-domains can be checked for efficiently with conditional statements (see subsection 6.2.3). While such checks incur a negligible cost compared to the rest of the execution, we nonetheless want to limit the number of sub-regimes—to improve readability of the generated code and to reduce the running time of our regime inference procedure itself.

Regina's algorithm works in two phases: first running the bottom-up phase to explore the sub-domains up to an initial depth, and then the top-down phase that explores the space further, on demand, with one of two available search procedures. Figure 6.4 illustrates the high-level algorithm. The bottom-up phase, explained in detail in subsection 6.2.1, exhaustively subdivides the input space and then attempts to merge sub-regimes with equal optimizations. The top-down phase, explained in subsection 6.2.2, includes two alternative search procedures, each of which divides the given domain on demand, guided by a (static) cost function. Both phases are motivated by two techniques that have been successful in reducing over-approximations in rounding error analysis in the tools Daisy and FPTuner, respectively.

Our approach is generic w.r.t. the floating-point optimization and the cost metric that is being optimized. In the following, we will thus use the function `optimize` to stand for some optimization routine that takes as input a domain, an expression and possibly a target rounding error bound and that returns a new, optimized expression. The `cost` function takes an expression and its domain as input and returns a numeric value reflecting the optimization objective; we will assume that lower cost is better. In section 6.3, we show how we instantiate these algorithms with two different optimizations, mixed-precision tuning and rewriting, that optimize performance and accuracy, respectively.

We illustrate our algorithm using a running example, in which we infer a regime for mixed-precision tuning on the `carthesianToPolar_radius` function from Figure 6.1a. In our example, the `optimize` function uses Daisy's mixed-precision tuning, and `cost` statically estimates the abstract performance of each tuned regime of `carthesianToPolar_radius`. `cost` does not estimate the actual running time, but rather an abstract cost that only needs to distinguish which of two regimes is likely to be faster. The detailed instantiation of `optimize` and `cost` for mixed-precision tuning is described in subsection 6.3.1. In contrast to our approach, Daisy's mixed-precision tuning alone applied on the `carthesianToPolar_radius`'s whole input domain (`x`$\in$ $[1.0, 100.0]$, `y`$\in$ $[1.0, 100.0]$) did not result in any measurable performance improvements.

6.2.1. Bottom-Up Phase

The regime inference algorithm starts by exploring possible regimes in a bottom-up phase. It is inspired by interval subdivision, a technique that has been used in static rounding error analysis tools to reduce over-approximations [14, 39].

Figure 6.5 shows the pseudo-code of the bottom-up phase. First, it subdivides the input domain uniformly into smaller pieces and optimizes each one individually. We split the each variable's interval into equal pieces, as it is not obvious up front, i.e. before running the actual optimization, which sub-division will be beneficial. The number of initial sub-regimes clearly influences the possible improvements of regime inference, however, calling optimization on too many sub-regimes is expensive. In our implementation we currently limit the maximum number of sub-regimes that a method can generate to 32. Hence, depending on the number of input variables, we subdivide each variable's domain between 16 and 2 times[3]. We found empirically that larger initial sub-division size only increases the algorithm's running time, and does not change resulting regimes significantly. Understandably, on sufficiently small sub-domains an optimizer can no longer improve an individual sub-regime cost (e.g. in mixed-precision it already uses the lowest available precision).

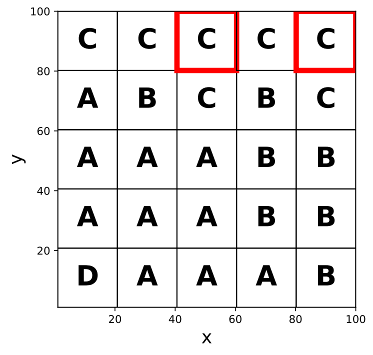
The obtained optimized expressions on individual sub-domains form the initial regime. We have observed that often the optimized program bodies in sub-regimes are equal in

[3]For programs with more than 5 variables we subdivide the 5 largest input intervals in half and leave the rest unchanged.

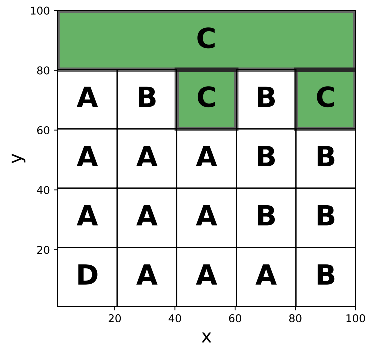
```
def bottom_up_phase(inputRanges, program, target):
  // produce regimes
  subdomains = splitTillMax(inputRanges)
  // run optimization
  regime = []
  for sub in subdomains:
    optProgram = optimize(sub, program, target)
    regime = regime ∪ (sub, optProgram)
  // merge sub-regimes
  regime = mergeSameBodies(regime)
  return regime

// merge sub-regimes with the same bodies
def mergeSameBodies(regime):
  subdomains = regime.subdomains
  for sub1, sub2 in neighbors(subdomains)
    if programIn(regime,sub1) == programIn(regime,sub2):
      upd = sub1 ∪ sub2
      expr = programIn(regime,sub1)
      toRemove = {(sub1,expr),(sub2,expr)}
      regime = regime ∪ (upd,expr) \ toRemove
      subdomains = subdomains ∪ {upd} \ {sub1,sub2}
```

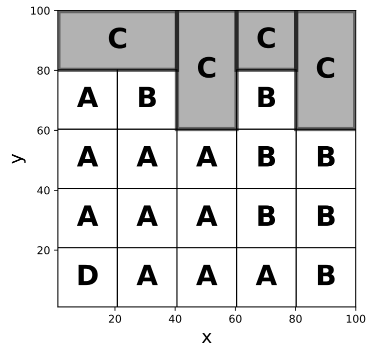
Figure 6.5.: Bottom-up phase



(a) Input regime



(b) Optimal



(c) Sub-optimal

Figure 6.6.: Merge strategy

this initial regime. In the second step, our algorithm tries to find a smaller regime by merging the neighboring sub-domains whose body is the same (function `mergeSameBodies` in Figure 6.5). We call two sub-domains (and the corresponding sub-regimes) *neighboring* if they differ in ranges for exactly one variable, and after merging these ranges the new sub-domain does not overlap with any existing sub-domain. Having a smaller regime (i.e. fewer sub-regimes) is beneficial for readability of the generated code, as well as for the running time of our tool, as the running time of the successive top-down phase heavily depends on size of the given regime.

The initial regime for our running example is shown in Figure 6.6a. Here, the input domain is split into 25 sub-domains (splitting the interval for both variables `x` and `y` into 5 equal-sized intervals), and the labels A-D denote different optimized expressions: A - all operations and intermediate results are assigned a `double` precision, B - one intermediate result is assigned `quad` precision, the rest use `double`, in C two variables have `quad` precision, and in D - four.

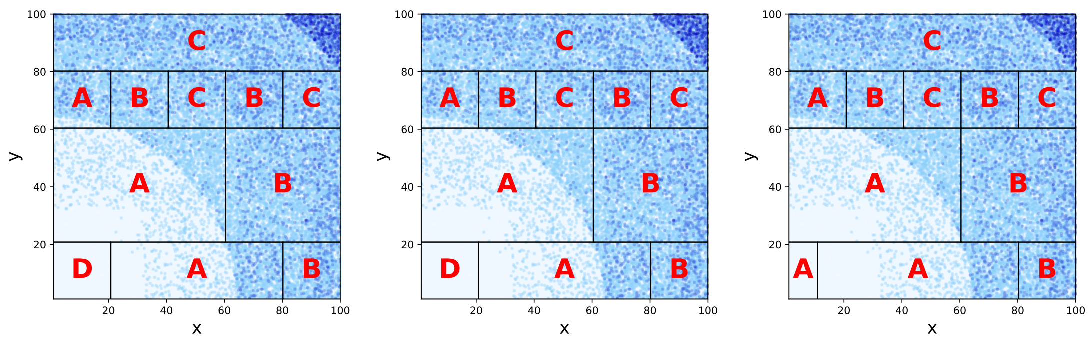
Finding the optimal merge with minimum number of resulting sub-regimes is an exact set cover problem, which is known to be NP-complete. Hence, our algorithm uses a heuristic to decide which neighbors should be merged: it starts with the first input variable (as they appear in the source code) and performs all possible merges along this variable, then repeats for the rest of input variables. The algorithm merges along each variable once. Depending on the order of variables in the source code such a heuristic can overlook beneficial joins. Consider an initial regime in Figure 6.6a that contains 10 sub-regimes with the optimized expression C. Red squares mark sub-regimes that have neighbors with the same optimized expressions along two variables: `x` and `y`. Merging along the variable `x` (horizontal axis) is clearly beneficial, as it will result in fewer sub-regimes in total (see Figure 6.6b). However, if the variable `y` appears in the list of function arguments before `x`, the algorithm will first merge along `y` (vertically) and create a sub-optimal result shown in Figure 6.6c.

For our running example the bottom-up phase produces a regime that includes 11 sub-regimes, as shown in Figure 6.7a. Compared to the version of `carthesianToPolar_radius` optimized on the whole specified domain, a program with this regime runs 45% faster. Starting from this regime, our algorithm will try to further improve performance in the top-down phase.

6.2.2. Top-Down Phase

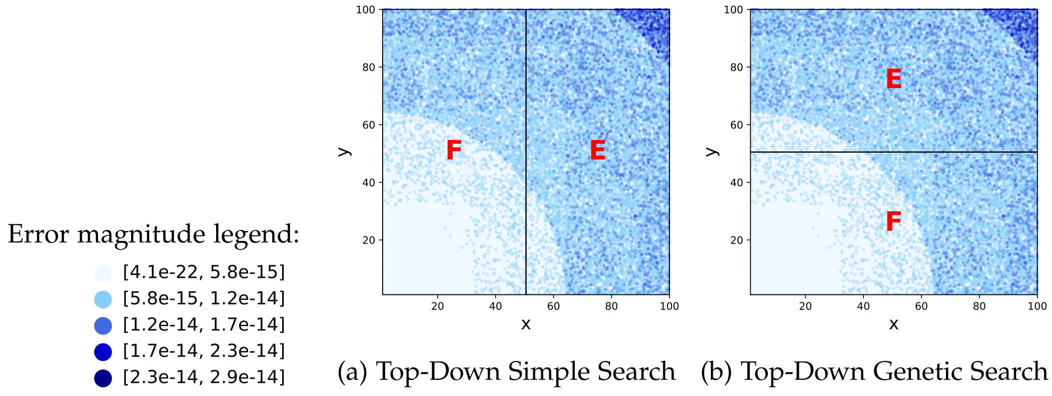
The next phase is inspired by the branch-and-bound technique that is being used by FPTuner [15] to find a domain-specific optimum of an arithmetic expression. Since the errors are not necessarily distributed uniformly over the input domain, it is often beneficial to sub-divide on-demand along selected variables. Our top-down phase starts from the regime found by the bottom-up phase (in Figure 6.7a) and tries to find a regime with even lower cost by repeatedly splitting variables' domains. We consider two approaches: 'simple' and genetic searches. Both search approaches are guided by a static cost function, they stop if either no further cost improvement is possible in a single split, or when the algorithm has reached some maximum number of iterations.

For multivariate programs it is non-trivial to choose how to split a domain. One can split along one variable—split the range of one variable, while keeping ranges of the other variables intact—split along a subset of variables or all of them. Furthermore, we need to select a split point on each of the ranges. As for the merging strategies of the bottom-up phase, there is no way to know in advance which direction of split will be



(a) Bottom-Up Phase (b) Bottom-Up + Top-Down Simple (c) Bottom-Up + Genetic

Figure 6.7.: Regimes inferred at different stages of the algorithm.



(a) Top-Down Simple Search (b) Top-Down Genetic Search

Figure 6.8.: Regimes inferred by top-down phase starting with the whole input domain.

beneficial. Similarly, finding the most beneficial split point would require additional analysis of the domain and is expensive.

Simple Search Due to this inherent complexity, our simple search procedure attempts to find a lower-cost regime using a heuristic: split only along one variable in the middle of its range.

The pseudo-code for the top-down phase with the simple search is shown in the Figure 6.9. The algorithm attempts to split across each variable separately, then selects the split with the lowest cost and disregards the others. As a result, at every step the algorithm splits along one variable that provides the largest cost improvement. Such a strategy allows us to create regimes that are at least as good as the starting point (according to the cost function).

The algorithm terminates if it found a regime, upon which it cannot improve by splitting further. In general, the top-down algorithm is not guaranteed to terminate, therefore we limit the depth of splitting by a constant `maxDepth`.

```
def top_down_phase(bUpRegime, program, target):
  def simpleSearch(regime, oldCost, depth):
    candidates = []
    depth += 1
    for var in program.inputVars:
      newRegime = []
      newSplit = splitInHalf(regime.subdomain, var)
      for sub in newSplit:
        newProgram = optimize(sub, program, target)
        newRegime = newRegime ∪ {(sub, newProgram)}
      newCost = cost(newRegime)
      candidates = candidates ∪ (newRegime, newCost)
    // regime with the smallest cost
    bestRegime, bestCost = sortByCostAsc(candidates).hea
    // check for improvement
    if bestCost < oldCost:
      if depth < maxDepth
        return simpleSearch(bestRegime, bestCost, depth)
      else:
        return bestRegime // the depth is exhausted
    else:
      return regime // previous regime

  bUpCost = cost(bUpRegime)
  return simpleSearch(bUpRegime, bUpCost, 0)
```

(a) Top-down phase with simple search

```
def geneticSearch(regimes, counter):
  candidates = []
  for r in regimes:
    newCost = cost(r)
    candidates = candidates∪{(r, newCost)}
    sortedRegimes = sortByCostAsc(candidates)
  if counter > maxGenerations:
    bestRegime, bestCost = sortedRegimes.head
    return bestRegime
  else:
    nextPopulation = []
    for i in range(0 until populationSize):
      // mutate regimes
      regToMutate = rankedChoice(sortedRegimes)
      vars = regToMutate.inputVars
      sortedVars = sortByRangeWidth(vars)
      varToMutate = rankedChoice(sortedVars)

      newSplit = splitRandomly(regToMutate,
           varToMutate)
      newProgram = optimize(newSplit,program,target)
      newRegime = (newSplit, newProgram)
      nextPopulation = nextPopulation ∪ newRegime

    return geneticSearch(nextPopulation, counter+1)
```

(b) Genetic search procedure

Figure 6.9.: Top-down phase with alternative regime search procedures

For our running example the inferred regime has not changed after applying the simple search (see Figure 6.7b), splitting either of sub-domains in the middle did not improve the cost.

Genetic Search Note that the regime returned by the top-down phase is not necessarily optimal. As illustrated by our running example, the search can give up too early, when a single split exactly in the middle of a variable's range does not improve the cost, but a different (potentially deeper) split would. To allow different splits and help overcome the local minima problem, we propose an alternative search procedure `geneticSearch` from Figure 6.9b that uses randomization in form of a genetic algorithm. Unlike the simple search, the genetic approach allows for multiple possible regimes to exist in parallel and selects sub-domains to be split and a split point randomly.

We instantiate the genetic algorithm framework in the following way: a regime represents an individual to be mutated, a collection of regimes is a population, and splitting the range of a variable is a mutation. As in the simple top-down search we start from the regime returned by the bottom-up phase, this regime forms the initial population. In every generation, the algorithm chooses a regime and variable that will be mutated using ranked choice [204]. First, all regimes are sorted based on their cost and one is selected. Inside the selected regime, the variables are ranked according to the width of their range. Once the variable is selected, the algorithm splits the range at a random point (but at least 5% of the width).

Note that selecting a regime and variable to be mutated with ranked choice is simply an instantiation of randomness and can be replaced by any other heuristic. Because of the randomness and multiple regimes existing in parallel, the genetic search is less likely to get stuck in a local minimum, compared to the simple search that at each iteration keeps only one regime with the smallest cost and splits sub-domains exactly in the middle of variable's range.

We use a population size of 10 and repeat the loop for 10 generations (we have not observed a noticeable improvement in the results with larger values). On the final population the algorithm performs a post-processing step that merges identical neighbors using the `mergeSameBodies` function from bottom-up algorithm.

Starting from the results of the bottom-up phase the genetic top-down phase has inferred a regime for our running example shown in Figure 6.7c.[4] While it is largely similar to the results of the bottom-up phase with and without the simple top-down search, a slight shift of the sub-domain bounds (bottom left corner) allows a sound optimizer to prove smaller error bounds and assign lower precision, thus further increasing the performance gain by 10%. A version of `carthesianToPolar_radius` with the resulting regime in Figure 6.7c runs 54% faster than the whole-domain optimized version.

Input Regime The quality of regimes produced by the top-down phase with both simple and genetic search clearly depends on the starting point, i.e. input regime. While it is also possible to start from the whole domain (the program's initial input ranges) even with randomization the top-down phase is likely to give up too early when it gets stuck in a local optimum. To illustrate this, we have applied the top-down phase with both search procedures to our running example's whole domain. It inferred 2 sub-regimes shown in Figure 6.8, both of which are using at least 5 variables in `quad` precision. Even though with more fine-grained regime it is possible to use lower precision (as illustrated in Figure 6.7), the top-down phase was not able to find a lower-cost regime with one split. The bottom-up phase, on the other hand, has already explored the domain up to a certain depth. For many benchmarks this preliminary exploration is sufficient to overcome the local optimum.

6.2.3. Code Generation

Once a regime was found, our tool Regina generates code that uses conditional branches to select the appropriate expressions at runtime. A naive approach to generating the

[4]Our heuristic merging strategy could not merge two neighboring sub-regimes with expression A in the bottom-left corner. `mergeSameBodies` merges along each variable only once, here, first `x`, then `y`. When merging along `x`, the sub-domain x$\in$ $[1.0, 10.9]$,y$\in$ $[1.0, 19.8]$ was further split along `y` and therefore did not satisfy the definition of a neighboring sub-domain.

output code for a regime of size n would be to output a linear succession of conditional statements that cover exactly one sub-regime each. However, this approach requires $O(kn)$ number of tests, where k is the number of input variables, and n is the size of a regime.

Instead, Regina generates nested conditional statements leading to the individual regime bodies, where in each test we check the value of only a single variable. The asymptotic behavior of the number of tests performed is now $O(log(n))$, keeping the cost of conditional branches low, especially since every test itself is relatively cheap.

While for a single input variable it is always possible to generate one path for each sub-regime, this does not hold in general for multivariate functions. By generating nested conditionals, we generate code with more paths than the number of sub-regimes. That is, a sub-regime may be described by a union of several paths. Regina generates the conditional branches one input dimension at a time, starting with the variable for which there exists a sub-regime with the largest sub-domain.

