SAARLAND UNIVERSITY

FACULTY OF NATURAL SCIENCES AND TECHNOLOGY I

DEPARTMENT OF COMPUTER SCIENCE

MAX-PLANCK INSTITUTE FOR SOFTWARE SYSTEMS

MASTER THESIS

Estimation of Relative Error Bounds for Floating-Point Arithmetic Expressions

---

submitted by

Anastasiia IZYCHEVA

Reviewers:

Dr. Eva DARULOVA

Prof. Bernd FINKBEINER

July 12, 2017

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, _____ _____

(Datum/Date) (Unterschrift/Signature)

Abstract

Current state-of-the-art tools for verifying finite-precision computations compute absolute error bounds on numerical errors. Unfortunately, absolute errors are often not a good estimate of accuracy as they do not take into account the magnitude of the computed values. Relative errors, which compute the errors relative to the magnitude, are thus preferable. However, no such general static verification technique exists today.

Computing relative errors accurately is already challenging by itself, in addition to that we have to deal with the fact that the definition of relative errors presents a potential inherent division by zero. We investigate different strategies for computing tight relative error bounds and present a practical solution for dealing with division by zero in this thesis. We have implemented several strategies within the approximating and verifying compiler Daisy and evaluated them on multiple benchmarks with encouraging results.

Contents

List of Tables

Chapter 1

Introduction

Embedded systems and scientific computing domains include many arithmetic computations. The algorithms are usually designed for the real-valued computations. However, if we want to implement continuous real-valued computations, we would need to have also continuous memory, or in other words *infinite* memory. Hence, real-valued computations have to be approximated and implemented in finite precision arithmetic.

One such arithmetic, that is ubiquitous in numerical software, is floating-point arithmetic. Floating-points are supported by most of today's hardware and software and therefore are a common default for implementing numerical computations. As floating-point arithmetic is an approximation of real (continuous) arithmetic, it inevitably introduces roundoff errors.

The roundoff errors are individually small but can accumulate and affect the validity of computed results. Furthermore, due to roundoff errors some useful real arithmetic properties do not hold anymore, for example in floating-point arithmetic addition is not associative $(a + b) + c \neq a + (b + c)$. Since most people reason in real arithmetic and design the algorithms assuming real arithmetic properties to hold, there arises a question: is what numerical software computes actually what the algorithm designer wanted to compute? To see how critical a deviation is, one can bound the difference between the real-valued result and the result of the floating-point computation.

One way to capture this difference is to compute the *absolute roundoff error*:

$$err_{\text{abs}} = \max_{x \in I} \left| f(x) - \hat{f}(\hat{x}) \right| \tag{1.1}$$

where x is a real-valued input, $f(x)$ is a real-valued function and $\hat{x}$ and $\hat{f}(\hat{x})$ are their floating-point counterparts. Note that for all possible inputs the error is in general unbounded; to compute useful bounds the input domain is restricted. A common way to define the input domain I is to use interval constraints for inputs, i.e. $I = [a, b]$, where $a, b \in \mathbb{R}$.

To compute absolute roundoff errors we need to bound the difference between a real-valued and a floating-point expression. The former is continuous, but the latter is highly discontinuous, making reasoning in both arithmetics simultaneously hard. Thus, automated tool support is essential.

Automated tools should be able to compute a sound absolute roundoff error bound, that is if we can state that for all possible inputs from input domain I the actual roundoff error does not exceed the computed bound. Thus, we want to compute a *sound worst-case* absolute roundoff error bound and we want to compute it as tight, i.e. as close to the real bound, as possible. For nonlinear expressions $f(x)$ computing tight bounds is challenging because of the correlations between variables that cannot be easily captured by standard range analyses (e.g. in standard interval arithmetic [1] $x - x \neq 0$). This leads to an over-approximation of the computed error bound.

Today's static analysis tools implement a variety of different strategies to deal with this over-approximation due to nonlinear expressions. Rosa [2] uses forward dataflow analysis combined with a nonlinear decision procedure, Fluctuat [3] extends a similar analysis with interval subdivision, FPTaylor [4] solves a global optimization problem using branch-and-bound method. Using all these techniques the tools compute quite tight absolute error bounds.

However, absolute errors are not always a good estimate for the result's accuracy. Suppose an automated tool reports an error of 0.01 for some function $f(x)$. Whether this error is small and acceptable or not depends on the application as well as the magnitude of the result's value. If the result's value $f(x) \gg 0.01$, the error may be acceptable, while if $f(x) \approx 0.01$ we should probably revise the computation or increase its precision. The measure that captures this relationship is called *relative error*:

$$err_{\text{rel}} = \max_{x \in I} \left| \frac{f(x) - \hat{f}(\hat{x})}{f(x)} \right| \tag{1.2}$$

In addition to the fact that relative errors are more informative, they are also more natural for user specifications. The user then needs to specify only one value to measure the quality of computations, and this specification can be applied to different input ranges without any changes.

Moreover, over-approximations of absolute errors grow with the size of the input domain. In general, the bigger the input domain is, the bigger the function range is and the less tight the error bounds are. Since relative errors take into account the resulting range, we expect the over-approximation to be smaller, which makes relative errors more suitable for modular verification.

Existing static analysis tools usually report both absolute and relative errors. The relative error in this case is computed as a relation between the absolute error and function value, i.e:

$$err_{\text{rel\_approx}} = \frac{\max_{x \in I} \left| f(x) - \hat{f}(\hat{x}) \right|}{\min_{x \in I} |f(x)|} \tag{1.3}$$

While this is definitely a sound approach, this is an over-approximation, because

err_{rel_approx} does not take into account the correlations between nominator and denominator. Thus, relative error bounds computed using Equation 1.3 are not necessarily tight.

In this thesis, we explore techniques to compute *sound* and *tight* worst-case relative error bounds.

One strategy that we investigate is to compute relative errors *directly* following the definition of relative error in Equation 1.2 as opposed to computing it via absolute errors as in Equation 1.3. However, the direct computation simply following the definition does not result in tight bounds for several reasons. First, the expression in Equation 1.2 is quite complex and thus its evaluation may be too expensive. Second, the division makes the relative error expression err_{rel} nonlinear even for linear $f(x)$. Thus, it has many nonlinear correlations between variables, which are hard to capture. There is a technique that allows to deal with both complexity of the expression and its nonlinearity, the *Symbolic Taylor Expansion* approach of FPTaylor which was previously applied to absolute error computations. We extend this approach to compute relative errors directly and show experimentally that the direct computation improves the bounds up to six orders of magnitude compared to computing them via absolute errors.

Another well-known technique for tightening the bounds that we explore is interval subdivision. Interval subdivision means that input intervals are broken into several smaller sub-intervals. Since smaller input domains in general cause smaller over-approximation committed by static analysis, the hope is that the over-approximation on several small sub-domains is less than for one big domain. We combine interval subdivision with both forward analysis and the optimization-based approaches and investigate its effect on the tightness of computed relative error bounds.

The second important challenge arising while computing relative error bounds using both Equation 1.2 and Equation 1.3 is potential inherent division by zero. Whenever the range of $f(x)$ contains zero, no useful bound can be computed. This indeed occurs often and today's static analysis tools report no relative error for most standard benchmarks for this reason.

In this thesis, we propose a practical (but preliminary) solution for dealing with division by zero. We localize its effect by using interval subdivision: our approach attempts to compute relative error for every sub-domain and records those sub-domains where division by zero occurred. For the latter we report absolute errors. Our experiments show that for the most of standard benchmarks the proposed practical approach provides more useful information than state-of-the-art tools.

Contributions

– We present the first systematic study of fully automated computation techniques for sound worst-case relative roundoff error bounds.