For mixed-precision tuning, code generation furthermore needs to account for the fact that different sub-regimes may assign different precisions to the function's input and output variables. To preserve soundness, Regina assigns for each input variable and the return expression the highest precision that one of the sub-regimes has assigned. For each regime part where the input or return precisions do not exactly match the upper bound of all input and return precisions, we introduce downcasts. Note that this casting procedure is also accounted for in the cost function in order to penalize inferred regimes that necessitate many casts.

6.3. Optimizations

We instantiate regime inference in our implemented tool Regina with two optimizations: mixed-precision tuning and rewriting that optimize for average running time and worst-case absolute error, respectively. Note that our regime inference algorithms can be instantiated with any optimization objective. For instance, one could optimize the regimes with respect to worst-case performance, or average rounding error. The only adjustment necessary to change the objective is to provide an appropriate cost function.

6.3.1. Regime Inference for Mixed-Precision Tuning

For mixed-precision tuning we consider the two existing openly available sound tuning tools Daisy and FPTuner.

Daisy We first instantiate Regina with Daisy's mixed-precision tuning routine [22], by calling Daisy as the `optimize` function. Daisy uses delta-debugging, a kind of divide-and-conquer algorithm, to search through different mixed-precision assignments, and calls a dataflow analysis to compute the rounding error of each.

FPTuner Separately, we instantiate Regina with FPTuner's mixed-precision optimization [23], which uses a fundamentally different technique. FPTuner formulates the search for a mixed-precision assignment as an optimization problem, which it solves using a sound branch-and-bound interval solver. While this technique often produces better results (programs with lower running time) [22], the tuning process is also more expensive than the one in Daisy.

Cost Function For instantiating our regime inference algorithms, we further need a cost function that reflects the optimization objective and that will guide the search and compare the quality of regimes. We choose to optimize the *average performance* of a program and assume that a program's inputs are uniformly distributed in the input domain. We assume a uniform distribution for convenience and simplicity, and note that it is possible to take into account a different distribution by adjusting the parameters of the cost function. We optimize for average, instead of worst-case, performance, because often one of the sub-regimes will need to use the highest precision, and thus no optimizations would be possible under a worst-case metric. That said, it is possible to account for the worst-case execution time by estimating it additionally and pruning the regimes that do not satisfy a constraint.

To estimate the (abstract) average performance of a program with regime, our cost function first computes an abstract *arithmetic cost* on each individual sub-regime. We use the term arithmetic cost to denote the performance of a floating-point expression without branches (i.e. on a single sub-domain). To compute the arithmetic cost, we use Daisy's existing simple mixed-precision cost function. It assigns to each 128-bit arithmetic and cast operation twice the cost of the same operation in 64 bits, and has been shown to work well for mixed-precision tuning between double and quad precision [22]. Note that it is also possible to tune between any other pairs of precisions (e.g. 64 and 32 bits), provided a suitable arithmetic cost function. For computing the arithmetic cost, we are deliberately using an existing cost function previously shown to be adequate, as it is Regina's parameter and not a contribution of this work.

Since the goal is to increase *average* performance, next, the cost function computes a weighted average arithmetic cost of each sub-regime. Finally, the cost function adds an offset for the number of sub-regimes to account for branching. The final cost of a regime is thus computed as follows:

$$cost_{mp}(regime) = \sum_{i=1}^{n} w_i A_i + (n-1)$$

where n is the number of sub-regimes, w_i is an ith sub-regime's weight that corresponds to the sub-domain's volume normalized to the whole domain's volume[5], and A_i is an arithmetic cost of the ith sub-regime. Even though evaluation of branching conditions

[5]For non-uniformly distributed inputs the weight w_i can reflect the probability of the i-th sub-regime being executed.

is cheap, we still add a small offset $(n-1)$ to avoid inferring branches with negligible performance improvements.

6.3.2. Regime Inference for Rewriting

For rewriting, Regina calls Daisy's rewriting routine [22] as the `optimize` function. This optimization searches for an order of evaluation that is equivalent to the original expression under a real-valued semantics, but for which Daisy can prove a smaller rounding error bound (using its sound analysis). Since floating-point arithmetic does not satisfy common real-valued identities such as associativity and distributivity, reordering a computation in general leads to different results (and roundoff errors), even though the expression is equivalent under the reals. Daisy searches through the different evaluation orders using a genetic algorithm, applying real-valued identities as the mutation operation.

The goal of our regime inference for sound rewriting is to minimize the *worst-case rounding error* across sub-regimes. Accordingly, Regina uses a regime's maximum rounding error as the `cost` function. First, the cost function computes the arithmetic cost of an individual sub-regime. We use Daisy's worst-case rounding error analysis with the interval abstract domain to bound variables' ranges and affine arithmetic for errors. The overall cost is the maximum error seen across all sub-regimes:

$$cost_{rw}(regime) = \max_{i \in [1,n]} err_i$$

where n is the number of sub-regimes, and err_i is the worst-case absolute rounding error of the i-th sub-regime. Since we are optimizing for accuracy and not performance, and testing inputs' bounds does not affect accuracy of the computed value, we do not add any cost to prune additional branches. The regime inference algorithms themselves limit a total number of sub-regimes, so the resulting program will not have unreasonably many branches, and the branches generated can be evaluated efficiently. If needed, the cost function can straight-forwardly be extended to also account for the increased running time.

6.4. Experimental Evaluation

We evaluate Regina on a standard benchmark set for floating-point analysis, and compare sound optimizations with and without regime inference. In particular, we focus on the following research questions:

RQ1: Does regime inference improve over whole-domain optimizations?

RQ2: Is the two-phase approach beneficial over each one separately?

```
:precision binary64
:pre x ∈ [0.001, 1.5]
```
$\frac{1}{x} - \frac{1}{\tan(x)}$

(a) Input expression

```
:precision binary64
:pre x ∈ [0.001, 1.5]
```
if ($\frac{1.0}{x} - \frac{1.0}{tan(x)} \leq 0.008964$) { $x \cdot 0.3(3) + (0.02(2) \cdot x^3 + 0.002116 \cdot x^5$) }
else { $\frac{1.0 \cdot ((\tan(x))^3 - x^3)}{x \cdot (\tan(x))^3 + (x \cdot \tan(x)) \cdot (x + \tan(x))}$ }

(b) Herbie's optimized expression with regime

Figure 6.10.: NMSE-example-3.9 benchmark

6.4.1. Benchmarks

We evaluate regime inference on the FPBench benchmark set [84], a standard benchmark set for floating-point verification and optimization tools. For those benchmarks that originally contain loops, we generate a new version that consists of the loop body only (i.e. corresponds to one loop iteration). We exclude benchmarks that contain conditional statements, as well as benchmarks for which Daisy is not able to compute a roundoff error, e.g. when Daisy's analysis is not precise enough to show that a division is safe, i.e. does not divide by zero.

Many FPBench benchmarks already come with preconditions that bound the ranges of inputs. We use these as the initial domains for our optimization. When a precondition is missing or does not provide a closed range for all variables, we add input range bounds ourselves. For a few benchmarks, Daisy is not able to compute the roundoff error for the original precondition, but it is able to do so for a slightly modified—more constrained—one. In these cases, we consider the modified precondition for our experiments. In total, we consider 100 out of the 131 FPBench benchmarks, including 32 that contain elementary function calls and 15 that contain square root operations. All benchmarks with the pre- and post-conditions that we used for the evaluation are available open-source[6].

The existing and chosen input variable domains cover realistic preconditions, but are relatively small in the sense that they do not contain e.g. very large values close to the maximum possible values (for instance, double floating-point precision supports exponents of up to 2^{1023}). With such preconditions, most of the benchmarks are numerically stable in the sense that the committed rounding errors are not very large, as computed by state-of-the-art sound rounding error analysis tools [14,71].

6.4.2. Comparison with Herbie

Regina is the first tool that infers regimes for *sound* floating-point optimizations, i.e. those that guarantee that the rounding error of an optimized program does not exceed

[6] `https://github.com/malyzajko/daisy/tree/regimes/testcases/regime-inference`

a specified bound. In contrast, today's state-of-the-art tools that infer regimes [19–21] use *dynamic analysis* to estimate rounding errors and therefore do not provide rounding error guarantees. Hence, there is no regime inference tool that we can directly compare to.

For completeness, we nonetheless perform a comparison with the dynamic analysis-based tool Herbie [19] that is closest to Regina in terms of the optimization that is being applied. Herbie's goal is to reduce (large) rounding errors by rewriting an arithmetic expression. Error guarantees aside, the goal is similar to when Regina is instantiated with Daisy's rewriting optimization.

First, Herbie randomly samples points and identifies which inputs cause large rounding errors, then it isolates these inputs into a sub-domain, and, when possible, improves the errors on this sub-domain with rewriting. Unlike Daisy, Herbie rewrites not only using real-valued identities, but also polynomial approximations (that are not real-semantics preserving). As a consequence, Herbie's greedy rewriting often provides larger accuracy improvements, while Daisy's sound rewriting improves by a smaller factor but on more programs [200].

Two other tools—AutoRNP [20] and the tool by Wang et.al. [21]—are further away from Regina's goal. Like Herbie, they identify large rounding errors using a dynamic analysis, but their rewrite rules are more specialized or do not preserve real-valued semantics, i.e they only use approximations. Since the more specialized rules are largely not applicable to the general-purpose FPBench benchmarks, and it is not meaningful to compare error bounds obtained on semantically different expressions, we do not compare Regina's results with AutoRNP and the tool by Wang et.al.

We run Herbie on all of our benchmarks four times to account for randomness, since it is using heuristic search and randomly sampled inputs. Herbie created regimes only for two benchmarks out of 100. In all four runs, Herbie created a regime for the *nmse_example_3.9* benchmark, the example regime is shown in Figure 6.10b (exact output slightly differs among the runs). Additionally, in one of the runs Herbie also found a regime for a second benchmark, *nmse_example_3.3*. For both benchmarks Herbie used rewrite rules that do not preserve real-valued semantics, thus, we cannot compare the optimized expression's rounding error with Regina's results (the same reason we do not compare with AutoRNP, Wang et.al).

These limited results are not particularly surprising, given that Herbie's stated goal is to repair *large* rounding errors—numerical instabilities. The results confirm that regime inference for—especially sound—floating-point optimizations of numerically stable code is missing.

We conclude that Herbie (and the other existing repair techniques [20,21]) are complementary to Regina's goal: they can be used to first repair a program with large errors, so that Regina can optimize branches of the resulting program.

6.4.3. Experimental Setup

Mixed-Precision Tuning

The goal of mixed-precision tuning is to reduce the running time of an arithmetic expression as compared to a uniform precision implementation, while nonetheless meeting a user-provided error bound. For our evaluation, we thus have to define target error bounds, the precisions that we consider for tuning, as well as a suitable comparison baseline.

Following previous work[7], we generate two sets of *target error bounds*. We first compute the rounding error for a given benchmark assuming uniform 64 bit double precision, and then multiply this error by 0.5 or 0.1 to obtain the target error bound. We choose two error bounds for each benchmark, because different bounds provide for different optimization opportunities. Choosing a smaller target error (using the factor 0.1), we generally expect less opportunities for mixed-precision tuning, and less improvements w.r.t. a uniform precision baseline. We will denote the benchmark set with error factor 0.5 by *half-double benchmarks*, and the benchmark set with factor 0.1 by *order benchmarks* (for an order-of-magnitude smaller error).

For comparison with Daisy's mixed-precision tuning, we compute the 64 bit double precision errors using Daisy, and for the comparison with FPTuner we compute the baseline errors correspondingly with FPTuner (since they use different techniques, the errors generally differ). It is not a goal of this paper to compare Daisy's or FPTuner's tuning, rather we want to show that regime inference is beneficial for both techniques.

As in previous work [23], we consider mixed-precision tuning with *double and quad precision*, where quad is implemented by the GCC quadmath library [203]. The goal is to improve the running time over a uniform quad precision implementation of each benchmark. For hardware platforms where single and double precision (32 and 64 bit) have different running times, tuning would be equally possible (with an appropriate cost function).

We compare the running time of programs generated by Regina against the running time of programs generated by Daisy's mixed-precision tuning. To ensure a fair comparison, we run Daisy's tuning using its subdivision method for computing ranges, using the same number of subdivisions as in the bottom-up phase. By doing so, we avoid seemingly improving over Daisy simply by using a more accurate range computation method. We compare Regina instantiated with FPTuner's mixed-precision tuning routine against FPTuner alone.

Mixed-precision programs are generated as C code that is compiled with `g++` 9.3 with the flags `-O2 -fPIC` and whose running time we measure using C's `high_resolution_clock` on 10^6 uniformly distributed random inputs. We repeat the measurement three times and take the average of those three runs for comparisons. We compute the *improvements* as (baselineTime - regimeTime)/baselineTime. We checked that performance improvements

[7]It has been observed that mixed-precision tuning is most useful when the target error bound is *just* below a uniform precision error [14,23].

computed this way are accurate within 0.02, hence we count a benchmark's performance as improved if the improvement is larger than 0.02.

Rewriting

We evaluate regime inference instantiated with Daisy's rewriting and compare the accuracy improvements w.r.t. to rewriting without regimes. Similarly to mixed-precision tuning experiments, we run Daisy's vanilla rewriting with the subdivision method for computing the ranges for a fair comparison. We compute the improvement in worst-case absolute error as $(baselineError - regimeError) / baselineError$.

Hardware Details

Because FPTuner requires Ubuntu, we run our mixed-precision tuning experiments on a compute cluster node with a dual-core Intel Xeon E5 v2 processor at 3.3 GHz and 16x16GB RAM running Ubuntu 16.04.7. We run the experiments with the rewriting optimization on a Mac mini with an 6-core Intel i5 processor at 3 GHz with 16 GB RAM running macOS Catalina, because the rewriting optimization runs in parallel and runs significantly faster on a 6-core machine.

We set a timeout of 30min per benchmark for all experiments.

6.4.4. RQ 1: Improvements over Whole-Domain Optimizations

Table 6.1 summarizes our experimental results for the three different optimizations: Daisy's mixed-precision tuning, FPTuner's mixed-precision tuning and Daisy's rewriting. We have marked in bold the overall best results. For FPTuner, we report only results of the first *or* the second phase of our algorithm alone, because the running time of FPTuner's mixed-precision tuning is very high, and running the two-phase algorithm led to timeouts for most benchmarks.

Regina improves running time over Daisy's mixed-precision tuning for 73 half-double benchmarks with an average improvement of 65.7%, and improves 46 order benchmarks with an average improvement of 65.6%. Regina also improves over FPTuner's mixed-precision tuning for up to 31 half-double benchmarks with an average improvement of 52.2%, and for 18 order benchmarks with an improvement of 56.3%. The number of order benchmarks improved is lower, as expected, since a smaller error bound provides less opportunities for optimizations.

Regina with rewriting is able to improve the worst-case error for 62 out of 100 benchmarks with an average improvement of 54.4%. That is, with regime inference, we are able to essentially half the optimized (proven) error at compile time.

The improvements by regime inference that we report in Table 6.1 are w.r.t. to the *already optimized baseline*. For comparison, Daisy's vanilla mixed-precision tuning without regimes improves performance over a *uniform quad precision baseline* by only 28% and 25%, respectively for half-error and order benchmarks, and Daisy's rewriting without

	method	improv. >0.02	avrg. improv.	# best	regime size >1	avrg. regime size	avrg. runtime (s)	TO
Daisy mixed-tuning	*half-double*							
	bottom+genetic	**73**	**65.7%**	**54**	57	5.3	67.8	14
	bottom+top	74	64.4%	56	62	5.6	50.0	14
	bottomUp	74	63.5%	39	63	5.2	52.2	13
	topDown	64	60.6%	35	60	3.9	66.5	11
	genetic	72	62.5%	37	60	3.1	93.9	9
	order-error							
	bottom+genetic	**46**	**65.6%**	**35**	52	4.8	160.5	21
	bottom+top	45	65.6%	28	55	5.0	141.9	24
	bottomUp	47	58.5%	21	56	4.6	133.9	21
	topDown	40	52.4%	21	37	2.5	55.4	10
	genetic	45	61.1%	28	38	2.0	181.7	11
FPTuner	*half-double*							
	bottomUp	**31**	**52.2%**	**-**	50	7.8	665.6	23
	topDown	27	44.5%	-	30	2.7	677.8	19
	order-error							
	bottomUp	**18**	**56.3%**	**-**	30	5.3	656.6	35
	topDown	18	43.3%	-	25	3.2	574.1	18
Rewriting	**bottom+genetic**	**62**	**54.4%**	**59**	45	7.2	383.5	15
	bottom+top	53	40.5%	48	48	7.4	191.4	7
	bottomUp	43	44.3%	43	45	7.0	279.8	7
	topDown	43	52.5%	39	56	7.9	212.0	11
	genetic	58	48.1%	41	54	4.7	306.5	11

Table 6.1.: Summary statistics for different optimizations, comparing regime inference against optimizations without regimes. Column 2 gives the number of benchmarks for which there is an improvement over the optimized baseline without regimes, column 3 gives the average improvement over those benchmarks, column 4 gives the number of benchmarks for which a method produces an improvement that is within 2% of the best result among all methods. We do not report the number of the best results per method for FPTuner (marked '-'), as it is not meaningful for comparing only two methods. Columns 5 and 6 give the number of benchmarks where generated regime has multiple sub-regimes, and the average size of regimes. Columns 7 and 8 give the average running time of regime inference and the number of benchmarks that timed out.

regimes improves accuracy w.r.t. the *original* expressions by only 13%. We conclude that regime inference with Regina provides significant performance improvements over mixed-precision tuning without regimes, as well as accuracy improvements over rewriting.

Details: Mixed-Precision Tuning In fact, e.g. the bottom-up approach generates *uniform double* precision code for 24 half-double benchmarks, i.e. does not use mixed precision at all. The reason why Daisy was not able to discover this uniform precision is that even though we run Daisy with the interval subdivision method for computing accurate ranges, the optimization itself nonetheless considers the entire domain at once, which leads to over-approximations. For the order benchmarks, Regina generates uniform double precision code for 6 benchmarks. Hence, an additional side-effect of regime inference is that it reduces inherent over-approximations of the static analysis for individual optimizations.