- We re-implement in Daisy the optimization-based approach for computing the absolute roundoff errors and provide the comparison with the absolute error bounds computed by the FPTaylor tool itself.
- We extend the optimization-based approach of FPTaylor to relative errors and thus provide the first feasible and fully automated approach for computing relative errors *directly*.
- We perform an extensive experimental evaluation of the implemented techniques and compare the relative error bounds and running times with state-of-the-art tools.
- We demonstrate that the direct computation of relative errors scales better than their computation via absolute errors.
- We show that interval subdivision is beneficial for reducing the over-approximation in absolute error computations, but less so for relative errors computed directly.
- We demonstrate that interval subdivision provides a practical solution to the division by zero challenge of relative error computation for certain benchmarks
- We have implemented all techniques within a single tool, called Daisy [5], allowing us to perform a direct comparison of the techniques. We release Daisy as open source, it is available at `https://gitlab.mpi-sws.org/AVA/daisy-public`.

Organization of Thesis

This thesis is organized as follows:

- Chapter 2 provides necessary background on floating-point arithmetic and range arithmetic and describes how today's static analysis tools use these for the absolute error bound computations.
- Chapter 3 describes the tool Daisy, where we have implemented all strategies for computing relative error bounds. This chapter details features of Daisy before the work presented in this thesis has been integrated into it.
- In Chapter 4 we present our implementation of the optimization-based approach for absolute roundoff errors from FPTaylor. While computing absolute errors is not the main focus of the thesis, it provides an interesting comparison with absolute errors computed using FPTaylor. We use this implementation and further extend it to relative errors.

- Chapter 5 details the techniques for computing tight relative error bounds that we have implemented inside Daisy: interval subdivision and direct relative error computation (instead of computing via absolute errors). We provide details on the implementation and report experimental evaluation results for all implemented strategies in comparison with state-of-the-art tools.
- In Chapter 6 we explain our practical approach for handling potential division be zero in computation of relative errors. We also show on the experimental result that the proposed solution is able to provide relative error bounds for sub-domains of several benchmarks, for which state-of-the-art tools could not compute any relative error estimate.
- Chapter 7 presents on overview of related work.

Chapter 2

Background

We start with necessary background about floating-point arithmetic and range arithmetic. We then explain how these are used in state-of-the-art techniques for automated sound worst-case absolute roundoff error estimation.

2.1 Floating-Point Arithmetic

Floating-point arithmetic is an approximation of real arithmetic. To be able to quantify the difference between these two, we first look at the implementation of floating-point arithmetic. Today most hardware and software supports the *IEEE 754* standard for floating-points [6]. In this thesis we also assume this standard and in the following subsection we provide the key aspects of it. For more detailed information about floating-point arithmetic, please see [6, 7].

IEEE 754

The IEEE 754 standard defines how floating-point arithmetic is implemented on a machine, by defining e.g. a concrete representation for floating-point numbers, a definition of arithmetic operations on these numbers, etc.

The general representation of a floating-point number has a base β (which is an even number) and a precision p. Using these parameters a floating-point number is represented as $\pm mm\ldots m \times \beta^e$, where $\pm$ is defined by a sign bit s, e is an exponent and $mm\ldots m$ is called the mantissa (or significand) and has p digits. That is $s.m_1m_2\ldots m_{p-1} \times \beta^e$ represents the number:

$$\pm \left(m_0 + m_1\beta^{-1} + \ldots + m_{p-1}\beta^{-(p-1)}\right)\beta^e, \qquad (0 \leq m_i \leq \beta) \tag{2.1}$$

The floating-point representation has two more parameters, the largest and the smallest allowed exponents, e_{max} and e_{min}. Knowing e_{max}, e_{min}, β and p, the amount of bits needed to encode the number can be calculated as [7]

$$\lceil \log_2(e_{max} - e_{min} + 1) \rceil + \lceil \log_2(\beta^p) \rceil + 1$$

where the final $+1$ is for the sign bit.

Parameter	Format			
	Single	**Single-Extended**	**Double**	**Double-Extended**
p	24	32	53	64
e_{max}	+127	1023	+1023	>16383
e_{min}	-126	$\leq$ -1022	-1022	$\leq$ -16382
Exponent width in bits	8	$\leq$ 11	11	15
Format width in bits	32	43	64	79

TABLE 2.1: IEEE 754 format parameters

IEEE 754 defines for $\beta = 2$ four different precisions: single, double, single-extended and double-extended. For example single precision occupies a single 32 bit word, out of which 1 bit is used for the sign, 8 bits for the exponent, and 23 bits for the mantissa. Parameter values for all data types are shown in Table 2.1. Most state-of-the-art tools for computing the roundoff error bounds support double precision, which is the most widely used default precision used today. To make a comparison with state-of-the-art tools in this thesis we also chose double precision, although our approach is parametric in this, and thus applicable to other floating-point precisions as well.

There are several special values, described in IEEE 754, whose representation differs from Equation 2.1 in that their exponents are out of normal range.

- *Zero* is encoded with exponent $e = e_{min} - 1$ and all zeros in the mantissa. As the sign bit has two possible values, there are two zero values: -0 and $+0$, but they compare as equal. Both are treated as normal floating-point numbers.

- *Denormalized numbers* (also called denormals or subnormals) are numbers between zero and the least representable normal value in the selected precision. Their exponent is $e = e_{min} - 1$ and the mantissa encodes the number itself. Denormals are intended to widen the range of real numbers for which some useful properties of real arithmetic hold. For example, if the difference between variables x and y is sufficiently small such that it cannot be represented as a normal floating-point number, the result of an operation $x - y$ would be flushed to zero, and then the property $x = y \leftrightarrow x - y = 0$ does not necessarily hold. A behavior of denormals preserving this property (and some other properties) is called gradual underflow.

- *Not a number (NaN)* represents an undefined or unrepresentable value. NaNs provide an alternative to halting a computation whenever operations like $0/0$ or $\sqrt{-1}$ occur. If at least one of the operands is NaN, the whole expression results in NaN. In IEEE 754, NaNs are represented with exponent $e = e_{max} + 1$ and a nonzero mantissa, which contains system-dependent information generated when the first NaN in the computation was generated (except when both

operands are NaN, then the information can describe either of them, and not necessarily the first generated).

– *Infinity* represents a value with an arbitrarily large magnitude. The standard distinguishes positive and negative infinities, which differ in the sign bit. For both $\pm\infty$, the exponent is $e = e_{max}+1$ and the mantissa contains only zeros. Infinities are in general useful, as they provide a way to continue a computation when an overflow occurs. Similarly when a division by zero occurs $c/0 = \pm\infty$, as long as $c \neq 0$. While all operations containing NaN result in NaN, expressions containing $\pm\infty$, however, might result in an ordinary floating-point number, e.g. $1/\infty = 0$. In general, the rule for determining the result of an operation where one operand is infinity is the following: replace infinity with a finite number x and take a limit with $x \to \infty$. When the limit does not exist, the result is a NaN, e.g ∞/∞ is NaN.

The standard also defines five rounding modes: rounding to nearest (ties to even, ties away from zero) and directed rounding (towards zero, $+\infty$, $-\infty$). We will consider the rounding mode that is used by the Java Virtual Machine (JVM) and is the default for most today's software, i.e. rounding to the nearest, ties to even.

According to IEEE 754, arithmetic operations are treated as if the result was first computed in infinite precision and then rounded to the specified finite precision. For our rounding mode this means that the result from any basic operation $(+, -, *, /, \sqrt{})$ is the closest representable floating-point number. Thus, the abstract model for floating-point variables and operations is the following:

$$\begin{gathered} \tilde{x} = x(1+e)+d \\ x \odot y = (x \circ y)(1+e)+d, \quad \circ \in \{+,-,*,/\}, \odot \in \{\oplus,\ominus,\otimes,\oslash\} \\ \tilde{\sqrt{x}} = (\sqrt{x})(1+e)+d, \\ |e| \leq \epsilon_M, |d| \leq \delta \end{gathered} \tag{2.2}$$

where ϵ_M is the maximum relative roundoff error introduced by rounding at each operation, and δ is the maximum absolute error used to represent roundoff error for denormals. For double precision floating-point numbers $\epsilon_M = 2^{-53}$ and $\delta = 2^{-1075}$. The abstraction in Equation 2.2 is only valid in the absence of overflows and invalid operations resulting in NaN, i.e. not for infinities and NaNs.

2.2 Range Arithmetic

To obtain a roundoff error for a program one can compute the floating-point function value for a single program execution and compare it with a result of a higher-precision computation or a manually computed real value. However, if we want to compute the roundoff error not only for a single input, but for a whole range, we need to apply range arithmetic to track function ranges and error bounds. In the following subsections we give a brief overview of different types of range arithmetic.

2.2.1 Interval Arithmetic

One of the basic types of range arithmetic is interval arithmetic [1]. In interval arithmetic every variable x is associated with a closed interval $[a, b]$, meaning that all possible values of x lie inside this interval, i.e. $a \leq x \leq b$. An interval $[a, a]$, for which the lower and the upper bounds match, is called a point interval.

Given a real-valued function $f(x_1, \ldots, x_n)$ and a list of intervals associated with its inputs $[a_i, b_i]$ interval arithmetic computes an over-approximation $[c, d]$ of the resulting function range over the input domain, i.e.:

$$\forall x_i.\, x_i \in [a_i, b_i] \rightarrow f(x_1, \ldots, x_n) \in [c, d] \tag{2.3}$$

To be able to perform range analysis, one needs to know how the basic arithmetic operations are defined. Assuming X and Y are intervals, the resulting range for one arithmetic operation is defined as

$$X \circ Y = \{z \mid \exists x \in X, y \in Y.\, z = x \circ y\} \quad \circ \in \{+, -, \times, /\} \tag{2.4}$$

Unfortunately, interval arithmetic often gives pessimistic bounds, because it cannot capture the correlation between variables. This is easy to see on a simple example, take for instance $x \in [0, b]$ and $f(x) = x - x$. Clearly, the resulting range for $f(x)$ is always zero independent from the interval of x. However, interval arithmetic computes the resulting range $[-b, b]$.

At the same time, this method is easy to implement and is reasonably fast, thus it provides an interesting accuracy-performance trade-off.

2.2.2 Affine Arithmetic

Unlike interval arithmetic, affine arithmetic [8] is able to track linear correlations between variables. Values of variables are represented by affine forms:

$$\breve{x} = x_0 + \sum_{i=1}^{n} x_i \epsilon_i, \tag{2.5}$$

where $\epsilon_i \in [-1, 1]$. Here x_0 denotes the mid point of the represented interval, so-called central value, and each term $x_i \epsilon_i$ is a noise term around the central value with maximum magnitude x_i. ϵ_i are called noise symbols, they capture linear correlations between variables.

The range represented by an affine form is computed as

$$[\breve{x}] = [x_0 - \sum_{i=1}^{n} |x_i|, x_0 + \sum_{i=1}^{n} |x_i|]$$

Linear arithmetic operations are performed term wise. A general linear affine operation $\alpha\breve{x} + \beta\breve{y} + \zeta$ consists of addition, subtraction, addition of a constant (ζ) and

multiplication by a constant (α, β). For affine forms $\breve{x}$ and $\breve{y}$, the operation is defined as:

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

Coming back to the example with function $f(x) = x - x$, we see that affine arithmetic captures the dependency between operands, and thus reduces the over-approximation by computing the resulting range zero:

$$x - x = (x_0 + x_1\epsilon_1) - (x_0 + x_1\epsilon_1) = x_0 - x_0 + x_1\epsilon_1 - x_1\epsilon_1 = 0$$

Unlike linear operations, which are computed exactly, the nonlinear operations like multiplication, inverse or square root have to be approximated. Multiplication is computed as:

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

where η is an over-approximation of the higher-order terms of the nonlinear operation. η can be computed in several different ways, the simplest one is $\eta = \sum_{i=1}^{n} |x_i| \times \sum_{i=1}^{n} |y_i|$.

Division is computed as $\frac{x}{y} = x \times \frac{1}{y}$ and other than that follows Equation 2.7. For unary functions, such as the square root and inverse, first an affine approximation

$$\sqrt{x} = \alpha \times x + \zeta + \theta$$

is computed, where α and ζ are determined by a linear approximation of the function and θ represents all errors committed by rounding and approximation. After that the affine operation Equation 2.6 is performed term wise. Details on α, ζ and θ computation can be found in [9].

While affine arithmetic is beneficial for linear operations it does not always give tighter bounds than interval arithmetic. Both types perform well in certain cases, and both are used for absolute roundoff error bounds computation by state-of-the-art tools.

2.3 State-of-the-Art in Absolute Error Estimation

In this section, we review state-of-the-art automated tools for absolute error estimation. The error definitions from Chapter 1 compute the difference between an original real-valued function $f(x)$ and its floating-point implementation $\hat{f}(\hat{x})$. Not every real number can be exactly represented by a floating-point number, thus, due to the necessary rounding $\hat{f}(\hat{x})$ becomes a highly discontinuous function. This makes bounding the difference $\left|f(x) - \hat{f}(\hat{x})\right|$ directly hard. A common way to overcome this difficulty is to replace $\hat{f}(\hat{x})$ with an abstraction using Equation 2.2.

Using this abstraction, we replace $\hat{f}(\hat{x})$ with the function $\tilde{f}(x, e, d)$ where x are input variables and e and d are the roundoff errors introduced for each floating-point operation. In general, for multivariate functions x, e and d are vector valued, but for ease of reading we will not use the vector notation in the equations. Note that our abstraction $\tilde{f}(x, e, d)$ is a real-valued function. Using this function we and all state-of-the-art tools approximate absolute errors as:

$$err_{abs} \leq \max_{x \in I, |e_i| \leq \epsilon_M, |d_i| \leq \delta} \left| f(x) - \tilde{f}(x, e, d) \right| \tag{2.8}$$

Since the input domain I is a range, applying range arithmetic to the expression to be maximized, we obtain absolute roundoff error bounds. However, this naive approach over-approximates the bounds when the function $f(x)$ is nonlinear. State-of-the-art tools attempt to reduce the over-approximation and compute tight bounds using different techniques, which we detail below.

2.3.1 Rosa

Rosa [2] computes absolute error bounds using a forward data-flow analysis and a combination of abstract domains. Note that the magnitude of the absolute roundoff error at an arithmetic operation depends on the magnitude of the operation's value, and this is in turn determined by the input parameter ranges. Thus, Rosa tracks two values for each intermediate abstract syntax tree (AST) node: a sound approximation of the real valued range and the worst-case absolute error. The data-flow analysis transfer function for errors uses the ranges to propagate errors from subexpressions and to compute the new roundoff error committed by the arithmetic operation in question. The following example illustrates how the ranges are used to propagate the error on a multiplication operation. Here the real range of a variable x is denoted as $[x]$ and the associated error term as err_x, thus $[\hat{x}] = [x] + err_x$.

$$\begin{aligned} \hat{x} \times \hat{y} &= ([x] + err_x)([y] + err_y) = \\ &= [x] \times [y] + [x] \times err_y + [y] \times err_x + err_x \times err_y + \rho \end{aligned} \tag{2.9}$$

where ρ is the new roundoff error. Thus, the first term is the ideal real-valued range and the remaining terms contribute to the error. It is easy to see, that the larger the real ranges $[x]$ and $[y]$ are, the larger the computed error is, so that it is important to estimate real ranges accurately.