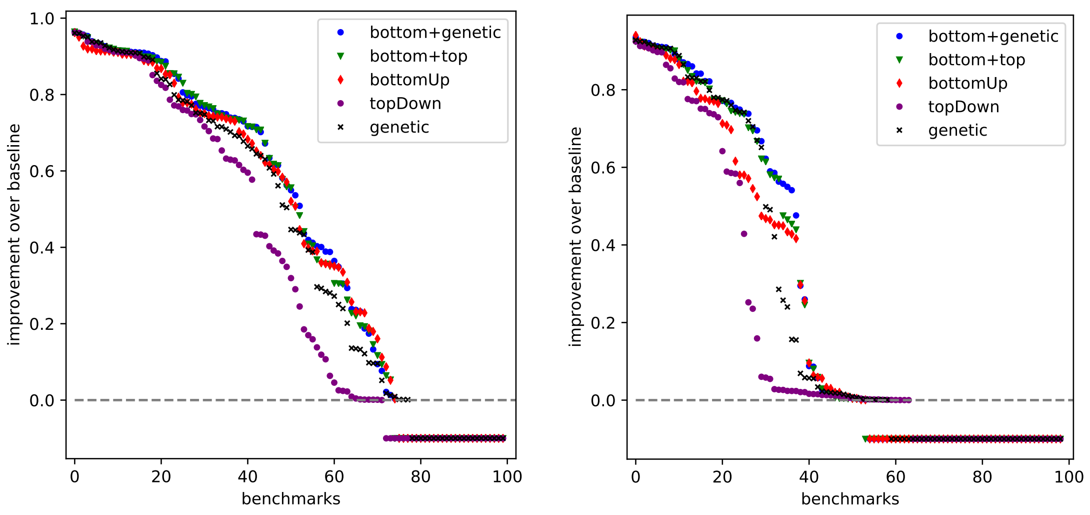
Details: Rewriting The bottom part of Table 6.1 further shows that the number of benchmarks with regime size greater than one, i.e. with several sub-regimes, is smaller than the number of benchmarks improved overall. This is due to the fact that during regime inference, the verification is performed on smaller sub-domains, which leads to a smaller overall computed error bound, which in turn may help to discover a suitable rewriting. Since Regina merges sub-regimes with equal expressions in the end, we may end up with just a single expression. Thus, the 'on-demand' splitting performed by the bottom+genetic, bottom+top and genetic methods helps to find sub-domains for which suitable rewriting can be found (and proven).

RQ1 Conclusion: Our experiments confirm that our regime inference algorithm is general with respect to floating-point optimizer and optimization. It reliably improves performance using sound optimizers Daisy and FPTuner, which internally use different techniques, and accuracy using Daisy's rewriting. Our regime inference provided improvements for a significant portion of the benchmarks—over a baseline that has already run an optimization.

6.4.5. RQ 2: Evaluation of Two-Phase Approach

Table 6.1 also lists a number of variations of regime-inference methods. Our full two-phase algorithm (as described in section 6.2) runs the bottom-up then the top-down phase with genetic or simple search and is denoted by 'bottom+genetic' and 'bottom+top', respectively. Furthermore, we evaluate each of the two phasess separately: 'bottomUp' method stands for applying the bottom-up phase alone, and 'topDown' and 'genetic' methods are results of applying the top-down phase with a corresponding search procedure to the whole specified input domain (as opposed to resulting regime of the bottom-up phase). Note that we limit the number of regimes that the top-down and genetic methods are allowed to consider to the same number of subdivisions that the bottom approach generates for a fair comparison.

We compare the performance of these methods visually in the cactus plots in Figure 6.11 and Figure 6.12, for Daisy's mixed-precision tuning and rewriting optimizations, respectively. That is, we have sorted the performance and accuracy improvements for each method individually, hence vertically aligned points do not always correspond



(a) Half-double benchmarks

(b) Order benchmarks

Figure 6.11.: Performance improvements of Regina over Daisy's mixed-precision tuning (cactus plot). Series of values higher and more to the right are better.

to the same benchmark. Values below 0.0 in Figure 6.11 correspond to timeouts and slowdowns. For clarity, Figure 6.12 shows only those benchmarks for which one of the methods provides some accuracy improvement.

Overall, we observe that the combined approach (bottom+genetic and bottom+top) performs better than each phase of the algorithm alone (bottomUp, topDown and genetic). The top-down phase with the simple search procedure alone (topDown method), while still outperforming Daisy, performs worst overall. Our hypothesis is that it gets stuck in a local optimum, whereas the top-down phase with genetic search and combined phases overcome these local optima thanks to randomization and an initial exploration of the domain.

The effectiveness of the combined two-phase approach comes at the expense of increased running time to compute the regime inference, and correspondingly also more timeouts (> 30min). We have not observed memory to be an issue for Regina.

Since the genetic search procedure relies on randomization, we evaluated the influence of different random seeds on the generated performance over three runs. We observe that the variation in generated performance improvements is small (e.g. averages are within 2%), hence we conclude that the genetic method is able to improve performance reliably.

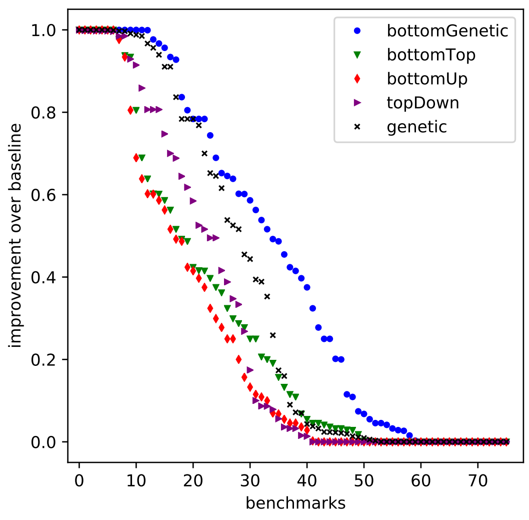


Figure 6.12.: Accuracy improvements of Regina over Daisy's whole-domain rewriting optimization. Series of values higher and more to the right are better.

RQ2 Conclusion: Based on our experimental results, we conclude that the regime inference algorithm performs best when it uses the two-phase approach: a combination of breadth-first and depth-first searches. Initial exploration using the bottom-up (breadth-first) strategy overcomes local minima; the discovered sub-domains can be further specialized and improved with either of the top-down strategies (simple or genetic).

6.5. Related Work

Regime Inference As discussed before, closest to our work are tools Herbie [19], AutoRNP [20] and the tool by Wang et al. [21]. They use a dynamic analysis to locate inputs with large rounding errors and based on these, infer regimes on which different types of repairs are applied: piecewise-quadratic or Taylor-based approximations [19,20], or expression rewrites based on real-valued identities [19,21]. Herbie has recently been extended with a new set of rewriting rules inferred by the tool Ruler [205]; additionally, a Pareto version of Herbie combines rewriting with precision tuning [197], which is similar in goal to Regina with Daisy's combination of these optimizations [22]. By relying on a dynamic analysis, these techniques fundamentally target and generate a different kind of regime. Specifically, for numerically stable expressions, i.e. those where rounding errors do not vary widely, it is—by definition—difficult for dynamic analysis to identify problematic inputs and thus to find partitions.

It is also not straight-forwardly possible to use a dynamic analysis to determine input domain partitions and then to run a sound optimization technique. Since sound tools inherently need to use abstractions, they will likely only be able to prove (and optimize) very different input partitions than the one identified by dynamic analysis.

Domain Partitioning Sound piece-wise polynomial approximations as used in our performance optimization described in chapter 5 and in related tools [163, 176] can also be viewed as a special form of regime inference. Precisely, the sub-domains of the piece-wise polynomial and their individual polynomials on each sub-domain form a regime. However, such regimes are limited to a single input variable.

Partitioning of programs' input domain is widely used by sound verification tools to reduce the over-approximation on error abstractions. Existing techniques apply to floating-point programs [14, 39] as well as a mixture of floating-point code with bit-level operations [206–208].

Floating-Point Optimizations We instantiated our regime inference for mixed-precision tuning with optimization routines of Daisy and FPTuner. Daisy and FPTuner are using sound dataflow analysis and branch-and-bound optimizers to find their precision assignments. Alternatively, one might consider mixed-precision tuning that uses a combination of backward static analysis and SMT solving [25] and rewriting with abstract equivalence graphs in Salsa [52]. For programs where soundness guarantees are not required, our regime inference could also be instantiated with optimizations guided by dynamic analysis, as in Precimonious [51], STOKE-Float [182] or algorithmic differentiation in ADAPT [209] recently applied by FloatSmith [10] and other dynamic optimizations [12, 77, 78, 197, 210–213]. Another target for regime inference may be optimizing techniques applied to numerical programs in the context of approximate computing, such as arithmetic operations that with a certain probability return an erroneous value [24] (for more details on approximate computing see section 5.4).

Opposite to the fully automated approaches proposed in this thesis, one may want to involve developers in crafting optimized code. A recent tool Odyssey [214] has introduced an interactive mode where users can learn about especially problematic inputs obtained with dynamic analysis, modify the input domains on the fly and tune the optimized expressions.

6.6. Conclusion and Future Work

In this chapter, we have shown that regime inference is beneficial not only for repairing large floating-point rounding errors, but for sound floating-point optimizations targeting numerically stable code as well.

Even though we consider relatively simple interval-based regimes, these have proven to be remarkably successful in optimizing the performance and accuracy of straight-line expressions. The success comes exactly because of this simplicity: interval-based regimes allow for efficient runtime checks and are well-supported by today's sound floating-point analyzers.

We observe that the major cost in sound regime inference are the individual optimizations themselves, and we show that a combination of breadth-first and depth-first search is an effective strategy for exploring input domains.

Other Optimizations Regime inference may also be beneficial for other optimizations beyond rewriting and mixed tuning. For instance, one could instantiate regime inference with the performance optimization introduced in chapter 5 or the similar combination of Daisy and Metalibm for floats [176]. Applying regime inference may reduce the degrees of polynomials needed for the approximations since inputs for elementary function calls will be subdivided. However, the approximation synthesis technique by itself is relatively slow and may lead to long optimization times when applied with REGINA.

Fixed-Point Precision To use regime inference with the exact version of our performance optimization from chapter 5 we first need to extend the inference algorithm to fixed-point precisions. A crucial difference between floats and fixed points is floating-point's dynamic range. Because the fractional bits for fixed-points are allocated (flexibly) at compile time, we cannot simply merge neighboring sub-domains with the same optimized bodies. When the sub-domains are merged, the range of values that have to be represented by the selected precisions grows. Hence, the fixed-point format used on the merged domain may need to be adjusted to avoid overflow, and precision (total number of bits) may have to be increased. We therefore have to account for this change when merging and splitting the sub-regimes.

Programs with Complex Control Flow Due to the limitations of the optimizations, with which we parametrized regime inference, it currently works only on straight-line kernels. When methods for sound optimizations of numerical programs with loops and conditionals appear, our regime inference algorithm could be applied on top of it.

7. Conclusions and Future Work

Implementing numerical software is hard; a developer has to get a lot of details right for a program to compute the expected result. Luckily, there exist plenty of tools to support developers in this challenging task.

Solving the general problem of analyzing and optimizing *all numerical programs* with arbitrary control-flow components is hard, so instead we break it down into smaller subproblems, for which solutions are possible. In this thesis, we described our contributions that improve sound analyses and optimizations of numerical programs. We summarize our key takeaway points.

Takeaway 1 A functional way of specifying numerical algorithms can help scale up the rounding error analysis while still reporting reasonably tight error bounds. The use of semantic information from the functional specifications is particularly efficient when it is combined with abstractions.

Takeaway 2 The "noise" introduced by finite-precision implementations can be viewed as a positive thing. When analyzing a noisy program, the candidate solutions do not have to be exact. We can benefit from efficiency of (unsound) heuristics and use them to come up with a good starting point for analyses (for instance, a candidate invariant), and later confirm them with sound methods.

Takeaway 3 Errors of different origins behave similarly in numerical programs. For instance, both polynomial approximation and finite-precision errors propagate through kernel's computations together. Therefore, we can controllably increase different sources of errors and create fast approximate implementations in more than one way.

Takeaway 4 Specializing to small parts of a program's input domain allows better optimizations. This approach is beneficial for increasing accuracy and performance (both with polynomial approximations and mixed-precision tuning). The specialization allowed us to generate simpler approximations and go around worst-case errors blocking the optimization without sacrificing soundness.

To summarize, we have improved sound methods for reasoning about finite-precision programs and made a significant step toward supporting developers in writing efficient and accurate implementations of numerical algorithms.

That said, there are many things to discover yet.

Outlook Apart from the important sub-problems solved in this thesis, other classes of programs are yet to be handled. For instance, to improve the scalability of analyses beyond loops with data structures and explore ways of keeping the computed sound error bounds tight when scaling up. The optimization support for loops and conditionals is also limited and can be further advanced.

In this thesis, we considered popular floating- and fixed-point number systems. However, there exist many alternative representations. Specialized for performance for machine learning Bfloat16 [215], TensorFloat32 [216] and MSFP (Microsoft Floating Point) [217] representations have been used alone and in a combination with IEEE-754 floats [215,216]. A different flavor of finite precision is implemented in the posit number system [63,218]. Unlike IEEE-754 floating points, posits dynamically adjust the number of precision bits depending on the represented value, but fundamentally also provide only finite precision.

These formats are likely to result in different error profiles in a program's input domain, i.e. largest and smallest errors may appear for different inputs than with IEEE-754 floats. However, our methods are independent of a particular error profile. Therefore, with some modifications, for instance, with a proper selection of the relative error parameter in the abstraction (Equation 2.13), we expect our analyses and optimizations to be useful also for alternative-precision programs.

As pointed out by a recent study [219], numerical computations become increasingly heterogeneous on both software and hardware levels. As the heterogeneity of systems increases so does the need to adapt analysis and optimization tools beyond IEEE-754 floats and regular CPUs. One direction for improving existing analyses and optimizations is to explicitly take into account differences in hardware, for instance, when a part of the program is executed on a GPU.

Another interesting direction of work would be to make the tools for rigorous support of numerical software interactive. As a recent experience of the tool Odyssey [214] suggests, numerical experts working together with the automated approaches can craft more efficient and accurate code than the automated tools or the developers alone. While Odyssey uses Herbie's (unsound) dynamic analysis, it would be interesting to extend the approach with sound methods as well.

With all that, we would be able to create better software with reliable and efficient numerical computations!

List of Figures

List of Tables

Bibliography

[1] P. Gervasio, A. Quarteroni, and F. Saleri, *Scientific Computing with MATLAB and Octave*, vol. 2. 03 2014. → page 1

[2] C. P. C. Woodford, *Numerical Methods with Worked Examples: Matlab Edition*. Springer, 2 ed., 2012. → page 1

[3] *Journal of Mathematical Chemistry*, vol. 61. Springer, 2023. → page 1

[4] J.D.Murray, *Mathematical Biology*. Springer, 3 ed., 2002. → page 1

[5] K. Quinn, "Ever had problems rounding off figures? this stock exchange has," *The Wall Street Journal*, p. 37, 1983. → page 1

[6] P. Mark J. Nigrini, "Round numbers: A fingerprint of fraud," 2018. → page 1

[7] T. T. Johnson, D. M. Lopez, P. Musau, H.-D. Tran, E. Botoeva, F. Leofante, A. Maleki, C. Sidrane, J. Fan, and C. Huang, "Arch-comp20 category report: Artificial intelligence and neural network control systems (ainncs) for continuous and hybrid systems plants," in *International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH20)*, vol. 74, pp. 107–139, 2020. → pages 1, 8, 29, 32

[8] J.-L. Lions, L. Luebeck, J.-L. Fauquembergue, G. Kahn, W. Kubbat, S. Levedag, L. Mazzini, D. Merle, and C. O'Halloran, "Ariane 5 flight 501 failure report by the inquiry board," 1996. → page 1

[9] General Accounting Office Washington DC Information Management and Technology Div., "Patriot missile defense: Software problem led to system failure at dhahran, saudi arabia." https://apps.dtic.mil/sti/citations/ADA344865, 02 1992. → page 1

[10] M. O. Lam, T. Vanderbruggen, H. Menon, and M. Schordan, "Tool integration for source-level mixed precision," in *2019 IEEE/ACM 3rd International Workshop on Software Correctness for HPC Applications (Correctness)*, pp. 27–35, 2019. → pages 1, 24, 100, 127

[11] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "MPFR: A Multiple-Precision Binary Floating-Point Library with Correct Rounding," *ACM Trans. Math. Softw.*, vol. 33, pp. 13–es, jun 2007. → pages 2, 44, 131

[12] N. Ho, E. Manogaran, W. Wong, and A. Anoosheh, "Efficient floating point precision tuning for approximate computing," in *ASP-DAC'17*, 2017. $\rightarrow$ pages 1, 127

[13] E. Darulova and V. Kuncak, "Towards a Compiler for Reals," *TOPLAS*, vol. 39, no. 2, 2017. $\rightarrow$ pages 2, 3, 4, 8, 17, 20, 22, 31, 32, 40, 42, 55, 56, 61, 81, 94, 105

[14] E. Darulova, A. Izycheva, F. Nasir, F. Ritter, H. Becker, and R. Bastian, "Daisy - Framework for Analysis and Optimization of Numerical Programs," in *TACAS*, 2018. $\rightarrow$ pages 2, 3, 8, 17, 20, 24, 25, 31, 44, 56, 62, 71, 72, 86, 105, 107, 109, 110, 119, 121, 127

[15] A. Solovyev, M. Baranowski, I. Briggs, C. Jacobsen, Z. Rakamarić, and G. Gopalakrishnan, "Rigorous estimation of floating-point round-off errors with symbolic taylor expansions," vol. 41, no. 1, 2018. $\rightarrow$ pages 2, 3, 105, 112

[16] L. Titolo, M. A. Feliú, M. Moscato, and C. A. Muñoz, "An abstract interpretation framework for the round-off error analysis of floating-point programs," in *Verification, Model Checking, and Abstract Interpretation* (I. Dillig and J. Palsberg, eds.), (Cham), pp. 516–537, Springer International Publishing, 2018. $\rightarrow$ pages 2, 3, 4, 20, 21, 40, 42

[17] V. Magron, G. Constantinides, and A. Donaldson, "Certified Roundoff Error Bounds Using Semidefinite Programming," *ACM Trans. Math. Softw.*, vol. 43, no. 4, 2017. $\rightarrow$ pages 2, 3, 20, 21, 56

[18] N. Damouche, M. Martel, and A. Chapoutot, "Improving the numerical accuracy of programs by automatic transformation," *STTT*, vol. 19, no. 4, pp. 427–448, 2017. $\rightarrow$ pages 2, 3

[19] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock, "Automatically Improving Accuracy for Floating Point Expressions," in *PLDI'15*, 2015. $\rightarrow$ pages 2, 5, 9, 23, 100, 103, 104, 120, 126

[20] X. Yi, L. Chen, X. Mao, and T. Ji, "Efficient Automated Repair of High Floating-Point Errors in Numerical Libraries," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, 2019. $\rightarrow$ pages 2, 5, 23, 100, 103, 104, 120, 126