One may think that just evaluating an expression in interval arithmetic and interpreting the width of the resulting interval as the error bound would be sufficient. While this is certainly a sound approach, it computes too pessimistic error bounds in general. This is especially true if we consider relatively large ranges on inputs; we cannot distinguish which part of the interval width is due to the input interval or due to accumulated roundoff errors. Hence, we need to compute ranges and errors separately.

For error propagation Rosa uses affine arithmetic. For bounding ranges it implements different range arithmetics with different accuracy-efficiency trade-offs: interval arithmetic, affine arithmetic and a combination of interval arithmetic with a nonlinear arithmetic decision procedure. We now explain the latter procedure in more detail.

Refining interval arithmetic with SMT. While both interval and affine arithmetic are reasonably fast methods, the over-approximation they give is often too large, especially if input ranges are not sufficiently small. Darulova and Kuncak [10] propose a technique to reduce the over-approximation by combining interval arithmetic with a nonlinear SMT (Satisfiability Modulo Theories) constraint solver.

For each range to be computed, first the technique computes an initial sound estimate of the range $I_0 = [l, u]$ with interval arithmetic. Note that lower and upper bounds are computed separately following the same procedure. The second step of the algorithm is to check whether the computed range I_0 is already tight. For the lower bound l, it means that the bound is increased by a small nonnegative value p (this value is a precision threshold) and then the Z3 SMT solver [11] is asked whether the expression's resulting range lies below this new bound $l+p$. If the solver reports affirmative, the lower bound l was already tight. Symmetrically, the check is performed for the upper bound. If the initial range I_0 is not tight, then it is taken as a starting point and narrowed down by a binary search. At each step of the search the technique checks whether the narrowed range is sound using the nonlinear *nlsat* decision procedure [12] within the Z3 SMT solver. The search terminates when one of the following conditions is met: Z3 returns unknown or times out, the difference between bounds on the subsequent steps is less than a specified precision threshold or the maximum number of iterations of the algorithm is reached.

The two last conditions serve to limit the amount of calls to the SMT solver, because these calls are fairly expensive. One more heuristic method to reduce the amount of calls implemented in Rosa is to call Z3 only every 10 arithmetic operations. The values for these three parameters can be changed by user. The current configuration has empirically resulted in a good accuracy-performance trade-off.

2.3.2 Fluctuat

Fluctuat [3] is an abstract interpreter which separates errors similarly to Rosa and which uses affine arithmetic for computing both the ranges of variables and for the error bounds. In order to reduce the over-approximations introduced by affine arithmetic for nonlinear operations, Fluctuat uses interval subdivision. The user can designate up to two variables in the program whose input ranges will be subdivided into intervals of equal width. The analysis is performed separately on each subinterval and the overall error is then the maximum error over all subintervals. Interval

subdivision increases the runtime of the analysis significantly, especially for multivariate functions, and the choice of which variables to subdivide and by how much is usually not straight-forward.

Another tool Gappa [13] also uses forward dataflow analysis, but combines it with interval arithmetic instead of affine.

2.3.3 FPTaylor

FPTaylor [4] takes a different approach, unlike Daisy and Fluctuat it formulates the roundoff error bounds computation as an optimization problem, where the absolute error expression $|f(x) - \hat{f}(\hat{x})|$ is maximized, subject to interval constraints on its parameters. Due to the discrete nature of floating-point arithmetic, FPTaylor optimizes the continuous, real-valued abstraction (Equation 2.8). However, this expression is still too complex and features too many variables for optimization procedures in practice, resulting in bad performance as well as bounds which are too coarse to be useful (see subsection 5.4.2 for our own experiments).

FPTaylor introduces the Symbolic Taylor approach, where the objective function of Equation 2.8 is simplified using a first order Taylor approximation with respect to e and d:

$$\tilde{f}(x, e, d) = \tilde{f}(x, 0, 0) + \sum_{i=1}^{k} \frac{\partial \tilde{f}}{\partial e_i}(x, 0, 0) e_i + R(x, e, d) \tag{2.10}$$

where

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

where $y_1 = e_1, \ldots, y_k = e_k, y_{k+1} = d_1, \ldots, y_{2k} = d_k$ 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.

The approach is symbolic in the sense that the Taylor approximation is taken wrt. e and d only and x remains a symbolic argument. Thus, $f(x, 0, 0)$ represents the function point where all inputs x remain symbolic and no roundoff errors are present, i.e. $e = d = 0$ and $f(x, 0, 0) = f(x)$, i.e. the real-valued ideal function. Choosing $e = d = 0$ as the point at which to perform the Taylor approximation and replacing e_i with its upper bound ϵ_M reduces the initial optimization problem to:

$$err_{abs} \leq \epsilon_M \max_{x \in I} \sum_{i=1}^{k} \left| \frac{\partial \tilde{f}}{\partial e_i}(x, 0, 0) \right| + M_R \tag{2.11}$$

where M_R is an upper bound for the error term $R(x, e, d)$ (more details can be found in [4]). FPTaylor uses interval arithmetic to estimate the value of M_R as the term is second order of e and thus small in general.

To solve the optimization problem in Equation 2.11, FPTaylor uses rigorous branch-and-bound optimization based on interval arithmetic, which is implemented in the

global optimization tool Gelpia [14]. Alternatively, FPTaylor works with several external global optimization tools and libraries, such as NLopt [15]. Although algorithms implemented in NLopt are not rigorous, they are fast and can be used for obtaining preliminary results before running a slower optimization technique. Branch-and-bound is also used to compute the resulting real function range $\max_{x \in I} |f(x)|$, which is needed for instance to compute relative errors or to check for overflows.

Real2Float [16], another tool, takes the same optimization-based approach, but uses semi-definite programming for the optimization itself.

Chapter 3

Daisy

We have implemented all techniques, which we are aiming to combine and compare in the tool called Daisy [5]. Daisy, a successor of Rosa, is a source-to-source compiler from real arithmetic computations to a finite-precision implementation.

Before this project, Daisy (and the other tools described in Chapter 2) computed sound worst-case absolute error bounds and used them for relative error computation (when possible). Daisy is internally structured in several phases:

Frontend. As input, Daisy takes a program written in a real-valued specification language (a functional subset of Scala) consisting of one or more functions. An example function looks like this:

```
def bspline0(u: Real): {Real} = {
      require(0 <= u && u <= 1)
      (1 - u) * (1 - u) * (1 - u) / 6.0
}
```

Daisy parses the input program and builds an abstract syntax tree (AST) for the arithmetic expression in the function body.

Note that the input program is not executable. The specification describes the real-valued function, where all input variables and the resulting value have data type **`Real`**.

Specification Phase. After the input program has been parsed, Daisy extracts preconditions. In the precondition (the **`require`** clause) the user provides the ranges of all input variables. This information is needed for the next phase.

Range Error Analysis Phase. The approach for sound roundoff error estimation is adapted from Rosa, which splits the analysis into two steps:

- range analysis: compute real-valued ranges for all intermediate expressions
- error analysis: using the previously computed ranges, propagate errors from sub-expressions and compute the new worst-case roundoffs

Such a separation is necessary to compute tight error bounds for ranges of inputs.

Similarly to Rosa, Daisy tracks roundoff errors using affine arithmetic, and for range analysis it offers the choice between interval, affine arithmetic and Rosa's procedure, where initially computed intervals are refined using a nonlinear decision procedure.

For different floating-point types, the roundoff error is different and the user can choose the uniform precision for the resulting program. By default, Daisy uses double floating-point precision. During the Range Error Analysis phase Daisy computes a sound worst-case absolute roundoff error, with respect to the selected floating-point precision. In order to make sure that it does not introduce roundoff errors internally, all computations are done using infinite-precision rationals based on big integers (BigInt in Scala). At the end of the phase, Daisy reports the computed roundoff error alongside with the resulting real-valued function range.

Code Generation Phase. As output, Daisy generates a finite-precision implementation of the input program, thus it is a transformer between high-level languages. It generates executable code for the selected floating-point precision, currently in Scala, but generation of some other high-level language code (e.g. C/C++) would be a straight-forward extension.

3.1 Extensions

As part of this thesis, we have implemented several new phases, each providing an alternative to the Range Error Analysis Phase to compute roundoff errors. Note that each of the phases is used separately. We provide here a short overview of the extensions and give more details in the following sections.

- *Taylor Error Phase* (Chapter 4). In this phase, we have reimplemented the optimization based approach of FPTaylor for the computation of absolute roundoff errors.
- *Relative Through Absolute Error Phase*. During this phase, we compute relative error bounds using a combination of interval subdivision and state-of-the-art absolute error computation as in Equation 1.3 (Chapter 5).
- *Relative Error Phase* (Chapter 5). In this phase, we have implemented a direct relative error computation, without taking an intermediate step for computing absolute errors.

Chapter 4

Optimization Based Approach in Daisy

As we mentioned before, Daisy (as well as Rosa and Fluctuat) uses forward dataflow analysis for the estimation of error bounds and FPTaylor uses an optimization-based approach. Since we want to extend FPTaylor's approach to the direct computation of relative errors, we need to implement it in Daisy. But before going into relative errors we have reimplemented FPTaylor's approach for computing absolute errors with some modifications. While computing absolute errors is not the main focus of the thesis, it provides an interesting comparison with absolute errors computed using FPTaylor.

We build the expression to be optimized in a similar way as FPTaylor does (subsection 2.3.3), thus the expression is a first order Taylor approximation similarly to Equation 2.11 with a difference that we do not pull terms e_i out of the sum. The second order remainder R is expected to be small, so that we use interval arithmetic to compute a bound on our M_R. Experiments have shown that for our benchmarks the remainder term is indeed small (on average, 14 orders of magnitude less than the derivative terms). We note that even if the remainder term is not small, the approach remains sound. To bound the first order terms $\frac{\partial \tilde{g}}{\partial e_i}$, we need a sound optimization procedure to maintain overall soundness, which limits the available choices significantly.

FPTaylor uses the global optimization tool Gelpia [14], which internally uses a branch-and-bound based algorithm. Unfortunately, we found it difficult to integrate because of its custom interface. Furthermore, we observed Gelpia's unpredictable behavior in our experiments (e.g. nondeterministic crashes and substantially varying running times for repeated runs on the same expression).

Instead, we use Rosa's approach which combines interval arithmetic with a solver-based refinement (section 2.3.1). Our approach is parametric in the solver and we experiment with Z3 [11] and dReal [17]. Both support the SMT-lib interface, provide rigorous results, but are based on fundamentally different techniques. Z3 implements a complete decision procedure for nonlinear arithmetic (in the nlsat solver [12]), whereas dReal implements the framework of δ-complete decision procedures, which is based on a branch-and-bound algorithm internally.

Benchmark	Under-approx	FPTaylor	opt based Z3	opt based dReal
Univariate benchmarks				
bspline0	2.79e-17	1.39e-16	**1.19e-16**	**1.19e-16**
bspline1	1.64e-16	**5.15e-16**	6.51e-16	6.51e-16
bspline2	1.51e-16	**5.43e-16**	5.82e-16	5.82e-16
bspline3	5.07e-17	**8.33e-17**	1.11e-16	1.11e-16
sine	2.56e-16	**5.55e-16**	6.54e-16	6.54e-16
sineOrder3	3.32e-16	9.52e-16	**8.00e-16**	**8.00e-16**
sqroot	1.86e-13	**3.61e-13**	3.97e-13	3.97e-13
Multivariate benchmarks				
doppler	5.87e-14	**1.58e-13**	1.74e-13	1.72e-13
himmilbeau	4.84e-13	**1.32e-12**	1.42e-12	1.42e-12
invPendulum	2.02e-14	**3.84e-14**	4.44e-14	4.44e-14
jet	4.13e-12	**1.34e-11**	2.49e-11	2.19e-11
kepler0	3.81e-14	**9.44e-14**	1.15e-13	1.18e-13
kepler1	1.13e-13	**3.56e-13**	4.99e-13	4.96e-13
kepler2	4.59e-13	**1.95e-12**	2.28e-12	2.54e-12
rigidBody1	1.79e-13	**3.86e-13**	5.08e-13	5.08e-13
rigidBody2	1.96e-11	**5.23e-11**	6.48e-11	6.48e-11
traincar_state8	5.88e-15	1.35e-14	**1.33e-14**	**1.33e-14**
traincar_state9	6.06e-15	1.22e-14	**1.20e-14**	**1.20e-14**
turbine1	6.01e-15	**2.31e-14**	2.80e-14	2.82e-14
turbine2	1.01e-14	**2.55e-14**	3.67e-14	3.67e-14
turbine3	4.01e-15	**1.24e-14**	1.65e-14	1.65e-14

TABLE 4.1: Absolute roundoff error bounds computed with FPTaylor and the optimization-based approach in Daisy

Note that the queries we send to both solvers are (un)satisfiability queries, and not optimization queries. For the nonlinear decision procedure this is necessary as it is not suitable for direct optimization, but the branch-and-bound algorithm in dReal is performing optimization internally. The reason for our roundabout approach for dReal is that while the tool has an optimization interface, it uses a custom input format and is difficult to integrate. We expect this to affect mostly performance, however, and not accuracy.

4.1 Experimental Comparison

We compare our implementation of the optimization based approach in Daisy against FPTaylor.

All experiments are performed in double floating-point precision (the precision FPTaylor supports), although all techniques in Daisy are parametric in the precision. The experiments were performed on a desktop computer running Debian GNU/Linux 8 64-bit with a 3.40GHz i5 CPU and 7.8GB RAM.

The benchmarks *bsplines*, *doppler*, *jetengine*, *rigidBody*, *sine*, *sqrt* and *turbine* are nonlinear functions from [2]; *invertedPendulum* and the *traincar* benchmarks are linear embedded examples from [18]; and *himmilbeau* and *kepler* are nonlinear examples

Benchmark	FPTaylor	opt based Z3	opt based dReal
Univariate benchmarks			
bsplines	16.91s	29.81s	18.78s
sines	10.89s	45.13s	63.76s
sqroot	5.48s	19.01s	11.07s
Multivariate benchmarks			
doppler	10s	2m 33s	3m 42s
himmilbeau	8.77s	35.70s	29.94s
invPendulum	6.25s	5.78s	2.92s
jet	18.87s	9m 41s	14m 8s
kepler	2m 25s	5m 41s	11m 43s
rigidBody	10.47s	49.80s	26.61s
traincar	54.97s	32.92s	3m 25s
turbine	31.44s	1m 50s	5m 34s
total	5m 19s	23m 23s	41m 4s

TABLE 4.2: Comparison of running times for computing absolute roundoff error bounds using FPTaylor and the optimization-based approach of Daisy

from the Real2Float project [16].

Table 4.1 shows absolute error bounds computed with FPTaylor and Daisy's optimization based approach. Here and further in the thesis, bold marks the best result, i.e. the tightest computed bound. The column 'Under-approx' gives an (unsound) under-approximation for the absolute error bound obtained with dynamic evaluation on 100000 inputs. The under-approximation provides an idea of the order of magnitude of the true errors. Column 'FPTaylor' contains absolute error bounds computed by FPTaylor; columns 'Opt based Z3' and 'Opt based dReal' show the error bounds for the optimization based approach implemented in Daisy evaluated with solvers Z3 and dReal respectively. For both solvers a timeout for each (un)satisfiability query is 1 second.