[21] X. Wang, H. Wang, Z. Su, E. Tang, X. Chen, W. Shen, Z. Chen, L. Wang, X. Zhang, and X. Li, "Global Optimization of Numerical Programs via Prioritized Stochastic Algebraic Transformations," in *ICS'19*, 2019. $\rightarrow$ pages 2, 5, 23, 100, 103, 104, 120, 126

[22] E. Darulova, S. Sharma, and E. Horn, "Sound mixed-precision optimization with rewriting," in *ICCPS*, 2018. $\rightarrow$ pages 3, 5, 23, 24, 103, 105, 106, 107, 116, 117, 118, 126

[23] W.-F. Chiang, M. Baranowski, I. Briggs, A. Solovyev, G. Gopalakrishnan, and Z. Rakamarić, "Rigorous Floating-Point Mixed-Precision Tuning," in *POPL*, 2017. → pages 3, 5, 23, 24, 99, 103, 105, 106, 109, 117, 121

[24] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard, "Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels," in *OOPSLA*, 2014. → pages 3, 98, 127

[25] N. Damouche and M. Martel, "Mixed precision tuning with salsa," in *PECCS*, pp. 185–194, SciTePress, 2018. → pages 3, 5, 24, 99, 127

[26] J. P. Lim, M. Aanjaneya, J. Gustafson, and S. Nagarakatte, "An approach to generate correctly rounded math libraries for new floating point variants," *Proc. ACM Program. Lang.*, vol. 5, jan 2021. → pages 3, 99

[27] A. Miné, L. Mauborgne, X. Rival, J. Feret, P. Cousot, D. Kästner, S. Wilhelm, and C. Ferdinand, "Taking Static Analysis to the Next Level: Proving the Absence of Run-Time Errors and Data Races with Astrée," in *ERTS*, 2016. → pages 3, 17, 57

[28] F. Benz, A. Hildebrandt, and S. Hack, "A Dynamic Program Analysis to Find Floating-Point Accuracy Problems," in *Programming Language Design and Implementation (PLDI)*, 2012. → pages 3, 16, 56

[29] Z. Fu, Z. Bai, and Z. Su, "Automated backward error analysis for numerical code," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, (New York, NY, USA), pp. 639–654, Association for Computing Machinery, 2015. → pages 3, 56

[30] W. Chiang, G. Gopalakrishnan, Z. Rakamaric, and A. Solovyev, "Efficient search for inputs causing high floating-point errors," in *PPoPP'14*, 2014. → pages 3, 56

[31] T. Bao and X. Zhang, "On-the-fly detection of instability problems in floating-point program execution," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages& Applications*, OOPSLA '13, (New York, NY, USA), pp. 817–832, Association for Computing Machinery, 2013. → pages 3, 56

[32] D. Zou, R. Wang, Y. Xiong, L. Zhang, Z. Su, and H. Mei, "A Genetic Algorithm for Detecting Significant Floating-Point Inaccuracies," in *International Conference on Software Engineering (ICSE)*, 2015. → pages 3, 16, 56

[33] D. Zou, M. Zeng, Y. Xiong, Z. Fu, L. Zhang, and Z. Su, "Detecting floating-point errors via atomic conditions," *Proc. ACM Program. Lang.*, vol. 4, dec 2019. → pages 3, 16, 56

[34] Y. Xia, S. Guo, J. Hao, D. Liu, and J. Xu, "Error detection of arithmetic expressions," *The Journal of Supercomputing*, vol. 77, no. 6, pp. 5492–5509, 2021. → pages 3, 56

[35] A. Sanchez-Stern, P. Panchekha, S. Lerner, and Z. Tatlock, "Finding root causes of floating point error," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, (New York, NY, USA), pp. 256–269, Association for Computing Machinery, 2018. $\rightarrow$ pages 3, 16, 56

[36] A. Anta, R. Majumdar, I. Saha, and P. Tabuada, "Automatic verification of control system implementations," in *EMSOFT'10*, 2010. $\rightarrow$ pages 3, 105, 106

[37] E. Goubault and S. Putot, "Robustness analysis of finite precision implementations," in *Programming Languages and Systems* (C.-c. Shan, ed.), (Cham), pp. 50–57, Springer International Publishing, 2013. $\rightarrow$ pages 3, 22, 56

[38] D. Lohar, E. Darulova, S. Putot, and E. Goubault, "Discrete choice in the presence of numerical uncertainties," *IEEE TCAD*, Nov 2018. $\rightarrow$ pages 3, 22, 56, 109

[39] E. Goubault and S. Putot, "Static Analysis of Finite Precision Computations," in *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2011. $\rightarrow$ pages 4, 6, 20, 29, 30, 31, 32, 47, 56, 105, 110, 127

[40] A. R. Bradley, "Sat-based model checking without unrolling," in *Verification, Model Checking, and Abstract Interpretation, VMCAI*, pp. 70–87, 2011. $\rightarrow$ pages 5, 61

[41] I. Dillig, T. Dillig, B. Li, and K. L. McMillan, "Inductive invariant generation via abductive inference," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA*, pp. 443–456, 2013. $\rightarrow$ pages 5, 61, 79

[42] G. Fedyukovich, S. J. Kaufman, and R. Bodík, "Sampling invariants from frequency distributions," in *FMCAD (Formal Methods in Computer Aided Design)*, 2017. $\rightarrow$ pages 5, 61

[43] P. Garg, C. Löding, P. Madhusudan, and D. Neider, "ICE: A robust framework for learning invariants," in *Computer Aided Verification (CAV)*, 2014. $\rightarrow$ pages 5, 61, 78

[44] A. Adjé, S. Gaubert, and E. Goubault, "Coupling policy iteration with semi-definite relaxation to compute accurate numerical invariants in static analysis," *Logical Methods in Computer Science*, vol. 8, no. 1, 2012. $\rightarrow$ pages 5, 61, 73, 78

[45] T. M. Gawlitza and H. Seidl, "Numerical invariants through convex relaxation and max-strategy iteration," *Formal Methods Syst. Des.*, vol. 44, no. 2, pp. 101–148, 2014. $\rightarrow$ pages 5, 78

[46] A. Miné, J. Breck, and T. W. Reps, "An algorithm inspired by constraint solvers to infer inductive invariants in numeric programs," in *Programming Languages and Systems (ESOP)*, 2016. $\rightarrow$ pages 5, 31, 32, 61, 62, 63, 73

[47] P. Roux and P. Garoche, "Practical policy iterations - A practical use of policy iterations for static analysis: the quadratic case," *Formal Methods Syst. Des.*, vol. 46, no. 2, pp. 163–196, 2015. → pages 5, 8, 61, 62, 71, 72, 73, 78

[48] S. de Oliveira, S. Bensalem, and V. Prevosto, "Synthesizing invariants by solving solvable loops," in *Automated Technology for Verification and Analysis (ATVA)*, 2017. → pages 5, 8, 61, 62, 63, 72, 78

[49] E. Darulova, V. Kuncak, R. Majumdar, and I. Saha, "Synthesis of fixed-point programs," in *Proceedings of the Eleventh ACM International Conference on Embedded Software*, EMSOFT '13, IEEE Press, 2013. → pages 5, 15, 24, 103

[50] A. Izycheva, E. Darulova, and H. Seidl, "Synthesizing efficient low-precision kernels," in *ATVA*, 2019. → pages 5, 11

[51] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough, "Precimonious: Tuning Assistant for Floating-point Precision," in *SC*, 2013. → pages 5, 24, 100, 103, 106, 127

[52] N. Damouche and M. Martel, "Salsa: An automatic tool to improve the numerical accuracy of programs," in *Automated Formal Methods* (N. Shankar and B. Dutertre, eds.), vol. 5 of *Kalpa Publications in Computing*, pp. 63–76, EasyChair, 2018. → pages 5, 23, 127

[53] P. Prusaczyk, W. Kaczmarek, J. Panasiuk, and K. Besseghieur, "Integration of robotic arm and vision system with processing software using tcp/ip protocol in industrial sorting application," in *AIP Conference Proceedings*, vol. 2078, p. 020032, AIP Publishing LLC, 2019. → page 6

[54] X. Wan, W. Wang, J. Liu, and T. Tong, "Estimating the sample mean and standard deviation from the sample size, median, range and/or interquartile range," *BMC medical research methodology*, vol. 14, pp. 1–13, 2014. → page 6

[55] P. Heckbert, "Fourier transforms and the fast fourier transform (fft) algorithm," *Computer Graphics*, vol. 2, pp. 15–463, 1995. → page 6

[56] T. Yu, K. Song, P. Miao, G. Yang, H. Yang, and C. Chen, "Nighttime single image dehazing via pixel-wise alpha blending," *IEEE Access*, vol. 7, pp. 114619–114630, 2019. → page 6

[57] A. Das, I. Briggs, G. Gopalakrishnan, S. Krishnamoorthy, and P. Panchekha, "Scalable yet rigorous floating-point error analysis," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2020. → pages 6, 20, 21, 29, 30, 31, 32, 47, 56

[58] A. Anta and P. Tabuada, "To sample or not to sample: Self-triggered control for nonlinear systems," *IEEE Transactions on Automatic Control*, vol. 55, no. 9, pp. 2030–2042, 2010. → pages 8, 98

[59] Xilinx, "Vivado design suite." `https://www.xilinx.com/products/design-tools/vivado.html`, 2018. → pages 9, 25, 87

[60] A. Isychev and E. Darulova, "Scaling up roundoff analysis of functional data structure programs," in *Static Analysis* (M. V. Hermenegildo and J. F. Morales, eds.), (Cham), pp. 371–402, Springer Nature Switzerland, 2023. → page 11

[61] A. Izycheva, E. Darulova, and H. Seidl, "Counterexample- and simulation-guided floating-point loop invariant synthesis," in *Static Analysis* (D. Pichardie and M. Sighireanu, eds.), (Cham), pp. 156–177, Springer International Publishing, 2020. → page 11

[62] R. Rabe, A. Izycheva, and E. Darulova, "Regime inference for sound floating-point optimizations," *ACM Trans. Embed. Comput. Syst.*, vol. 20, sep 2021. → page 11

[63] Gustafson and Yonemoto, "Beating floating point at its own game: Posit arithmetic," *Supercomput. Front. Innov.: Int. J.*, vol. 4, no. 2, 2017. → pages 13, 130

[64] "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, 2019. → pages 13, 66, 85

[65] S. L. Harris and D. Harris, "5 - Digital Building Blocks," in *Digital Design and Computer Architecture* (S. L. Harris and D. Harris, eds.), pp. 236–297, Morgan Kaufmann, 2022. → page 15

[66] M. O. Lam, J. K. Hollingsworth, and G. Stewart, "Dynamic floating-point cancellation detection," *Parallel Computing*, vol. 39, no. 3, pp. 146–155, 2013. High-performance Infrastructure for Scalable Tools. → page 16

[67] R. Moore, *Interval Analysis*. Prentice-Hall, 1966. → pages 16, 20

[68] L. H. de Figueiredo and J. Stolfi, "Affine Arithmetic: Concepts and Applications," *Numerical Algorithms*, vol. 37, no. 1-4, 2004. → pages 17, 20

[69] E. Darulova and V. Kuncak, "Trustworthy numerical computation in scala," in *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, (New York, NY, USA), pp. 325–344, Association for Computing Machinery, 2011. → page 17

[70] A. Izycheva and E. Darulova, "On sound relative error bounds for floating-point arithmetic," in *Formal Methods in Computer Aided Design (FMCAD)*, 2017. → page 18

[71] A. Solovyev, C. Jacobsen, Z. Rakamaric, and G. Gopalakrishnan, "Rigorous Estimation of Floating-Point Round-off Errors with Symbolic Taylor Expansions," in *FM*, 2015. → pages 20, 29, 31, 56, 119

[72] S. Owre, J. M. Rushby, , and N. Shankar, "PVS: A prototype verification system," in *11th International Conference on Automated Deduction (CADE)* (D. Kapur, ed.), vol. 607 of *Lecture Notes in Artificial Intelligence*, (Saratoga, NY), pp. 748–752, Springer-Verlag, jun 1992. → page 22

[73] M. Moscato, L. Titolo, A. Dutle, and C. Muñoz, "Automatic Estimation of Verified Floating-Point Round-Off Errors via Static Analysis," in *SAFECOMP*, 2017. → pages 22, 31, 56

[74] R. Poli, W. Langdon, and N. Mcphee, *A Field Guide to Genetic Programming*. 01 2008. → page 23

[75] N. Fossati, D. Cattaneo, M. Chiari, S. Cherubin, and G. Agosta, "Automated Precision Tuning in Activity Classification Systems: A Case Study," in *Proceedings of the 11th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures / 9th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms*, PARMA-DITAM'2020, (New York, NY, USA), Association for Computing Machinery, 2020. → page 23

[76] Gurobi Optimization, LLC, "Gurobi Optimizer Reference Manual," 2023. → page 24

[77] H. Guo and C. Rubio-González, "Exploiting community structure for floating-point precision tuning," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018. → pages 24, 100, 127

[78] M. O. Lam, J. K. Hollingsworth, B. R. de Supinski, and M. P. Legendre, "Automatically adapting programs for mixed-precision floating-point computation," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, (New York, NY, USA), pp. 369–378, Association for Computing Machinery, 2013. → pages 24, 100, 127

[79] D. Cattaneo, M. Chiari, N. Fossati, S. Cherubin, and G. Agosta, "Architecture-aware precision tuning with multiple number representation systems," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pp. 673–678, 2021. → pages 24, 100

[80] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger, "An overview of the scala programming language," 2004. → page 24

[81] C. I. Byrnes and A. Isidori, "New results and examples in nonlinear feedback stabilization," *Systems & Control Letters*, vol. 12, no. 5, pp. 437–442, 1989. → page 24

[82] E. Darulova, S. Sharma, E. Horn, D.Lohar, H.Becker, E.Postan, F. Ritter, A.Izycheva, R. Monat, F. Nasir, R. Bastian, A.Volkova, R. Bankanal, R. Rabe, J. Bard, A.Gupta, "Daisy - open-source repository." $\rightarrow$ page 25

[83] P. Cousot and R. Cousot, "Basic concepts of abstract interpretation," in *Building the Information Society: IFIP 18th World Computer Congress Topical Sessions 22–27 August 2004 Toulouse, France*, pp. 359–366, Springer, 2004. $\rightarrow$ page 29

[84] N. Damouche, M. Martel, P. Panchekha, C. Qiu, A. Sanchez-Stern, and Z. Tatlock, "Toward a standard benchmark format and suite for floating-point analysis," in *NSV'16*, 2016. $\rightarrow$ pages 31, 45, 105, 119

[85] B. Hagedorn, L. Stoltzfus, M. Steuwer, S. Gorlatch, and C. Dubach, "High performance stencil code generation with lift," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, (New York, NY, USA), pp. 100–112, Association for Computing Machinery, 2018. $\rightarrow$ page 34

[86] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org. $\rightarrow$ page 34

[87] T. Cheng and X. Rival, "An Abstract Domain to Infer Types over Zones in Spreadsheets," in *SAS'12 - 19th International Static Analysis Symposium* (A. Miné and D. Schmidt, eds.), vol. 7460 of *Lecture notes in computer science*, (Deauville, France), pp. 94–110, Springer, Sept. 2012. $\rightarrow$ page 44

[88] N. Halbwachs and M. Péron, "Discovering properties about arrays in simple programs," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, (New York, NY, USA), pp. 339–348, Association for Computing Machinery, 2008. $\rightarrow$ page 44

[89] P. Cousot, R. Cousot, and F. Logozzo, "A parametric segmentation functor for fully automatic and scalable array content analysis," in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, (New York, NY, USA), pp. 105–118, Association for Computing Machinery, 2011. $\rightarrow$ page 44

[90] F. De Dinechin, C. Q. Lauter, and G. Melquiond, "Assisted Verification of Elementary Functions Using Gappa," in *ACM Symposium on Applied Computing*, 2006. $\rightarrow$ page 56

[91] I. Laguna, "Fpchecker: Detecting floating-point exceptions in gpu applications," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ASE '19, pp. 1126–1129, IEEE Press, 2020. → page 56

[92] X. Yi, L. Chen, X. Mao, and T. Ji, "Efficient global search for inputs triggering high floating-point inaccuracies," in *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 11–20, IEEE, 2017. → page 56

[93] D. Liew, D. Schemmel, C. Cadar, A. F. Donaldson, R. Zähl, and K. Wehrle, "Floating-Point Symbolic Execution: A Case Study in N-Version Programming," in *ASE*, 2017. → page 56

[94] H. Guo and C. Rubio-González, "Efficient Generation of Error-Inducing Floating-Point Inputs via Symbolic Execution," in *International Conference on Software Engineering (ICSE)*, 2020. → page 56

[95] E. T. Barr, T. Vo, V. Le, and Z. Su, "Automatic detection of floating-point exceptions," in *Principles of Programming Languages (POPL)*, 2013. → page 56

[96] D. Lohar, C. Jeangoudoux, J. Sobel, E. Darulova, and M. Christakis, "A two-phase approach for conditional floating-point verification," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2021. → page 56

[97] R. Abbasi, J. Schiffl, E. Darulova, M. Ulbrich, and W. Ahrendt, "Deductive verification of floating-point java programs in key," in *Tools and Algorithms for the Construction and Analysis of Systems* (J. F. Groote and K. G. Larsen, eds.), (Cham), pp. 242–261, Springer International Publishing, 2021. → page 57

[98] G. Gange, Z. Ma, J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey, "A fresh look at zones and octagons," *ACM Trans. Program. Lang. Syst.*, vol. 43, sep 2021. → page 57

[99] B. Jeannet and A. Miné, "Apron: A library of numerical abstract domains for static analysis," in *Computer Aided Verification, 21st International Conference, CAV*, pp. 661–667, 2009. → pages 57, 73, 78

[100] G. Singh, M. Püschel, and M. T. Vechev, "Fast polyhedra abstract domain," in *Principles of Programming Languages (POPL)*, 2017. → page 57

[101] M. Journault and A. Miné, "Inferring functional properties of matrix manipulating programs by abstract interpretation," *Formal Methods in System Design*, vol. 53, no. 2, pp. 221–258, 2018. → page 57

[102] S. Kumar, A. Sanyal, R. Venkatesh, and P. Shah, "Property checking array programs using loop shrinking," in *Tools and Algorithms for the Construction and Analysis of Systems* (D. Beyer and M. Huisman, eds.), (Cham), pp. 213–231, Springer International Publishing, 2018. → page 57