Comparing absolute error bounds computed by FPTaylor and Daisy with the under-approximation we see that the values are close, which means bounds computed by both FPTaylor and Daisy are quite tight. We noticed that for most benchmarks the error bounds computed by FPTaylor are tighter, while for a few benchmarks *bspline0*, *sineOrder3* and *traincar_state8(-9)* Daisy's implementation (with either of the two available solvers) provides tighter bounds. Interestingly, for the majority of benchmarks (except *bspline3* and *kepler0*) error bounds reported in three last columns of Table 4.1 are very close to each other. Since the difference is so small, we can state that by replacing Gelpia with the combination of interval arithmetic and nonlinear decision procedure we can still achieve adequately good results in accuracy.

Table 4.2 provides the running times for FPTaylor and Daisy's optimization-based approach. Note that the running times are cumulated for benchmarks contained in one program (e.g. times for *sine* and *sineOrder3* are summed in *sines*).

Daisy's running times are noticeably greater for all benchmarks except for linear *invertedPendulum*, where a combination of Daisy with dReal outperforms FPTaylor by more than 2 times and *traincar*, where Daisy together with Z3 shows the smallest running time. We suspect that the difference in performance is due to the fact that in the interval refinement procedure we call the solvers repeatedly for each term, while FPTaylor only the queries global optimization for each term once.

Chapter 5

Bounding Relative Errors

The main goal of this thesis is to investigate how today's sound approaches for computing absolute errors fare for bounding relative errors and whether it is possible and advantageous to compute relative errors directly (and not via absolute errors). In this section, we first concentrate on obtaining tight bounds in the presence of nonlinear arithmetic, and postpone a discussion of the orthogonal issue of division by zero to the next section. Thus, we assume for now that the range of the function for which we want to bound relative errors does not include zero, i.e. $0 \notin f(x)$ and $0 \notin \tilde{f}(\tilde{x})$, for $x, \tilde{x}$ within some given input domain. We furthermore consider f to be an arithmetic expression that does not contain transcendental functions and is specified as straight-line code in the input program. Conditionals and loops have been shown to be challenging [10] even for absolute errors and we thus leave their proper treatment for future work. We consider function calls to be an orthogonal issue; they can be handled by inlining, thus reducing to straight-line code, or require suitable summaries in postconditions, which is also one of the motivations for this work.

We implement interval subdivision (section 5.1), which is a measure to reduce over-approximation and apply it to the state-of-the-art methods for estimation of absolute error bounds. We then use the potentially improved absolute errors to compute relative error bounds. We furthermore extend FPTaylor's approach to computing relative errors directly (section 5.2) and provide implementation details (section 5.3). We experimentally evaluate different configurations of implemented techniques and combinations of techniques on a set of standard benchmarks (section 5.4) and show based on the experiments that the direct approach for computing relative error bounds scales better than the one via absolute errors even with interval subdivisions applied.

5.1 Interval Subdivision

The over-approximation committed by static analysis techniques grows in general with the width of the input intervals, and thus with the width of all intermediate ranges. Intuitively, the worst-case error which we consider is usually achieved only

for a small part of the domain, over-approximating the error for the remaining inputs. Additionally the domain where worst-case errors are obtained may be different at each arithmetic operation.

Thus, by subdividing the input domain we can usually obtain tighter error bounds. Fluctuat has applied interval subdivision to absolute error estimation, but we are not aware about a study of its effectiveness for relative errors.

We apply subdivision to input variables and combine it with state-of-the-art absolute error computation, thus we compute:

$$\max_{k\in[1\ldots m]}\left(\frac{\max_{x_j\in I_{jk}}\left|f(x)-\tilde{f}(x,e,d)\right|}{\min_{x_j\in I_{jk}}|f(x)|}\right) \tag{5.1}$$

where m is an number of subdivisions for each input interval. That is, for multivariate functions, we subdivide the input interval for each variable and maximize the error over the Cartesian product. Clearly, the analysis running time is exponential in the number of variables. While Fluctuat limits subdivisions to two user-designated variables and a user-defined number of subdivisions each, we choose to limit the total number of analysis runs by a user-specified parameter p. First, we sort the input domains, larger domains come first. Then we compute potential number of analysis runs p_i if the i-th domain is subdivided. That is, given m (the desired number of subdivisions for each variable) and n (the number of input variables), p_i is computed as:

$$p_i = m^i + (n-i), \quad i \in [1..n] \tag{5.2}$$

If computed value is smaller than the specified limit $p_i \leq p$, i-th interval is subdivided m times and algorithm repeats for $(i+1)$-th input domain. If $p_i > p$, the i-th interval and all further $n-i$ intervals remain undivided.

5.2 Bounding Relative Errors Directly

Another strategy we explore is to compute relative errors directly, without taking the intermediate step through absolute errors. That is, we extend FPTaylor's optimization based approach and maximize the relative error expression using the floating-point abstraction from Equation 2.2:

$$\max|\tilde{g}(x,e,d)| = \max_{x\in I,|e_i|\leq\epsilon_M,|d_i|\leq\delta}\left|\frac{f(x)-\tilde{f}(x,e,d)}{f(x)}\right| \tag{5.3}$$

The hope is to preserve more correlations between variables in the nominator and denominator and thus obtain tighter and more informative relative error bounds.

We call the evaluation of Equation 5.3 without simplifications the *naive approach*. While it has been mentioned previously that this approach does not scale well [4], we include it in our experiments nonetheless, as we are not aware of any concrete

results actually being reported. That said, as expected, the naive approach returns error bounds which are so large that they are essentially useless (see section 5.4 for our experiments).

We thus simplify $\tilde{g}(x, e, d)$ by applying the Symbolic Taylor approach introduced by FPTaylor [4]. As before, we take the Taylor approximation around the point $(x, 0, 0)$, so that the first term of the approximation is zero as before: $\tilde{g}(x, 0, 0) = 0$. We obtain the following optimization problem:

$$\max_{x \in I, |e_i| \leq \epsilon_M, |d_i| \leq \delta} \sum_{i=1}^{k} \left| \frac{\partial \tilde{g}}{\partial e_i}(x, 0, 0) e_i \right| + M_R \tag{5.4}$$

where M_R is an upper bound for the remainder term $R(x, e, d)$. Unlike in Equation 2.11 we do not pull the factor e_i for each term out of the absolute value, as we plan to compute tight bounds for mixed-precision in the future, where the upper bounds on all e_i are not all the same (this change does not affect the accuracy for uniform precision computations though).

Note that for computing upper bounds we use exactly the same techniques as for the optimization based approach for absolute error bounds in Daisy (Chapter 4). That is, we apply the SMT solver-based refinement of intervals for the derivative terms $\frac{\partial \tilde{g}}{\partial e_i}(x, 0, 0) e_i$ and interval analysis for the remainder term M_R.

One more strategy that can potentially lead to tighter computed bounds is to combine the optimization-based approach with interval subdivision. That is, we compute:

$$\max_{k \in [1 \ldots m]} \left(\max_{x_j \in I_{jk}} \left| \frac{f(x) - \tilde{f}(x, e, d)}{f(x)} \right| \right) \tag{5.5}$$

where m is an number of subdivisions for each input interval and the subdivision is performed as described in section 5.1.

5.3 Implementation

We implement all the described techniques in the tool Daisy [5]. Daisy is parametric in the *approach* (naive, forward data-flow analysis or optimization based), the *solver* used (Z3 or dReal) and the number of *subdivisions* (including none). Any combinations of these three orthogonal choices can be run by simply changing Daisy's input parameters.

Daisy has been implemented in Scala, therefore all extensions (see section 3.1) are implemented in Scala as well. Some features of Daisy require additional software to be installed. Using the solver-based intervals refinement procedure (section 2.3.1) requires Z3 and dReal solvers to be installed, for the dynamic evaluation an external library MPFR [19] is needed.

5.3.1 Simplifications

In many cases the derivative expressions that we obtain by applying Taylor expansion remain complex. This might affect the running time of solvers (and thus also potentially the accuracy of the error bounds). We observed that the solvers do not necessarily perform arithmetic simplifications. Therefore we have implemented them inside Daisy. In general, there are many possible arithmetic simplification rules, we restrain ourselves to a limited set of rules, which we base on the observations of the unsimplified expressions. Daisy performs the simplification recursively on the complete AST until no rules can be applied further to any of the sub-expressions.

We split the simplifications into two groups: 'initial' and 'advanced'. The 'initial' group contains the rules for the simplifications that are performed often; mostly they are for arithmetic expressions with two operands:

- $e + 0 \to e$, $e - 0 \to e$ (same for $-e$)
- $e \times 1 \to e$, $e \times 0 \to 0$ (same for $-e$)
- $e/1 \to e$, $0/e \to 0$
- $e/e \to 1$, $-e/e \to -1$
- $e + e \to 2e$
- $-(-e) \to e$
- $-0 \to 0$
- $c_1 \circ c_2 \to c_3$, where c_1, c_2, c_3 are constants, $\circ \in \{+, -, \times, /\}$ and c_3 is computed as result of real arithmetic operation. For example, if $c_1 = 2, c_2 = 3$ and the operation $\circ = \times$ then $c_3 = 2 \times 3 = 6$
- $\frac{1}{e_1/e_2} \to \frac{e_2}{e_1}$

where e (or e_i) denotes any subexpression, i.e. it can be both a leaf of the AST or a subtree, c (or c_i) denotes a constant. These rules are applied to every intermediate expression, e.g. deriving the expression for Equation 5.4 we apply 'initial' simplifications to every derivative term $\frac{\partial \tilde{g}}{\partial e_i}(x, 0, 0)e_i$ and every second order derivative term $\frac{\partial^2 \tilde{g}}{\partial e_i \partial e_j}(x, e, d)e_i e_j$.

The 'advanced' group contains the rules for simplifications that are performed only once the complete expression Equation 5.4 is built. These rules mostly combine several levels of nodes of the AST:

- $e \times e \to e^2$, $e^n \times e \to e^{n+1}$
- $e^n \times e^m \to e^{n+m}$
- $(e^n)^m = e^{nm}$

- $(e_1 \times e_2) + (e_1) \rightarrow e_1 \times (e_2 + 1)$
- $(e_1 \times e_2) + (e_1 \times e_3) \rightarrow e_1 \times (e_2 + e_3)$ and symmetric rules if the repeating term is e_2 or e_3
- $((c_1 \times e) \times \ldots) \times c_2 \rightarrow c_3 \times e$, where c_1, c_2 are constants and $c_3 = c_1 \times c_2$, i.e. multiply the constants that belong to different leafs of the (sub-)tree, if the (sub-)tree contains only multiplications
- $\frac{e_1}{e_2} \times \frac{e_3}{e_4} \rightarrow \frac{e_1 e_3}{e_2 e_4}$
- $\frac{e_1}{e_2} \times e_3 \rightarrow \frac{e_1 e_3}{e_2}$ and $\frac{1}{e_2} \times e_3 \rightarrow \frac{e_3}{e_2}$
- $(e_1 \times c) \times e_1 \rightarrow c \times e_1^2$
- $(c \times e^n) \times e \rightarrow c \times e^{n+1}$
- $\frac{e_1/e_2}{e_3/e_4} \rightarrow \frac{e_1 e_4}{e_2 e_3}$
- $\frac{e_1 e_2 \ldots}{e_1 e_3 \ldots} \rightarrow \frac{e_2 \ldots}{e_3 \ldots}$, i.e. cancel repeating factors in nominator and denominator (e_1 in this example)

Applying these simplifications in some cases (benchmarks *bspline0*, *bspline2*, *bspline3*) resulted in that the simplified term only contained $\pm\epsilon_M$. Daisy does not call the SMT solver on simple expressions (leafs of the AST: variable, e_i and d_i from Equation 5.4), thus these simplifications reduce the amount of calls to SMT solver and hence also reduce analysis running time.

5.4 Experimental Evaluation

The experimental evaluation contains several parts. First we compare relative error bounds obtained with different subdivision parameters to find a good default configuration (subsection 5.4.1). Note that we only subdivide input intervals for evaluating the derivative terms $\frac{\partial \tilde{g}}{\partial e_i}(x, 0, 0)e_i$, we evaluate the remainder term M_R on the undivided domain. We then compare different techniques implemented within Daisy (using the default configuration found) against FPTaylor and the forward dataflow analysis approach from Daisy before the current project as representatives of state-of-the-art tools (subsection 5.4.2). We also investigate how the absence of denormals in the floating-point function abstraction affects the accuracy and running times of the optimization based approach for absolute and relative error bounds in Daisy (subsection 5.4.3). Finally we show the improved scalability of the direct relative error computation with respect to the computation via absolute errors (subsection 5.4.4).

We perform all our experiments on the standard benchmark set that we have described in section 4.1. Similarly in all tables bold marks the best result, i.e. tightest

Benchmark	# of vars	$p = 50$			$p = 100$		
		$m = 2$	$m = 5$	$m = 8$	$m = 2$	$m = 5$	$m = 8$
Univariate benchmarks							
bspline0	1	3.00e-15	3.00e-15	3.00e-15	3.00e-15	3.00e-15	3.00e-15
bspline1	1	3.22e-15	3.22e-15	3.22e-15	3.22e-15	3.22e-15	3.22e-15
bspline2	1	8.92e-16	8.92e-16	8.92e-16	8.92e-16	8.92e-16	8.92e-16
bspline3	1	6.66e-16	6.66e-16	6.66e-16	6.66e-16	6.66e-16	6.66e-16
sine	1	7.66e-16	7.66e-16	7.66e-16	7.66e-16	7.66e-16	7.66e-16
sineOrder3	1	8.94e-16	8.94e-16	8.94e-16	8.94e-16	8.94e-16	8.94e-16
sqroot	1	1.02e-15	1.02e-15	1.02e-15	1.02e-15	1.02e-15	1.02e-15
Multivariate benchmarks							
doppler	3	1.93e-13	1.93e-13	1.93e-13	1.93e-13	1.93e-13	1.93e-13
himmilbeau	2	7.83e-14	1.10e-14	7.05e-15	7.83e-14	1.10e-14	**5.80e-15**
invPendulum	4	1.21e-15	1.21e-15	1.21e-15	1.21e-15	1.21e-15	1.21e-15
jet	2	**4.47e-15**	4.92e-15	6.03e-15	**4.47e-15**	4.92e-15	**4.47e-15**
kepler0	6	3.66e-13	6.63e-13	**1.63e-15**	3.80e-13	7.05e-13	7.33e-13
kepler1	4	8.53e-12	8.10e-13	**2.85e-13**	8.33e-12	8.10e-13	4.63e-13
kepler2	6	9.65e-11	1.46e-11	8.58e-12	9.65e-11	1.46e-11	**5.61e-12**
rigidBody1	3	9.78e-16	9.78e-16	9.78e-16	9.78e-16	9.78e-16	9.78e-16
rigidBody2	3	2.21e-15	2.21e-15	2.21e-15	2.21e-15	2.21e-15	2.21e-15
traincar_state8	14	7.67e-14	7.67e-14	7.67e-14	7.67e-14	7.67e-14	7.67e-14
traincar_state9	14	3.45e-14	3.45e-14	3.45e-14	3.45e-14	3.45e-14	3.45e-14
turbine1	3	2.06e-15	2.06e-15	2.06e-15	2.06e-15	2.06e-15	2.06e-15
turbine2	3	4.12e-15	4.12e-15	4.12e-15	4.12e-15	4.12e-15	4.31e-15
turbine3	3	1.91e-14	1.91e-14	1.91e-14	1.91e-14	1.91e-14	1.91e-14

TABLE 5.1: Comparison of different configurations for subdivision

computed error bound, for each benchmark. When the values are the same for all techniques compared in one table, we do not mark the best result.