[103] L. Bin, T. Zhenhao, and Z. Jianhua, "Invariant synthesis for programs manipulating arrays with unbounded data," in *Proceedings of the 7th Asia-Pacific Symposium on Internetware*, Internetware '15, (New York, NY, USA), pp. 195–198, Association for Computing Machinery, 2015. → page 57

[104] G. Fedyukovich and R. Bodík, "Accelerating syntax-guided invariant synthesis," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2018. → pages 57, 79

[105] G. Fedyukovich, S. Prabhu, K. Madhukar, and A. Gupta, "Quantified invariants via syntax-guided synthesis," in *Computer Aided Verification* (I. Dillig and S. Tasiran, eds.), (Cham), pp. 259–277, Springer International Publishing, 2019. → pages 57, 79

[106] J. Dohrau, A. J. Summers, C. Urban, S. Münger, and P. Müller, "Permission inference for array programs," in *Computer Aided Verification* (H. Chockler and G. Weissenbacher, eds.), (Cham), pp. 55–74, Springer International Publishing, 2018. → page 57

[107] A. D. Falkoff and K. E. Iverson, "The evolution of apl," *SIGPLAN Not.*, vol. 13, pp. 47–57, aug 1978. → page 57

[108] APL Wiki, "Main Page — APL Wiki,," 2022. [Online; accessed 9-June-2023]. → page 57

[109] C. Hoekstra, "Combinatory logic and combinators in array languages," in *Proceedings of the 8th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*, ARRAY 2022, (New York, NY, USA), pp. 46–57, Association for Computing Machinery, 2022. → page 57

[110] M. Lochbaum, "BQN: Big Question Notation Lanugage," 2022. → page 57

[111] S. Gulwani and A. Tiwari, "Computing procedure summaries for interprocedural analysis," in *Programming Languages and Systems: 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24-April 1, 2007. Proceedings 16*, pp. 253–267, Springer, 2007. → page 57

[112] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, pp. 451–490, oct 1991. → page 58

[113] A. Sukumaran-Rajam and P. Clauss, "The polyhedral model of nonlinear loops," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 4, pp. 1–27, 2015. → page 58

[114] B. Mariano, Y. Chen, Y. Feng, G. Durrett, and I. Dillig, "Automated transpilation of imperative to functional code using neural-guided program synthesis," *Proc. ACM Program. Lang.*, vol. 6, apr 2022. → page 58

[115] A. Darte, "On the complexity of loop fusion," in *1999 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.PR00425)*, pp. 149–157, 1999. → page 58

[116] Z. Lin and C. Dubach, "From functional to imperative: Combining destination-passing style and views," in *Proceedings of the 8th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*, ARRAY 2022, (New York, NY, USA), pp. 25–36, Association for Computing Machinery, 2022. → page 59

[117] M. v. d. Oever, L. E. Grimley, and R. M. Veras, "Using Q-Learning to Select the Best among Functionally Equivalent Implementations," in *Proceedings of the 8th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*, ARRAY 2022, (New York, NY, USA), pp. 37–45, Association for Computing Machinery, 2022. → page 59

[118] A. Shaikhha, A. Fitzgibbon, S. Peyton Jones, and D. Vytiniotis, "Destination-passing style for efficient memory management," in *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing*, FHPC 2017, (New York, NY, USA), pp. 12–23, Association for Computing Machinery, 2017. → page 59

[119] J. Ragan-Kelley, A. Adams, D. Sharlet, C. Barnes, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand, "Halide: Decoupling algorithms from schedules for high-performance image processing," *Commun. ACM*, vol. 61, pp. 106–115, dec 2017. → page 59

[120] C. Schlaak, T.-H. Juang, and C. Dubach, "Optimizing Data Reshaping Operations in Functional IRs for High-Level Synthesis," in *Proceedings of the 23rd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES 2022, (New York, NY, USA), pp. 61–72, Association for Computing Machinery, 2022. → page 59

[121] V. Kuncak and J. Hamza, "Stainless verification system tutorial," in *2021 Formal Methods in Computer Aided Design (FMCAD)*, pp. 2–7, IEEE, 2021. → page 59

[122] M. Bucev and V. Kunčak, "Formally Verified Quite OK Image Format," in *Proceedings of the 22nd Conference on Formal Methods in Computer-Aided Design–FMCAD 2022*, no. CONF, pp. 343–348, TU Wien, 2022. → page 59

[123] O. Padon, N. Immerman, S. Shoham, A. Karbyshev, and M. Sagiv, "Decidability of inferring inductive invariants," *ACM SIGPLAN Notices*, vol. 51, no. 1, pp. 217–231, 2016. → page 61

[124] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008. $\rightarrow$ page 61

[125] C. W. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *Computer Aided Verification - 23rd International Conference, CAV*, pp. 171–177, 2011. $\rightarrow$ page 61

[126] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, and A. V. Nori, "Verification as learning geometric concepts," in *Static Analysis Symposium (SAS)*, 2013. $\rightarrow$ pages 61, 79

[127] R. Sharma and A. Aiken, "From invariant checking to invariant inference using randomized search," in *Computer Aided Verification (CAV)*, 2014. $\rightarrow$ pages 61, 78

[128] H. Zhu, S. Magill, and S. Jagannathan, "A data-driven CHC solver," in *Programming Language Design and Implementation (PLDI)*, 2018. $\rightarrow$ pages 61, 78

[129] D. Jovanovic and L. M. de Moura, "Solving non-linear arithmetic," in *Automated Reasoning - 6th International Joint Conference, IJCAR*, 2012. $\rightarrow$ pages 62, 70, 71

[130] N. Moshtagh, *Minimum Volume Enclosing Ellipsoid*, 2020 (retrieved May 21, 2020). $\rightarrow$ page 69

[131] A. Magnani, S. Lall, and S. Boyd, "Tractable fitting with convex polynomials via sum-of-squares," in *Proceedings of the 44th IEEE Conference on Decision and Control*, 2005. $\rightarrow$ page 69

[132] M. Brain, F. Schanda, and Y. Sun, "Building better bit-blasting for floating-point problems," in *Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, pp. 79–98, 2019. $\rightarrow$ page 71

[133] K. J. Astrom and R. M. Murray, *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, 2008. $\rightarrow$ page 73

[134] P. Roux and P. Garoche, "Integrating policy iterations in abstract interpreters," in *Automated Technology for Verification and Analysis (ATVA)*, 2013. $\rightarrow$ page 73

[135] B. J. Gal Lalire, M. Argoud, "A web interface to the interproc analyzer." $\rightarrow$ page 73

[136] R. Bagnara, P. M. Hill, and E. Zaffanella, "The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems," *Science of Computer Programming*, vol. 72, no. 1, pp. 3 – 21, 2008. $\rightarrow$ pages 73, 78

[137] P. Cousot and C. Radhia, "Static determination of dynamic properties of programs," in *ISOP*, pp. 106–130, 1976. $\rightarrow$ page 78

[138] A. Miné, “The octagon abstract domain,” *High. Order Symb. Comput.*, vol. 19, no. 1, pp. 31–100, 2006. → page 78

[139] G. Singh, M. Püschel, and M. Vechev, “A practical construction for decomposing numerical abstract domains,” *Proc. ACM Program. Lang.*, vol. 2, no. POPL, 2017. → page 78

[140] J. Feret, “Static analysis of digital filters,” in *Programming Languages and Systems (ESOP)*, 2004. → page 78

[141] M. Oulamara and A. J. Venet, “Abstract interpretation with higher-dimensional ellipsoids and conic extrapolation,” in *Computer Aided Verification (CAV)*, 2015. → page 78

[142] R. Bagnara, E. Rodríguez-Carbonell, and E. Zaffanella, “Generation of basic semi-algebraic invariants using convex polyhedra,” in *Static Analysis*, 2005. → page 78

[143] S. Sankaranarayanan, H. B. Sipma, and Z. Manna, “Non-linear loop invariant generation using gröbner bases,” in *Principles of Programming Languages*, POPL, 2004. → page 78

[144] A. Gupta and A. Rybalchenko, “InvGen: An Efficient Invariant Generator,” in *Computer Aided Verification (CAV)*, 2009. → page 79

[145] T. Nguyen, T. Antonopoulos, A. Ruef, and M. Hicks, “Counterexample-guided approach to finding numerical invariants,” in *Foundations of Software Engineering (ESEC/FSE)*, 2017. → page 79

[146] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. V. Nori, “A data driven approach for algebraic loop invariants,” in *Programming Languages and Systems (ESOP)*, 2013. → page 79

[147] S. Chakraborty, A. Gupta, and D. Unadkat, “Full-program induction: verifying array programs sans loop invariants,” *International Journal on Software Tools for Technology Transfer*, vol. 24, no. 5, pp. 843–888, 2022. → page 79

[148] X. Allamigeon, S. Gaubert, E. Goubault, S. Putot, and N. Stott, “A fast method to compute disjunctive quadratic invariants of numerical programs,” *ACM Trans. Embedded Comput. Syst.*, vol. 16, no. 5s, pp. 166:1–166:19, 2017. → page 79

[149] Z. Kincaid, J. Cyphert, J. Breck, and T. W. Reps, “Non-linear reasoning for invariant synthesis,” *Proc. ACM Program. Lang.*, vol. 2, no. POPL, pp. 54:1–54:33, 2018. → page 79

[150] L. Kovács, “Reasoning Algebraically About P-Solvable Loops,” in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008. → page 79

[151] J. Krämer, L. Blatter, E. Darulova, and M. Ulbrich, "Inferring interval-valued floating-point preconditions," in *Tools and Algorithms for the Construction and Analysis of Systems* (D. Fisman and G. Rosu, eds.), (Cham), pp. 303–321, Springer International Publishing, 2022. → page 79

[152] M. N. Seghir and D. Kroening, "Counterexample-guided precondition inference," in *Programming Languages and Systems* (M. Felleisen and P. Gardner, eds.), (Berlin, Heidelberg), pp. 451–471, Springer Berlin Heidelberg, 2013. → page 79

[153] Y. Moy, "Sufficient preconditions for modular assertion checking," in *Verification, Model Checking, and Abstract Interpretation* (F. Logozzo, D. A. Peled, and L. D. Zuck, eds.), (Berlin, Heidelberg), pp. 188–202, Springer Berlin Heidelberg, 2008. → page 79

[154] K. Claessen, N. Smallbone, and J. Hughes, "QuickSpec: Guessing Formal Specifications Using Testing," in *Tests and Proofs* (G. Fraser and A. Gargantini, eds.), (Berlin, Heidelberg), pp. 6–21, Springer Berlin Heidelberg, 2010. → page 79

[155] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1, pp. 35–45, 2007. Special issue on Experimental Software and Toolkits. → page 79

[156] Q. Xu, T. Mytkowicz, and N. S. Kim, "Approximate computing: A survey," *IEEE Design Test*, vol. 33, pp. 8–22, Feb 2016. → pages 85, 98

[157] H.-J. Boehm, "Towards an API for the Real Numbers," in *Programming Language Design and Implementation*, PLDI 2020, (New York, NY, USA), pp. 562–576, Association for Computing Machinery, 2020. → page 85

[158] R. D. Wolfinger and X. Lin, "Two taylor-series approximation methods for nonlinear mixed models," *Computational Statistics & Data Analysis*, vol. 25, no. 4, pp. 465–490, 1997. → page 85

[159] M. Hernández, "Chebyshev's approximation algorithms and applications," *Computers & Mathematics with Applications*, vol. 41, no. 3-4, pp. 433–445, 2001. → page 85

[160] T. Y. Chow, "What is a closed-form number?," *The American mathematical monthly*, vol. 106, no. 5, pp. 440–448, 1999. → pages 85, 87

[161] O. Kupriianova and C. Lauter, "Metalibm: A mathematical functions code generator," in *ICMS*, 2014. → pages 86, 87

[162] N.Briesbarre and S.Chevillard, "Efficient polynomial L-approximations," in *ARITH*, 2007. → page 87

[163] O. Kupriianova and C. Lauter, "A domain splitting algorithm for the mathematical functions code generator," in *Asilomar*, 2014. → pages 87, 101, 127

[164] N. Brunie, F. d. Dinechin, O. Kupriianova, and C. Lauter, "Code generators for mathematical functions," in *ARITH*, 2015. → page 87

[165] "Project CORPIN." https://www-sop.inria.fr/corpin/logiciels/ALIAS/Benches/. → pages 88, 94

[166] D. U. Lee, A. A. Gaffar, R. C. Cheung, O. Mencer, W. Luk, and G. A. Constantinides, "Accuracy-guaranteed bit-width optimization," *TCAD*, vol. 25, pp. 1990–2000, Oct. 2006. → page 91

[167] "Python sklearn - multi-layer perceptron regressor." `https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPRegressor.html`, 2019. → page 91

[168] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran, "AxBench: A Multiplatform Benchmark Suite for Approximate Computing," *IEEE Design Test*, vol. 34, no. 2, 2017. → page 94

[169] S. Cherubin and G. Agosta, "Tools for reduced precision computation: A survey," *ACM Comput. Surv.*, vol. 53, apr 2020. → page 98

[170] S. Lee, L. K. John, and A. Gerstlauer, "High-level synthesis of approximate hardware under joint precision and voltage scaling," in *DATE*, pp. 187–192, 2017. → page 98

[171] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *ESEC/FSE*, 2011. → page 98

[172] L. Renganarayana, V. Srinivasan, R. Nair, and D. Prener, "Programming with relaxed synchronization," in *RACES*, pp. 41–50, 2012. → page 98

[173] R. Ballester-Ripoll, P. Lindstrom, and R. Pajarola, "TTHRESH: Tensor Compression for Multidimensional Visual Data," *IEEE Transactions on Visualization and Computer Graphics*, vol. 26, no. 9, pp. 2891–2903, 2020. → page 98

[174] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 6869–6898, 2017. → page 98

[175] D. Lin, S. Talathi, and S. Annapureddy, "Fixed point quantization of deep convolutional networks," in *International conference on machine learning*, pp. 2849–2858, PMLR, 2016. → page 98

[176] E. Darulova and A. Volkova, “Sound approximation of programs with elementary functions,” in *CAV*, 2018. → pages 98, 127, 128

[177] I. Briggs and P. Panchekha, “Choosing mathematical function implementations for speed and accuracy,” in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, (New York, NY, USA), pp. 522–535, Association for Computing Machinery, 2022. → pages 98, 101

[178] F. de Dinechin and B. Pasca, “Designing custom arithmetic data paths with FloPoCo,” *IEEE Design & Test of Computers*, vol. 28, pp. 18–27, July 2011. → page 98

[179] J. Ansel, Y. L. Wong, C. Chan, M. Olszewski, A. Edelman, and S. Amarasinghe, “Language and compiler support for auto-tuning variable-accuracy algorithms,” in *CGO*, 2011. → page 99

[180] W. Baek and T. M. Chilimbi, “Green: A framework for supporting energy-conscious programming using controlled approximation,” in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’10, (New York, NY, USA), pp. 198–209, Association for Computing Machinery, 2010. → page 99

[181] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, “Neural acceleration for general-purpose approximate programs,” in *MICRO*, 2012. → page 99

[182] E. Schkufza, R. Sharma, and A. Aiken, “Stochastic Optimization of Floating-Point Programs with Tunable Precision,” in *PLDI’14*, 2014. → pages 99, 100, 127

[183] D. R.Braojos, G.Ansaloni, “A methodology for embedded classification of heartbeats using random projections,” in *DATE*, EPFL, 2013. → page 99

[184] W. Lee, R. Sharma, and A. Aiken, “On automatically proving the correctness of math.h implementations,” in *POPL*, 2018. → page 99

[185] V. Innocente and P. Zimmermann, “Accuracy of mathematical functions in single, double, extended double and quadruple precision,” 2023. → page 99

[186] J. P. Lim and S. Nagarakatte, “One polynomial approximation to produce correctly rounded results of an elementary function for multiple representations and rounding modes,” *Proc. ACM Program. Lang.*, vol. 6, jan 2022. → page 99

[187] C. Daramy-Loirat, D. Defour, F. de Dinechin, M. Gallet, N. Gast, C. Lauter, and J.-M. Muller, “CR-LIBM A library of correctly rounded elementary functions in double-precision,” research report, LIP,, Dec. 2006. → page 99

[188] R. Bodik and B. Jobstmann, “Algorithmic program synthesis: introduction,” *STTT*, vol. 15, no. 5, pp. 397–411, 2013. → page 99

[189] E. Kneuss, I. Kuraj, V. Kuncak, and P. Suter, "Synthesis modulo recursive functions," in *OOPSLA*, 2013. → page 99

[190] R. Alur, A. Radhakrishna, and A. Udupa, "Scaling enumerative program synthesis via divide and conquer," in *TACAS*, pp. 319–336, Springer, 2017. → page 99

[191] L. D'Antoni, R. Samanta, and R. Singh, "Qlose: Program repair with quantitative objectives," in *CAV*, pp. 383–401, Springer, 2016. → page 99

[192] C. Loncaric, E. Torlak, and M. D. Ernst, "Fast synthesis of fast collections," *ACM SIGPLAN Notices*, vol. 51, no. 6, pp. 355–368, 2016. → page 99

[193] A. V. Nori, S. Ozair, S. K. Rajamani, and D. Vijaykeerthy, "Efficient synthesis of probabilistic programs," in *PLDI*, ACM, 2015. → page 99

[194] Y. Feng, R. Martins, Y. Wang, I. Dillig, and T. W. Reps, "Component-based synthesis for complex apis," in *POPL*, 2017. → page 99

[195] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," in *POPL*, 2011. → page 99

[196] D. Neider, S. Saha, and P. Madhusudan, "Compositional synthesis of piece-wise functions by learning classifiers," *ACM Trans. Comput. Logic*, vol. 19, pp. 10:1–10:23, May 2018. → page 99

[197] B. Saiki, O. Flatt, C. Nandi, P. Panchekha, and Z. Tatlock, "Combining precision tuning and rewriting," in *2021 IEEE 28th Symposium on Computer Arithmetic (ARITH)*, pp. 1–8, 2021. → pages 100, 126, 127

[198] J. Bornholt, E. Torlak, D. Grossman, and L. Ceze, "Optimizing Synthesis with Metasketches," in *POPL*, 2016. → page 100

[199] R. Wang, D. Zou, X. He, Y. Xiong, L. Zhang, and G. Huang, "Detecting and fixing precision-specific operations for measuring floating-point errors," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, (New York, NY, USA), pp. 619–630, Association for Computing Machinery, 2016. → page 100

[200] H. Becker, P. Panchekha, E. Darulova, and Z. Tatlock, "Combining tools for optimization and analysis of floating-point computations," in *Formal Methods* (K. Havelund, J. Peleska, B. Roscoe, and E. de Vink, eds.), (Cham), pp. 355–363, Springer International Publishing, 2018. → pages 100, 120

[201] E. Darulova and V. Kuncak, "Certifying solutions for numerical constraints," in *Runtime Verification* (S. Qadeer and S. Tasiran, eds.), (Berlin, Heidelberg), pp. 277–291, Springer Berlin Heidelberg, 2013. → page 100

[202] R. Green, "Even faster math functions," 2020. → page 101

[203] "GCC libquadmath," 2020. → pages 107, 121

[204] D. Whitley, "The genitor algorithm and selection pressure: Why rank-based allocation of reproductive trials is best," in *Proceedings of the Third International Conference on Genetic Algorithms*, 1989. → page 115

[205] C. Nandi, M. Willsey, A. Zhu, Y. R. Wang, B. Saiki, A. Anderson, A. Schulz, D. Grossman, and Z. Tatlock, "Rewrite rule inference using equality saturation," *Proc. ACM Program. Lang.*, vol. 5, oct 2021. → page 126

[206] A. Miné, "Abstract domains for bit-level machine integer and floating-point operations," in *ATx'12/WInG'12: Joint Proceedings of the Workshops on Automated Theory eXploration and on Invariant Generation*, 2012. → page 127

[207] W. Lee, R. Sharma, and A. Aiken, "Verifying bit-manipulations of floating-point," in *PLDI'16*, 2016. → page 127

[208] E. Goubault, S. Putot, P. Baufreton, and J. Gassino, "Static analysis of the accuracy in control systems: Principles and experiments," in *Formal Methods for Industrial Critical Systems*, 2008. → page 127

[209] H. Menon, M. Lam, D. Osei-Kuffuor, M. Schordan, S. Lloyd, K. Mohror, and J. Hittinger, "Adapt: Algorithmic differentiation applied to floating-point precision tuning," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, 2018. → page 127

[210] K. Seetharam, L. C. T. Keh, R. Nathan, and D. J. Sorin, "Applying reduced precision arithmetic to detect errors in floating point multiplication," in *2013 IEEE 19th Pacific Rim International Symposium on Dependable Computing*, 2013. → page 127

[211] P. Kotipalli, R. Singh, P. Wood, I. Laguna, and S. Bagchi, "Ampt-ga: Automatic mixed precision floating point tuning for gpu applications," in *ICS'19*, 2019. → page 127

[212] S. Graillat, F. Jézéquel, R. Picot, F. Févotte, and B. Lathuilière, "Auto-tuning for floating-point precision with discrete stochastic arithmetic," *Journal of Computational Science*, vol. 36, 2019. → page 127

[213] M. O. Lam and J. K. Hollingsworth, "Fine-grained floating-point precision analysis," *The International Journal of High Performance Computing Applications*, vol. 32, no. 2, 2018. → page 127

[214] E. Misback, C. Chan, B. Saiki, E. Jun, Z. Tatlock, and P. Panchekha, "Odyssey: An interactive workbench for expert-driven floating-point expression rewriting," 2023. → pages 127, 130

[215] G. Tagliavini, S. Mach, D. Rossi, A. Marongiu, and L. Benin, “A transprecision floating-point platform for ultra-low power computing,” in *DATE'18*, IEEE, 2018. → page 130

[216] NVIDIA, “Tensorfloat-32 in the a100 gpu accelerates ai training.” https://blogs.nvidia.com/blog/2020/ 05/14/tensoroat-32-precision-format/, 2020. → page 130

[217] B. Darvish Rouhani, D. Lo, R. Zhao, M. Liu, J. Fowers, K. Ovtcharov, A. Vinogradsky, S. Massengill, L. Yang, R. Bittner, A. Forin, H. Zhu, T. Na, P. Patel, S. Che, L. Chand Koppaka, X. SONG, S. Som, K. Das, S. T, S. Reinhardt, S. Lanka, E. Chung, and D. Burger, “Pushing the limits of narrow precision inferencing at cloud scale with microsoft floating point,” in *Advances in Neural Information Processing Systems* (H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, eds.), vol. 33, pp. 10271–10281, Curran Associates, Inc., 2020. → page 130

[218] P. Behnam and M. Bojnordi, “Posit: A Potential Replacement for IEEE 754,” 2020. → page 130

[219] G. Gopalakrishnan, I. Laguna, A. Li, P. Panchekha, C. Rubio-González, and Z. Tatlock, “Guarding numerics amidst rising heterogeneity,” in *2021 IEEE/ACM 5th International Workshop on Software Correctness for HPC Applications (Correctness)*, pp. 9–15, 2021. → page 130

A. Supplementary Material

A.1. Benchmarks with Loops over Data Structures

We provide the benchmark set we collected for the experimental evaluation of DS2L in section 3.5. We only provide the *AllSame* variation of the specification, where all data structure elements have the same input range, other configurations are available upon request (we will publish all variations open-source once the double-blind review phase of our paper submission is over). The data structure size in the `require` clause corresponds to large benchmarks, small and medium sizes are shown in the comments.

A.1.1. Benchmarks for DS2L

```
import daisy.lang._
import Real._
import daisy.lang.Vector._
object ds2lBenchmarks {
// Vector Benchmarks
def avg(x: Vector): Real = {
 require(x >= -62.54 && x <= 15.02 && x.size(10000)) // small 100; medium 1000
  val n: Real = x.length()
  val z = x.fold(0.0)((acc: Real, i: Real) => acc + i)
    z / n
}
def variance(x: Vector): Real = {
 require(x >= -252.68 && x <= 72.42 && x.size(10000))  // small 100; medium 1000

  val n: Real = x.length()
  val y = x.fold(0.0)((acc: Real, i: Real) => acc + i)
  val avg = y / n
  val z = x.fold(0.0)((acc: Real, i: Real) => acc + pow(i - avg, 2))
  z / n
}
def stdDeviation(x: Vector): Real = {
 require(x >= -160.06 && x <= 360.98 && x.size(10000))  // small 100; medium 1000
  val n: Real = x.length()
  val y = x.fold(0.0)((acc: Real, i: Real) => acc + i)
  val avg = y / n
  val z = x.fold(0.0)((acc: Real, i: Real) => {
     acc + pow((i - avg), 2)
  })
  sqrt(z / n)
}
def roux(x: Vector): Real = {
 require(x >= -58.25 && x <= 61.32 && x.size(10000))  // small 100; medium 1000
  x.fold(0.0)((y: Real, i: Real) => {1.5 * i - 0.7 * y})
}
def goubault(x:Vector, y: Real): Real = {
 require(54.86 <= y && y <= 359.03
  && x >= -270.01 && x <= 385.38 && x.size(10000))  // small 100; medium 1000
  x.fold(y)((acc: Real, xi: Real) => {0.75 * xi - 0.125 * acc})
}
def harmonic(x: Vector, y: Vector): Vector = {
 require(x >= -5.32 && x <= 725.6 && x.size(10000)  // small 100; medium 1000
  && y >= -432.12 && y <= 78.94 && y.size(10000))
  //x1 := x1 + 0.01 * x2
  val x1: Real = y.fold(x.head)((acc: Real, xi: Real) => {acc + 0.01* xi})
  //x2 := -0.01 * x1 + 0.99 * x2
  val x2: Real = x.fold(y.head)((acc: Real, xi: Real) => {-0.01 * xi + 0.99 * acc})
  Vector(List(x1, x2))
}
def nonlin1(x: Vector, y: Vector): Vector = {
 require(x >= 0.0 && x <= 1.0 && x.size(10000)  // small 100; medium 1000
  && y >= 0.0 && y <= 1.0 && y.size(10000))
  //x := x + 0.01 * (-2*x - 3*y + x*x)
  val x1: Real = y.fold(x.head)((acc: Real, yi: Real) => {acc + 0.01 * (-2*acc - 3*yi + acc*acc)})
  //y := y + 0.01 * (x + y)
  val y1: Real = x.fold(y.head)((acc: Real, xi: Real) => {acc + 0.01 * (xi + acc)})
  Vector(List(x1, y1))
}
def nonlin2(x: Vector, y: Vector): Vector = {
 require(x >= 0.0 && x <= 1.0 && x.size(10000)  // small 100; medium 1000
  && y >= 0.0 && y <= 1.0 && y.size(10000))
  //x := x + 0.01 * (-x + 2*x*x + y*y)
  val x1: Real = y.fold(x.head)((acc: Real, yi: Real) => {acc + 0.01 * (-acc + 2*acc*acc + yi*yi)})
  //y := y + 0.01 * (-y + y*y)
  val y1: Real = x.fold(y.head)((acc: Real, xi: Real) => {acc + 0.01 * (-acc + acc*acc)})
  Vector(List(x1, y1))
}
def nonlin3(x: Vector, y: Vector): Vector = {
 require(x >= 0.0 && x <= 1.0 && x.size(10000)  // small 100; medium 1000
  && y >= 0.0 && y <= 1.0 && y.size(10000))
  // x := x + 0.01 * (-x + y*y)
  val x1: Real = y.fold(x.head)((acc: Real, yi: Real) => {acc + 0.01 * (-acc + yi*yi)})
  // y := y + 0.01 * (-2.0*y + 3.0*x*x)
  val y1: Real = x.fold(y.head)((acc: Real, xi: Real) => {acc + 0.01 * (-2.0*acc + 3.0*xi*xi)})
  Vector(List(x1, y1))
}
def heat1d(ax: Vector): Real = {
 require(ax >= 1.0 && ax <= 2.0 && ax.size(513)) // small 33; medium 65

  if (ax.length() <= 1) {
      ax.head
  } else {
    val coef = Vector(List(0.25, 0.5, 0.25))
    val updCoefs: Vector = ax.slideReduce(3,1)(v =>  (coef*v).sum())
    heat1d(updCoefs)
  }
}
def fftvector(vr: Vector, vi: Vector): Vector = {
 require(vr >= 68.9 && vr <= 160.43 && vr.size(512) // small 4, medium 128
  && vi >= -133.21 && vi <= 723.11 && vi.size(512))
  /* v: (real part of signal / Fourier coeff., imaginary part of signal / Fourier coeff. ) */
  if (vr.length() == 1)
      Vector(List(vr.head, vi.head))
  else {
    val scalar: Real = 1
    val Pi: Real = 3.1415926
    val n: Int = vr.length()
    val direction: Vector = Vector(List(0.0, -2.0))
    val evens: Vector = fftvector(vr.everyNth(2, 0), vi.everyNth(2, 0))
    val odds: Vector = fftvector(vr.everyNth(2, 1), vi.everyNth(2, 1))
    val resleft: Vector = evens.enumSlideFlatMap(2)((k, xv) => {
      val base: Vector = xv / scalar
      val oddV: Vector = odds.slice(2 * k, 2 * k + 1)
      val expV: Vector = (direction.*(Pi * k / n)).exp()
      val offset: Vector = (oddV x expV) / scalar
        base + offset
      })
    val resright: Vector = evens.enumSlideFlatMap(2)((k, xv) => {
      val base: Vector = xv / scalar
      val oddV: Vector = odds.slice(2 * k, 2 * k + 1)
      val expV: Vector = (direction.*(Pi * k / n)).exp()
      val offset: Vector = (oddV x expV) / scalar
        base - offset
      })
    resleft ++ resright
  }
}
// Matrix Benchmarks
def pendulum(t: Vector, w: Vector): Vector = {
require(t >= -2.0 && t <= 2.0 && t.size(10000) // small 10; medium 1000
  && w >= -5.0 && w <= 5.0 && w.size(10000))
      val h: Real = 0.01
      val L: Real = 2.0
      val g: Real = 9.80665
      val iter = Vector.zip(t,w) // into a Matrix
      val init = Vector(List(t.head, w.head))
      iter.fold(init)((acc, x) => {
    val kt = acc.at(1)
    val kw = -g/L * sin(acc.head)
    val v = Vector(List(kt,kw))
      acc + v*h
      })
    }
def alphaBlending(b: Matrix, c: Matrix, alpha: Real): Matrix = {
 require(0.0 <= alpha && alpha <= 1.0
  && b >= 223.35 && b <= 530.05 && b.size(500,500) // small (10,10); medium (100,100)
  && c >= -253.26 && c <= -108.41 && c.size(500,500)
  )
    b * alpha + c * (1 - alpha)
}

def fftmatrix(m: Matrix): Matrix = {
 require(m >= -326.68 && m <= 677.57 && m.size(512,2)) // small (4,2), medium (128,2)
 /* m: (real part of signal / Fourier coeff., imaginary part of signal / Fourier coeff. ) */
  if (m.numRows() == 1)
       m
  else {
    val scalar: Real = 1
    val Pi: Real = 3.1415926
    val n: Int = m.numRows()   /* signal length, has to be power of 2 */
    val direction: Vector = Vector(List(0.0, -2.0))
    val evens: Matrix = fftmatrix(m.everyNth(2, 0))
    val odds: Matrix = fftmatrix(m.everyNth(2, 1))
    val resleft: Matrix = evens.enumRowsMap((k:Int, x:Vector) => {
      val base: Vector = x / scalar
      val offset: Vector = (direction.*(Pi * k / n)).exp() x odds.row(k) / scalar
        base + offset
      })
    val resright: Matrix = evens.enumRowsMap((k:Int, x:Vector)  => {
      val base: Vector = x / scalar
      val offset: Vector = (direction.*(Pi * k / n)).exp() x odds.row(k) / scalar
        base - offset
      })
    resleft ++ resright
  }
}
def convolve2d_size3(image: Matrix, kernel: Matrix): Matrix = {
 require(image >= -153.55 && image <= 291.35 && image.size(81,81) // small (3,3), medium (9,9)
  && kernel >= -104.89 && kernel <= 57.21 && kernel.size(3, 3))
  val flippedK: Matrix = (kernel.flipud()).fliplr()
  val padded: Matrix = image.pad(1,1)
  val output: Matrix = padded.slideReduce(3, 1)(m => {
   val tmp: Matrix = flippedK.*(m)
   tmp.foldElements(0.0)((acc, x) => acc + x)
  })
  output
}
def sobel3(im: Matrix): Matrix = {
 require(im >= 251.34 && im <= 341.89 && im.size(81,81)) // small (3,3), medium (9,9)
  val kh: Matrix = Matrix(List(List(-1, 0, 1), List(-2, 0, 2), List(-1, 0, 1)))
  val kv: Matrix = Matrix(List(List(1, 2, 1), List(0, 0, 0), List(-1, -2, -1)))
  val padded: Matrix = im.pad(1,1)
  // inlined convolve 2d for kh
  val flippedKh: Matrix = (kh.flipud()).fliplr()
   val gx: Matrix = padded.slideReduce(3, 1)(m => {
    val tmp: Matrix = flippedKh * m
    tmp.foldElements(0.0)((acc, x) => acc + x)
  })
  // inlined convolute 2d for kv
  val flippedKv: Matrix = (kv.flipud()).fliplr()
  val gy: Matrix = padded.slideReduce(3, 1)(m => {
   val tmp: Matrix = flippedKv * m
   tmp.foldElements(0.0)((acc, x) => acc + x)
  })
  val pre: Matrix = gx * gx + gy * gy
  val g: Matrix = pre.sqrt()
  g * 255.0 / g.max()
}
def lorentz(m:Matrix): Vector = {
require(m >= 1.0 && m <= 2.0 && m.size(41,3)) // small (21,3); medium (31,3)
  val init: Vector = m.row(0)
    m.fold(init)((acc, v) => {
      val x:Real = acc.at(0)
      val y:Real = acc.at(1)
      val z:Real = acc.at(2)
      val tmpx:Real = x + 10.0*(y - x)*0.005
      val tmpy:Real = y + (28.0*x - y - x*z)*0.005
      val tmpz:Real = z + (x*y - 2.666667*z)*0.005
      Vector(List(tmpx,tmpy,tmpz))
    })
}
def lyapunov(x: Vector, weights1: Matrix, weights2: Matrix,
       bias1: Vector, bias2: Real): Vector = {
require(0.5307131 <= bias2 && bias2 <= 0.5307131
 && x >= -6.0 && x <= 6.0 && x.size(500)
 // small 10; medium 100 for vectors
 && bias1 >= -0.8746956 && bias1 <= 1.1860801 && bias1.size(500)
 // small (10,10); medium (100,100)
 && weights1 >= -0.6363012 && weights1 <= 1.0211772 && weights1.size(500,500)
 && weights2 >= -0.80846876 && weights2 <= 1.1081733 && weights2.size(1,500))
  val layer1: Vector = (weights1.x(x) + bias1).map(el => {
    val relu = Vector(List(el, 0.0))
      relu.max()
    })
  val layer2: Vector = (weights2.x(layer1) + bias2).map(el => {
    val relu = Vector(List(el, 0.0))
      relu.max()
    })
    layer2
  }
def controllerTora(x: Vector, weights1: Matrix, weights2: Matrix,
         weights3: Matrix, weights4: Matrix, bias1: Vector,
         bias2: Vector, bias3: Vector, bias4: Real): Vector = {
 require(10.197819 <= bias4 && bias4 <= 10.197819
  && x >= -2.0 && x <= 2.0 && x.size(500)
  // small 10; medium 100 for vectors
  && bias1 >= 0.040232 && bias1 <= 0.341392 && bias1.size(500)
  && bias2 >= 0.082624 && bias2 <= 0.318763 && bias2.size(500)
  && bias3 >= 0.096189 && bias3 <= 0.297542 && bias3.size(500)
  // small (10,10); medium (100,100) for matrices
  && weights1 >= -0.374036 && weights1 <= 0.319683 && weights1.size(500,500)
  && weights2 >= -0.426394 && weights2 <= 0.323056 && weights2.size(500,500)
  && weights3 >= -0.582338 && weights3 <= 0.566423 && weights3.size(500,500)
  && weights4 >= -0.293298 && weights4 <= 0.311236 && weights4.size(1,500))
  val layer1 = (weights1.x(x) + bias1).map(el => {
    val relu = Vector(List(el, 0.0))
      relu.max()
    })
  val layer2 = (weights2.x(layer1) + bias2).map(el => {
    val relu = Vector(List(el, 0.0))
      relu.max()
    })
  val layer3 = (weights3.x(layer2) + bias3).map(el => {
    val relu = Vector(List(el, 0.0))
      relu.max()
    })
  val layer4 = (weights4.x(layer3) + bias4)
    layer4
  }
}
```

Listing A.1: DS2L's benchmarks

A.1.2. Benchmarks formatted for Fluctuat

The benchmarks formatted for Fluctuat include its own mathematical library `fluctuat_math.h`. The range of inputs $[low, hi]$ is denoted by the command `DBETWEEN(low, hi);` that provides a single (scalar) double floating-point value. The size of input data structures is set in the variable `N` using the `#define` instruction.

```
# include <fluctuat_math.h>
#define N 10000 // small 100, medium 1000
double avg(double∗ xin) {
    double res, acc;
    int i;
    acc = 0.0; // init
    for(i=0; i<N; i++) {
        acc = acc + xin[i];
    }
    res = acc / N;
    return res;
}
int main() {
    int i;
    double res;
    double x[N];
    // specify input ranges
    for(i=0; i<=99; i++) {
       x[i] = DBETWEEN(-62.54, 15.02);
    }
    res = avg(x);
}
```

Listing A.2: avg

```
#include <fluctuat_math.h>
#define N 10000 // small 100, medium 1000
double variance(double∗ x) {
    double res, acc1, acc2, avg;
    int i;
    acc1 = 0.0; // init
    for(i=0; i<N; i++) {
        acc1 = acc1 + x[i];
    }
    avg = acc1 / N;
    acc2 = 0.0; // init
    for(i=0; i<N; i++) {
        acc2 = acc2 + pow(x[i] - avg, 2);
    }
    res = acc2 / N;
    return res;
}
int main() {
    int i;
    double res;
    double x[N];
    // specify input ranges
    for(i=0; i<=99; i++) {
      x[i] = DBETWEEN(-252.68, 72.42);
    }
    res = variance(x);
}
```

Listing A.3: variance

```
# include <fluctuat_math.h>
#define N 10000 // small 100, medium 1000
double stdDeviation(double* x) {
    double res, acc1, acc2, avg;
    int i;
    acc1 = 0.0; // init
    for(i=0; i<N; i++) {
        acc1 = acc1 + x[i];
    }
    avg = acc1 / N;
    acc2 = 0.0; // init
    for(i=0; i<N; i++) {
        acc2 = acc2 + pow(x[i] - avg, 2);
    }
    res = sqrt(acc2 / N);
    return res;
}
int main() {
    int i;
    double res;
    double x[N];
    for(i=0; i<N; i++) {
      x[0] = DBETWEEN(-160.06, 360.98);
    }
    res = stdDeviation(x);
}
```

Listing A.4: stdDeviation

```
# include <fluctuat_math.h>
#define N 10000 // small 100, medium 1000
double roux(double* x) {
    //x.fold(0.0)((y: Real, i: Real) => {1.5 * i - 0.7 * y})
    double acc;
    int i;
    acc = 0.0; // init
    for(i=0; i<N; i++) {
        acc = 1.5*x[i] - 0.7*acc;
    }
    return acc;
}
int main() {
    int i;
    double res;
  double x[N];
    // specify input ranges
    for(i=0; i<=99; i++) {
      x[i] = DBETWEEN(-58.25, 61.32);
  }
    res = roux(x);
}
```

Listing A.5: roux

```
#include <fluctuat_math.h>
#define N 10000 // small 100, medium 1000
double goubault(double x[], double y) {
    // x.fold(y)((acc: Real, xi: Real) => {0.75 * xi - 0.125 * acc})
    double acc;
    int i;
    acc = y; // init
    for(i=0; i<N; i++) {
        acc = 0.75*x[i] - 0.125*acc;
    }
    return acc;
}
int main() {
    int i;
    double y, res;
  double x[N];
    // specify input ranges
    for(i=0; i<=99; i++) {
      x[i] = DBETWEEN(-270.01, 385.38);
  }

    y = DBETWEEN(-1.0,0.0);
    res = goubault(x, y);
}
```

Listing A.6: goubault

```
#include <fluctuat_math.h>
#define N 10000 // small 100, medium 1000
void harmonic(double x[N], double y[N], double *res) {
    //x1 := x1 + 0.01 * x2
    // val x1: Real = y.fold(x.head)((acc: Real, xi: Real) => {acc + 0.01* xi})
    //x2 := -0.01 * x1 + 0.99 * x2
    // val x2: Real = x.fold(y.head)((acc: Real, xi: Real) => {-0.01 * xi + 0.99 * acc})
    double x1, x2;
    int i;
    x1 = 0.0;
    x2 = 0.0;
    for(i=0; i<N; i++) {
        x1 = x1 + 0.01 * y[i];
        x2 = -0.01 * x[i] + 0.99 * x2;
    }
    res[0] = x1;
    res[1] = x2;
}
int main() {
    int i;
    double res[2];
  double x[N], y[N];
    // specify input ranges
    for(i=0; i<=99; i++) {
      x[i] = DBETWEEN(-5.32, 725.6);
  }
for(i=0; i<=99; i++) {
      y[i] = DBETWEEN(-432.12, 78.94);
  }
    harmonic(x, y, res);
}
```

Listing A.7: harmonic

```
#include <fluctuat_math.h>
#define N 10000 // small 100, medium 1000
void nonlin1(double x[], double y[], double* res) {
    double x1, y1;
    int i;
    x1 = x[0];
    y1 = y[0];
    for(i=0; i<N; i++) {
        x1 = x1 + 0.01 * (-2*x1 - 3*y[i] + x1*x1);
        y1 = y1 + 0.01 * (x[i] + y1);
    }
    res[0] = x1;
    res[1] = y1;
}
int main() {
    int i;
    double res[2];
    double x[N], y[N];
    // specify input ranges
    for(i=0; i<=99; i++) {
      x[i] = DBETWEEN(0.0, 1.0);
  }
for(i=0; i<=99; i++) {
      y[i] = DBETWEEN(0.0, 1.0);
  }
    nonlin1(x, y, res);
}
```

Listing A.8: nonlin1

```
#include <fluctuat_math.h>
#define N 10000 // small 100, medium 1000
void nonlin2(double x[], double y[], double* res) {
    double x1, y1;
    int i;
    x1 = x[0];
    y1 = y[0];
    for(i=0; i<N; i++) {
        x1 = x1 + 0.01 * (-x1 +2*x1*x1 + y[i]*y[i]);
        y1 = y1 + 0.01 * (-y1 + y1*y1);
    }
    res[0] = x1;
    res[1] = y1;
}
int main() {
    int i;
    double res[2];
    double x[N], y[N];
    // specify input ranges
    for(i=0; i<=99; i++) {
      x[i] = DBETWEEN(0.0, 1.0);
  }
for(i=0; i<=99; i++) {
      y[i] = DBETWEEN(0.0, 1.0);
  }
    nonlin2(x, y, res);
}
```

Listing A.9: nonlin2

```
#include <fluctuat_math.h>
#define N 10000 // small 100, medium 1000
void nonlin3(double x[], double y[], double* res) {
    double x1, y1;
    int i;
    x1 = x[0];
    y1 = y[0];
    for(i=0; i<N; i++) {
        x1 = x1 + 0.01 * (-x1 + y[i]*y[i]);
        y1 = y1 + 0.01 * (-2.0*y1 + 3.0*x[i]*x[i]);
    }
    res[0] = x1;
    res[1] = y1;
}
int main() {
    int i;
    double res[2];
    double x[N], y[N];
    // specify input ranges
    for(i=0; i<=99; i++) {
      x[i] = DBETWEEN(0.0, 1.0);
  }
for(i=0; i<=99; i++) {
      y[i] = DBETWEEN(0.0, 1.0);
  }
    nonlin3(x, y, res);
}
```

Listing A.10: nonlin3

```
#include <fluctuat_math.h>
#define N 513 // small 33, medium 65
double heat1d(double ( *xm)[N], double ( *xp)[N], double* x0) {
    int i,j;
    for(j=1;j<N; j++) {
        for(i=2; i<(N-j); i++) {
        xm[j][i] = 0.25 * xm[j-1][i + 1] + 0.5 * xm[j-1][i] + 0.25 * xm[j-1][i - 1];
        xp[j][i] = 0.25 * xp[j-1][i - 1] + 0.5 * xp[j-1][i] + 0.25 * xp[j-1][i + 1];
    }
    xm[j][0] = 0.25 * xm[j-1][1] + 0.5 * xm[j-1][0] + 0.25 * x0[j-1];
    xp[j][0] = 0.25 * xp[j-1][1] + 0.5 * xp[j-1][0] + 0.25 * x0[j-1];
    x0[j] = 0.25*xm[0][j-1] + 0.5*x0[j-1] + 0.25*xp[0][j-1];
   }
   // Satire takes x0_32
   return x0[N-1];
}
int main() {
    int i,j;
    double res;
    double x0[N];
    double xm[N][N];
    double xp[N][N];
    double ax[N];
    // specify input ranges
    for(i=0; i<=32; i++) {
      ax[i] = DBETWEEN(1.0, 2.0);
  }

    // assign the ranges to match scala benchmark
    for(i=0; i<N; i++){
        x0[i] = ax[i];
        for(j=0; j<N; j++){
      xm[i][j] = ax[j];
      xp[i][j] = ax[j];
```

```
        }
    }
    heat1d(xm, xp, x0);
}
```

Listing A.11: heat1d

```
#include <fluctuat_math.h>
#define N 512 // small 4; medium 128
#define FFTLOGSIZE 9 // small 2; medium 7
void fftvector(double∗ vr, double∗ vi) {
    double scalar = 1.0;
    double pi = 3.1415926;
    int n = N; //LEN(m);
    double direction[2] = {0.0,2.0};
    int i,j,c;
    for(c=FFTLOGSIZE-1;c>0;c--){
      for(j=0;j<pow(2,c-1);j+=1) {
        int k=0;
        for(i=j;i<N;i+=pow(2,c)){
          int oddindex = i+ pow(2,c-1);
          double base[2] = {vr[i]/scalar, vi[i]/scalar};
          double tmp[2], offset[2];

          tmp[0] = exp(direction[0]∗pi∗k / n);
          tmp[1] = exp(direction[1]∗pi∗k / n);

          // val firstElt = Minus(Times(a,c), Times(b,d))
          // val secondElt = Plus(Times(a,d), Times(b,c))
          offset[0] = (tmp[0] ∗ vr[oddindex] - tmp[1] ∗ vi[oddindex]) / scalar;
          offset[1] = (tmp[0] ∗ vi[oddindex] - tmp[1] ∗ vr[oddindex]) / scalar;
           // lefts
           vr[i] = base[0] + offset[0];
           vi[i] = base[1] + offset[1];
           // rights
           vr[oddindex] = base[0] - offset[0];
           vi[oddindex] = base[1] - offset[1];
           k++;
        }
      }
        }
}
int main() {
    int i,j;
    double vr[N];
    double vi[N];
    // specify input ranges
    for(i=0; i<=3; i++) {
      vr[i] = DBETWEEN(68.9, 160.43);
```

```
  }
for(i=0; i<=3; i++) {
      vi[i] = DBETWEEN(-133.21, 723.11);
  }
    fftvector(vr, vi);
}
```

Listing A.12: fftvector

```
#include <fluctuat_math.h>
#define N 10000 // small 100; medium 1000
void pendulum(double t[N], double w[N], double* res){
    int i;
    double g,h,L;
    g = 9.80665;
    h = 0.01;
    L = 2.0;
    for(i=1; i<N; i++){
        double kt = w[i-1];
        double kw = -g/L * sin(t[i-1]);
        w[i] = w[i-1] + kw*h;
        t[i] = t[i-1] + kt*h;
    }
    res[0] = t[N-1];
    res[1] = w[N-1];
}
int main() {
    int i;
    double res[2];
    double t[N], w[N];
    // specify input ranges
    for(i=0; i<=99; i++) {
      t[i] = DBETWEEN(-2.0, 2.0);
  }
  for(i=0; i<=99; i++) {
      w[i] = DBETWEEN(-5.0, 5.0);
  }
  pendulum(t, w, res);
}
```

Listing A.13: pendulum

```
#include <fluctuat_math.h>
#define N 500 // small 10, medium 100
void alphaBlending(double b[N][N], double c[N][N], double alpha, double ( *res)[N]) {
    int i,j;
     // b * alpha + c * (1 - alpha)
    for(i=0; i<N; i++)
        for(j=0; j<N; j++) {
```

```
      res[i][j] = b[i][j] * alpha + c[i][j]* (1-alpha);
        }
}
int main() {
    int i,j;
    double alpha;
    double b[N][N];
    double c[N][N];
    double res[N][N];
    alpha = DBETWEEN(0.0, 1.0);
    for(i=0; i<N; i++)
     for(j=0; j<N; j++) {
       b[i][j] = DBETWEEN(223.35, 530.05);
    }
    for(i=0; i<N; i++)
     for(j=0; j<N; j++) {
       c[i][j] = DBETWEEN(-253.26, -108.41);
    }
    alphaBlending(b, c, alpha, res);
}
```

Listing A.14: alphaBlending

```
#include <fluctuat_math.h>
#define N 512 // small 4; medium 128
#define FFTLOGSIZE 9 // small 2; medium 7
void fftmatrix(double m[N][2]){
    double scalar = 1.0;
    double pi = 3.1415926;
    int n = N; //LEN(m);
    double direction[2] = {0.0,2.0};
    int i,j,c;
    for(c=FFTLOGSIZE-1;c>0;c--){
      // left
      for(j=0;j<pow(2,c-1);j+=1) {
        int k=0;
        for(i=j;i<N;i+=pow(2,c)){
          int oddindex = i+ pow(2,c-1);
          double base[2] = {m[i][0]/scalar, m[i][1]/scalar};
          double tmp[2], offset[2];

          tmp[0] = exp(direction[0]*pi*k / n);
          tmp[1] = exp(direction[1]*pi*k / n);

          // val firstElt = Minus(Times(a,c), Times(b,d))
          // val secondElt = Plus(Times(a,d), Times(b,c))
          offset[0] = (tmp[0]*m[oddindex][0] - tmp[1]*m[oddindex][1]) / scalar;
          offset[1] = (tmp[0]*m[oddindex][1] - tmp[1]*m[oddindex][0]) / scalar;
```

```
           // lefts
           m[i][0] = base[0] + offset[0];
           m[i][1] = base[1] + offset[1];
           // rights
           m[oddindex][0] = base[0] - offset[0];
           m[oddindex][1] = base[1] - offset[1];
           k++;
        }
      }
  }
}
int main() {
    int i,j;
    double m[N][2];
    // specify input ranges
    for (i=0; i < N; i++) {
      for (j=0; j < N; j++) {
        m[i][j] = DBETWEEN(-326.68, 677.57);
      }
    }
    fftmatrix(m);
}
```

Listing A.15: fftmatrix

```
#include <fluctuat_math.h>
#define N 81 // small 3; medium 9
#define M 3
void convolve2d_size3(double image[N][N], double kernel[M][M], double ( *res)[N]) {
    // output
    double padded[N+2][N+2];
    double flippedK[M][M];
    int i,j,k,l;
    // flip upside down and left to right
    for(i=0; i<=(M)/2; i++){
      for(j=0; j<=(M)/2; j++) {
         flippedK[i][j] = kernel[M-i-1][M-j-1];
         flippedK[M-i-1][j] = kernel[i][M-j-1];
         flippedK[M-i-1][M-j-1] = kernel[i][j];
         flippedK[i][M-j-1] = kernel[M-i-1][j];
        }
     }

    // pad
    for(i=0; i<N+2;i++)
        for(j=0; j<N+2;j++) {
        if (1<= i && i <= N && 1<= j && j <= N)
        padded[i][j] = image[i-1][j-1];
        else
```

```c
    padded[i][j]=0.0;
        }
    // slide + reduce
    for(i=0; i<N; i++)
        for(j=0; j<N; j++) {
      double tmp = 0.0;
      for(k=0;k<M;k++)
        for(l=0;l<M;l++){
          // reduce body
          tmp = tmp + padded[i+k][j+l] * flippedK[k][l];
        }
      res[i][j] = tmp;
  }
}
int main() {
    int i,j;
    double image[N][N];
    double kernel[M][M];
    double res[N][N];
    // specify input ranges
    for (i=0; i < N; i++) {
      for (j=0; j < N; j++) {
        image[i][j] = DBETWEEN(-153.55, 291.35);
        kernel[i][j] = DBETWEEN(-104.89, 57.21);
      }
    }
    convolve2d_size3(image, kernel, res);
}
```

Listing A.16: convolve2d_size3

```c
#include <fluctuat_math.h>
#define N 81 // small 3; medium 9
#define M 3
void sobel3(double image[N][N], double ( *res)[N]) {
    double kh[M][M],kv[M][M], flippedKh[M][M], flippedKv[M][M];
    int i,j,k,l;
    //kh initialize
    kh[0][0] = -1.0; kh[0][1] = 0.0; kh[0][2] = 1.0;
    kh[1][0] = -2.0; kh[1][1] = 0.0; kh[1][2] = 2.0;
    kh[2][0] = -1.0; kh[2][1] = 0.0; kh[2][2] = 1.0;
    //kv initialize
    kv[0][0] = 1.0; kv[0][1] = 2.0; kv[0][2] = 1.0;
    kv[1][0] = 0.0; kv[1][1] = 0.0; kv[1][2] = 0.0;
    kv[2][0] = -1.0; kv[2][1] = -2.0; kv[2][2] = -1.0;
    double padded[N+2][N+2];
    for(i=0; i<N+2;i++)
        for(j=0; j<N+2;j++) {
        if (1<= i && i <= N && 1<= j && j <= N)
        padded[i][j] = image[i-1][j-1];
        else
        padded[i][j]=0.0;
        }
    double gx[N][N], gy[N][N];
    // flip upside down and left to right
    for(i=0; i<=(M)/2; i++){
      for(j=0; j<=(M)/2; j++) {
         flippedKh[i][j] = kh[M-i-1][M-j-1];
         flippedKh[M-i-1][j] = kh[i][M-j-1];
         flippedKh[M-i-1][M-j-1] = kh[i][j];
         flippedKh[i][M-j-1] = kh[M-i-1][j];
        }
     }
    // slide + reduce
    for(i=0; i<N; i++)
        for(j=0; j<N; j++) {
      double tmp = 0.0;
      for(k=0;k<M;k++)
        for(l=0;l<M;l++){
          // reduce body
          tmp = tmp + padded[i+k][j+l] * flippedKh[k][l];
        }
      gx[i][j] = tmp;
        }
    // flip upside down and left to right
    for(i=0; i<=(M)/2; i++){
      for(j=0; j<=(M)/2; j++) {
         flippedKv[i][j] = kv[M-i-1][M-j-1];
         flippedKv[M-i-1][j] = kv[i][M-j-1];
         flippedKv[M-i-1][M-j-1] = kv[i][j];
         flippedKv[i][M-j-1] = kv[M-i-1][j];
        }
     }
    // slide + reduce
    for(i=0; i<N; i++)
        for(j=0; j<N; j++) {
      double tmp = 0.0;
      for(k=0;k<M;k++)
        for(l=0;l<M;l++){
          // reduce body
          tmp = tmp + padded[i+k][j+l] * flippedKv[k][l];
        }
      gy[i][j] = tmp;
        }
    double g[N][N];
    double maxg = -1.7e+308;
    for(i=0; i<N; i++)
```

```
        for(j=0; j<N; j++) {
      g[i][j] = sqrt(gx[i][j] * gx[i][j] + gy[i][j] * gy[i][j]);
      if (g[i][j] > maxg)
        maxg = g[i][j];
        }
     for(i=0; i<N; i++)
        for(j=0; j<N; j++) {
      res[i][j] = g[i][j] * 255.0 / maxg;
        }
}
int main() {
    int i,j;
    double im[N][N];
    double res[N][N];
    // specify input ranges
    for (i=0; i < N; i++) {
      for (j=0; j < N; j++) {
        im[i][j] = DBETWEEN(251.34, 341.89);
      }
    }

    sobel3(im, res);
}
```

Listing A.17: sobel3

```
#include <fluctuat_math.h>
#define N 41 // small 21; medium 31
void lorentz(double x, double y, double z, double* res){
    int i;
    for(i=0;i<N;i++){
        double tmpx,tmpy,tmpz;
        tmpx = (x + 10.0*(y - x)*0.005);
        tmpy = (y + (28.0*x - y - x*z)*0.005);
        tmpz = (z + (x*y - 2.666667*z)*0.005);
        x = tmpx;
        y = tmpy;
        z = tmpz;
    }
    res[0] = x;
    res[1] = y;
    res[2] = z;
}
int main() {
    int i,j;
    double res[3];
    double m[N][3];
    // specify input ranges
    for (i=0; i < N; i++) {
      for (j=0; j < N; j++) {
        m[i][j] = DBETWEEN(1.0, 2.0);
      }
    }
    double xin,yin,zin;
    xin = m[0][0];
    yin = m[0][1];
    zin = m[0][2];
    lorentz(xin, yin, zin, res);
}
```

Listing A.18: lorentz

```
#include <fluctuat_math.h>
#define N 500 // small 10; medium 100
double lyapunov(double x[N], double weights1[N][N],
                double weights2[1][N], double bias1[N], double bias2[1]) {
  int i,j;
  double layer1[N];
  double layer2;
  for (i=0; i < N; i++) {
    double _dot_tmp = 0;
    for (j=0; j < N; j++) {
      _dot_tmp = _dot_tmp + weights1[i][j] * x[j];
    }
    double _bias_tmp = _dot_tmp + bias1[i];
    if (_bias_tmp < 0)
      layer1[i] = 0.0;
    else
      layer1[i] = _bias_tmp;
  }
  double _dot_tmp = 0;
   for (j=0; j < N; j++) {
    _dot_tmp = _dot_tmp + weights2[0][j] * layer1[j];
    }
  double _bias_tmp = _dot_tmp + bias2[0];
  if (_bias_tmp < 0)
      layer2 = 0.0;
    else
      layer2 = _bias_tmp;
  return layer2;
}
int main() {
    int i,j;
    double res;
    double x[N];
    double weights1[N][N];
    double weights2[1][N];
    double bias1[N];
    double bias2[1];
    // specify input ranges
    for (i=0; i < N; i++) {
      bias1[i] =  DBETWEEN(-0.8746956, 1.1860801);
      x[i] = DBETWEEN(-6.0, 6.0);
      for (j=0; j < N; j++) {
        weights1[i][j] = DBETWEEN(-0.6363012, 1.0211772);
        weights2[i][j] = DBETWEEN(-0.80846876, 1.1081733);
      }
    }
  res = lyapunov(x, weights1, weights2, bias1, bias2);
}
```

Listing A.19: lyapunov

```
#include <fluctuat_math.h>
#define N 500 // small 10; medium 100
double controllerTora(double x[N], double weights1[N][N], double weights2[N][N],
            double weights3[N][N], double weights4[1][N], double bias1[N],
            double bias2[N], double bias3[N], double bias4[1]) {
  double layer1[N];
  double layer2[N];
  double layer3[N];
  double layer4;
  int i,j;
  for (i=0; i < N; i++) {
    double _dot_tmp = 0;
    for (j=0; j < N; j++) {
      _dot_tmp = _dot_tmp + weights1[i][j] * x[j];
    }
    double _bias_tmp = _dot_tmp + bias1[i];
    if (_bias_tmp < 0)
      layer1[i] = 0.0;
    else
      layer1[i] = _bias_tmp;
  }
  for (i=0; i < N; i++) {
    double _dot_tmp = 0;
    for (j=0; j < N; j++) {
      _dot_tmp = _dot_tmp + weights2[i][j] * layer1[j];
    }
    double _bias_tmp = _dot_tmp + bias2[i];
    if (_bias_tmp < 0)
      layer2[i] = 0.0;
    else
      layer2[i] = _bias_tmp;
  }

  for (i=0; i < N; i++) {
    double _dot_tmp = 0;
    for (j=0; j < N; j++) {
      _dot_tmp = _dot_tmp + weights3[i][j] * layer2[j];
    }
    double _bias_tmp = _dot_tmp + bias3[i];
    if (_bias_tmp < 0)
      layer3[i] = 0.0;
    else
      layer3[i] = _bias_tmp;
  }

  double _dot_tmp = 0;
   for (j=0; j < N; j++) {
    _dot_tmp = _dot_tmp + weights4[0][j] * layer3[j];
    }
  layer4 = _dot_tmp + bias4[0];
  return layer4;
} // [-80.75012614511054, 125.34998463158972] +/- 4.989472734247575e-05
int main() {
    int i,j;
    double res;
    double x[N];
    double weights1[N][N];
    double weights2[N][N];
    double weights3[N][N];
    double weights4[1][N];
    double bias1[N];
    double bias2[N];
    double bias3[N];
    double bias4[1];
    // specify input ranges
    for (i=0; i < N; i++) {
      bias3[i] = DBETWEEN(0.096189, 0.297542);
      bias2[i] = DBETWEEN(0.082624, 0.318763);
      bias1[i] = DBETWEEN(0.040232, 0.341392);
      x[i] = DBETWEEN(-2.0, 2.0);
      for (j=0; j < N; j++) {
        weights1[i][j] = DBETWEEN(-0.374036, 0.319683);
        weights2[i][j] = DBETWEEN(-0.426394, 0.323056);
        weights3[i][j] = DBETWEEN(-0.582338, 0.566423);
        weights4[i][j] = DBETWEEN(-0.293298, 0.311236);
      }
    }
    res = controllerTora(x, weights1, weights2, weights3, weights3, bias1, bias2, bias3, bias4);
}
```

Listing A.20: controllerTora

A.2. Experimental Data for DS2L

We provide the additional experimental results for small and medium benchmarks used to evaluate DS2L in section 3.5. As the amount of data points is too large to be included in one table, we split the results by size of the input DS.

Whenever a tool has failed to report the error bound we use "-" to denote it, we also indicate reported *overflow* explicitly, we write ∞ if the reported error bounds were $[-\infty, \infty]$. We use "TO" to denote 30-minute timeouts and any other other tool failures. Reported time is the analysis time in seconds. "*na*" in Satire's results denotes that we did not run Satire on these variations of *heat1d* or *lorentz*, as we only took the original benchmarks that had same ranges for all input DS elements.

	AllSame		*Diff10P*		*Diff30P*		*AllDiff*	
Benchmark	error	time	error	time	error	time	error	time
DS2L								
avg	3.65e-12	0.51	3.41e-12	0.59	2.01e-12	0.85	1.52e-12	2.18
variance	4.64e-06	4.90	2.23e-06	7.18	6.66e-07	8.57	4.85e-07	14.24
stdDev.	1130	4.93	2.00e-07	7.09	2.59e-08	8.67	2.54e-08	14.13
roux1	7.21e-14	0.77	1.40e-13	1.17	1.22e-13	1.83	1.85e-13	4.01
goubault	7.46e-14	0.78	7.85e-14	1.03	8.03e-14	1.65	7.90e-14	3.74
harmonic	4.56e-10	1.22	1.94e-10	1.99	1.71e-10	3.01	1.59e-10	9.64
nonlin1	*overflow*	-	*overflow*	-	*overflow*	-	*overflow*	-
nonlin2	*overflow*	-	*overflow*	-	*overflow*	-	*overflow*	-
nonlin3	4.32e-05	31.44	2.91e-05	40.22	1.63e-05	49.39	1.16e-05	75.38
pendulum	1.16e-04	28.29	5.50e-05	39.36	5.50e-05	49.78	6.65e-05	86.38
heat1d	1.27e-14	7.10	8.34e-15	10.46	8.22e-15	10.50	8.22e-15	10.13
conv.2d_size3	3.15e-10	0.84	1.68e-10	1.11	2.62e-10	1.43	6.76e-11	2.26
sobel3	5.50	2.94	6.15	3.46	5.55	4.00	1.78	5.24
fftmatrix	3.71e-09	54.83	3.66e-09	56.50	3.58e-09	57.28	2.56e-09	57.85
fftvector	1.84e-09	39.54	1.48e-09	39.58	1.12e-09	39.59	1.05e-09	40.87
lorentz	2.28e-13	1.93	2.17e-13	1.82	1.53e-13	1.80	1.84e-13	1.79
alphaBlend.	3.14e-13	0.20	3.14e-13	1.07	3.14e-13	2.19	3.14e-13	785.51
contr.Tora	8.75e-08	6.93	6.14e-08	480.16	4.06e-08	496.04	7.33e-09	1016.20
lyapunov	5.88e-10	3.23	4.16e-10	97.30	3.37e-10	101.95	1.04e-10	210.29
Fluctuat								
avg	2.37e-12	3.75	2.06e-12	4.00	1.75e-12	3.00	1.49e-12	3.50
variance	-	TO	-	TO	-	TO	-	TO
stdDev.	-	TO	-	TO	-	TO	-	TO
roux1	2.10e-13	8.67	8.16e-14	8.00	5.15e-14	8.00	9.86e-14	8.00
goubault	6.50e-14	5.00	6.50e-14	4.67	6.09e-14	5.00	5.91e-14	5.00
harmonic	1.68e-10	16.00	1.34e-10	14.67	1.16e-10	15.00	1.09e-10	16.00
nonlin1	-	TO	-	TO	-	TO	-	TO
nonlin2	-	TO	-	TO	-	TO	-	TO
nonlin3	2.21e-08	29.67	8.24e-09	30.00	6.02e-09	32.33	6.92e-09	35.50
pendulum	1.12e-04	13.00	5.22e-05	13.67	5.22e-05	16.33	6.33e-05	20.00
heat1d	6.66e-16	236.33	4.44e-16	249.33	4.44e-16	248.33	4.44e-16	247.33
conv.2d_size3	1.24e-10	1.00	1.07e-10	1.33	1.24e-10	1.00	4.09e-11	1.33
sobel3	91.9	6.67	90.3	6.33	86.3	6.67	54.4	6.33
fftmatrix	∞	2.67	∞	2.67	∞	2.67	∞	2.67
fftvector	∞	0.99	∞	2.67	∞	2.67	∞	2.33
lorentz	5.41e-14	0.73	5.28e-14	0.73	4.73e-14	0.73	4.84e-14	0.83
alphaBlend.	1.56e-13	744.67	1.56e-13	843.67	1.56e-13	820.33	1.55e-13	743.67
contr.Tora	-	TO	-	TO	-	TO	-	TO
lyapunov	5.69e-10	264.00	4.01e-10	270.50	3.03e-10	269.00	7.89e-11	258.00
Satire								
avg	3.49e-12	18.70	2.84e-12	15.66	2.34e-12	14.45	2.05e-12	15.99
variance	1.83e-08	476.62	1.09e-08	457.62	6.11e-09	455.67	4.61e-09	458.18
stdDev.	-	TO	-	TO	-	TO	-	TO
roux1	2.55e-13	126.46	1.19e-13	118.79	7.00e-14	122.02	1.64e-13	120.40
goubault	1.14e-13	90.19	1.14e-13	83.96	1.10e-13	82.39	9.79e-14	82.82
harmonic	1.62e-11	93.69	1.29e-11	90.72	1.07e-11	89.45	1.03e-11	90.97
nonlin1	-	TO	-	TO	-	TO	-	TO
nonlin2	-	TO	-	TO	-	TO	-	TO
nonlin3	9.17e-15	491.27	6.04e-15	496.15	4.13e-15	468.28	4.94e-15	482.47
lorentz	3.50e-14	875.69	*na*	*na*	*na*	*na*	*na*	*na*

Table A.1.: Experimental results on medium benchmarks. Reported error bounds are rounded to two digits after decimal point, time is in seconds. "TO" denotes a timeout, "na" stands for non-applicable.

	AllSame		*Diff10P*		*Diff30P*		*AllDiff*	
Benchmark	error	time	error	time	error	time	error	time
DS2L								
avg	4.62e-13	0.15	2.60e-13	0.18	1.59e-13	0.17	1.42e-13	0.23
variance	5.98e-07	0.92	2.62e-07	1.05	7.53e-08	1.09	4.98e-08	1.23
stdDev.	145	1.03	2.71e-08	1.12	2.05e-09	1.15	3.40e-09	1.20
roux1	7.21e-14	0.20	2.20e-13	0.28	1.88e-13	0.37	1.60e-13	0.54
goubault	7.46e-14	0.21	8.04e-14	0.29	4.25e-14	0.31	6.59e-14	0.56
harmonic	1.15e-11	0.30	5.84e-12	0.41	5.81e-12	0.55	5.82e-12	1.02
nonlin1	*overflow*	-	*overflow*	-	1.17e-07	4.05	1.06e-09	4.29
nonlin2	*overflow*	-	*overflow*	-	*overflow*	-	*overflow*	-
nonlin3	7.82e-14	3.73	3.72e-14	4.21	3.18e-14	4.11	2.78e-14	4.49
pendulum	2.27e-13	2.73	2.27e-13	2.72	7.51e-14	3.29	1.73e-13	3.75
heat1d	6.44e-15	2.61	4.32e-15	3.12	4.22e-15	3.94	4.47e-15	3.46
conv.2d_size3	3.15e-10	0.23	1.51e-10	0.29	1.51e-10	0.29	4.07e-11	0.42
sobel3	5.50	0.86	4.42	0.89	4.42	0.88	6.72e-01	0.96
fftmatrix	4.29e-12	0.95	4.29e-12	1.01	4.29e-12	1.03	1.88e-12	0.96
fftvector	2.85e-12	0.83	2.40e-12	0.82	1.32e-12	0.79	1.86e-12	0.79
lorentz	4.33e-14	1.32	4.33e-14	1.30	2.55e-14	1.32	3.77e-14	1.27
alphaBlend.	3.14e-13	0.06	3.14e-13	0.10	3.14e-13	0.16	3.13e-13	0.80
contr.Tora	1.42e-12	0.45	1.42e-12	1.22	1.20e-12	1.77	1.74e-13	1.53
lyapunov	9.90e-13	0.28	8.87e-13	0.54	6.64e-13	0.86	1.68e-13	0.68
Fluctuat								
avg	2.62e-13	0.11	1.99e-13	0.07	1.62e-13	0.07	1.41e-13	0.07
variance	1.15e-09	6.33	6.17e-10	6.00	3.25e-10	6.00	2.16e-10	6.00
stdDev.	516	6.67	1.35e-11	6.00	5.38e-12	6.00	7.13e-12	6.00
roux1	2.10e-13	0.16	1.83e-13	0.12	1.46e-13	0.12	6.80e-14	0.12
goubault	6.50e-14	0.13	6.50e-14	0.09	2.43e-14	0.09	4.67e-14	0.09
harmonic	2.92e-12	0.34	2.47e-12	0.21	2.25e-12	0.21	2.11e-12	0.21
nonlin1	3.07e-14	16.33	2.78e-14	13.00	2.48e-14	13.00	2.73e-14	13.67
nonlin2	∞	16.67	2.83e-12	16.33	1.86e-13	16.67	∞	14.33
nonlin3	7.84e-15	0.55	4.29e-15	0.36	3.53e-15	0.38	3.44e-15	0.48
pendulum	2.15e-13	0.36	2.15e-13	0.20	6.67e-14	0.23	1.63e-13	0.24
heat1d	6.66e-16	12.67	4.44e-16	12.33	4.44e-16	12.67	4.44e-16	12.67
conv.2d_size3	1.24e-10	0.08	6.23e-11	0.08	6.23e-11	0.08	1.56e-11	0.08
sobel3	91.9	0.24	61.6	0.24	61.6	0.24	31.0	0.24
fftmatrix	1.09e-12	0.01	1.09e-12	0.01	1.09e-12	0.01	6.92e-13	0.01
fftvector	6.93e-13	0.01	6.93e-13	0.01	3.46e-13	0.01	6.90e-13	0.00
lorentz	2.16e-14	0.23	2.16e-14	0.23	1.55e-14	0.22	1.90e-14	0.23
alphaBlend.	1.56e-13	0.11	1.56e-13	0.12	1.56e-13	0.12	1.55e-13	0.12
contr.Tora	2.28e-12	0.57	1.80e-12	0.58	1.25e-12	0.56	1.17e-13	0.49
lyapunov	8.07e-13	0.09	7.08e-13	0.09	4.58e-13	0.10	9.34e-14	0.10
Satire								
avg	3.65e-13	1.74	2.93e-13	2.22	2.47e-13	2.07	1.93e-13	1.83
variance	1.92e-09	12.22	1.19e-09	16.23	7.57e-10	16.04	4.97e-10	16.54
stdDev.	-	TO	-	TO	-	TO	-	TO
roux1	2.55e-13	11.61	2.26e-13	13.39	1.88e-13	13.27	8.58e-14	13.18
goubault	1.14e-13	13.44	1.14e-13	15.37	4.97e-14	14.77	9.74e-14	14.55
harmonic	7.90e-12	19.48	5.01e-12	21.31	4.66e-12	20.48	4.90e-12	20.74
nonlin1	4.28e-14	655.23	3.99e-14	678.74	3.49e-14	687.85	3.87e-14	700.75
nonlin2	-	TO	1.94e-15	100.92	1.94e-15	89.29	-	TO
nonlin3	8.43e-15	32.45	5.62e-15	27.50	4.90e-15	26.09	3.86e-15	28.80
lorentz	9.68e-15	577.99	*na*	*na*	*na*	*na*	*na*	*na*
heat1d	1.98e-14	92.76	*na*	*na*	*na*	*na*	*na*	*na*

Table A.2.: Experimental results on small benchmarks. Reported error bounds are rounded to two digits after decimal point, time is in seconds. "TO" denotes a timeout, "na" stands for non-applicable.