To evaluate the accuracy and performance of the different approaches for the case when no division by zero occurs, we modify the standard input domains of the benchmarks whenever necessary such that the function ranges do not contain zero and all tools can thus compute a non-trivial relative error bound. We show the benchmarks in Appendix B.

5.4.1 Finding Default Configuration for Subdivision

Our subdivision approach is parametric in the amount of subdivisions for each interval m and the total amount of optimizations p. In order to define good defaults for the parameters, we have executed multiple tests. The experiments are done for the optimization-based approach for several subdivision values $m = 2, 5, 8$ in combination with two different values for the limit for the total analysis runs number ($p = 50, 100$). We used the Z3 solver with a timeout of 1 second for testing all configurations.

Table 5.1 shows the relative error bounds for different subdivision values, Table 5.2 gives running times for these configurations. The column '# of vars' shows

Benchmark	$p = 50$			$p = 100$		
	$m = 2$	$m = 5$	$m = 8$	$m = 2$	$m = 5$	$m = 8$
Univariate benchmarks						
bspline0	6s	12s	18s	7s	12s	17s
bspline1	8s	16s	24s	9s	16s	24s
bspline2	10s	18s	26s	11s	18s	27s
bspline3	2s	6s	8s	3s	5s	9s
sine	1m 3s	1m 13s	1m 26s	1m 19s	1m 14s	1m 41s
sineOrder3	5s	10s	15s	5s	10s	15s
sqroot	14s	24s	36s	15s	25s	35s
Multivariate benchmarks						
doppler	2m 41s	4m 6s	2m 58s	2m 44s	4m 7s	7m 17s
himmilbeau	2m 9s	8m 34s	6m 15s	2m 14s	8m 30s	14m 59s
invPendulum	52s	1m 7s	25s	55s	1m 6s	2m 40s
jet	351s	58m 45s	45m 31s	34m 1s	47m 57s	1h 22m 35s
kepler0	12m 47s	11m 25s	1m 40s	32m 39s	11m 49s	21m 34s
kepler1	15m 54s	16m 54s	5m 8s	15m 29s	17m 15s	43m 8s
kepler2	12m 7s	15m 46s	5m 32s	41m 41s	1h 16m 50s	430s
rigidBody1	30s	1m 17s	29s	30.74s	1m 16s	2m 55s
rigidBody2	1m 8s	2m 34s	57s	1m 7s	2m 35s	5m 57s
traincar_state8	8m 10s	5m 57s	2m 13s	15m 39s	5m 57s	14m 52s
traincar_state9	7m 28s	5m 12s	1m 54s	14m 20s	5m 26s	13m 33s
turbine1	2m 48s	4m 54s	2m 59s	2m 27s	4m 59s	153s
turbine2	4m 21s	16m 8s	6m 7s	3m 52s	15m 4s	31m 57s
turbine3	3m 17s	7m 9s	3m 24s	2m 57s	7m 14s	14m 34s
total	1h 46m 49s	2h 42m 26s	**1h 29m 7s**	2h 52m 45s	3h 32m 46s	5h 11m 14s

TABLE 5.2: Comparison of running times for different configurations for subdivision

the amount of input variables for each benchmark. Columns '2', '5', '8' show the bounds computed with $m = 2, 5$ and 8 subdivisions for input intervals respectively.

We noticed that, perhaps surprisingly, a more fine-grained subdivision does not gain accuracy for most of the benchmarks. For those benchmarks where we observed different error bounds three out of five tightest bounds (*himmilbeau*, *jet* and *kepler2*) were computed with the configuration $m = 8, p = 100$. The second best choice in terms of accuracy is $m = 8, p = 50$, when the bounds for *himmilbeau*, *jet* and *kepler* differ from the tightest computed bounds only insignificantly.

One may think that the greater the number of subdivisions m is, the larger are the running times. Indeed it is the case when *all* input intervals are subdivided. Recall that, to limit this effect we introduced the total amount of optimizations p. For the univariate benchmarks the influence of p is not noticeable, since the total amount of optimizations is equal to the amount of subdivisions in this case, and is below the limit p. For multivariate benchmarks we observe that p limits the amount of variables where intervals are subdivided and hence limits runtime. Thus, the configuration with the *greatest* tested *subdivision* value 8 has the *smallest* total *running time* when $p = 50$.

Keeping in mind the accuracy-performance trade-off, we select the following default configuration: for univariate benchmarks $m = 2$ and $p = 50$; for multivariate benchmarks $m = 8$ and $p = 50$.

5.4.2 Comparison with State-of-the-Art Tools

We compare the different strategies for relative error computation on a set of standard benchmarks with representatives of state-of-the-art: FPTaylor and the forward dataflow analysis approach from Daisy. We do not compare with Fluctuat and Gappa directly as the underlying error estimation technique based on forward analysis with affine arithmetic is very similar to Daisy's. We rather choose to implement interval subdivision within Daisy.

Table 5.3 shows the relative error bounds computed with the different techniques and tools, and Table 5.4 the corresponding analysis times. The column 'Under-approx' is an unsound relative error bound obtained with dynamic evaluation on 100000 inputs. The under-approximation gives an idea of the order of magnitude of the true roundoff errors. Thus, if the difference between the under-approximation and error values computed by the analysis is several orders of magnitude, the analysis likely committed a big over-approximation. The 'Naive approach' column presents relative error bounds computed using the naive approach. We notice that the exponents of the computed bounds are mostly positive, meaning the values are huge. This confirms that an approximation of the relative error expression is indeed necessary.

The last four columns show the error bounds when relative errors are computed directly using the optimization based approach from section 5.2, with the Z3 and dReal solvers and with and without subdivisions. For subdivisions, we use $p = 50$ and $m = 2$ for univariate functions and $p = 50$ and $m = 8$ for multivariate, as we defined them to be good defaults (see subsection 5.4.1).

For most of the benchmarks we find that the direct evaluation of relative errors computes tightest error bounds with acceptable analysis times. Furthermore, for most benchmarks Z3, resp. its nonlinear decision procedure, is able to compute slightly tighter error bounds, but for three of our benchmarks dReal performs significantly better, while the running times are comparable.

We note that interval subdivision has a limited effect when combined with the direct relative error computation, and can, due to timeouts, actually decrease accuracy, while also increasing the running time significantly.

Comparing against state-of-the-art techniques (columns Daisy and FPTaylor, which compute relative errors via absolute errors), we notice that their results are sometimes several orders of magnitude less accurate than the direct relative error computation (e.g. six orders of magnitude for the *bspline3* and *doppler* benchmarks).

The column 'fwd analysis+subdiv' shows relative errors computed via absolute errors using the forward analysis with subdivision (with the same parameters as

Benchmark	Under-approx	Daisy	FPTaylor	Naive approach	fwd analysis + subdiv	Opt.-based approach for direct relative error. comp.			
						Z3	dReal	Z3 + subdiv	dReal + subdiv
Univariate benchmarks									
bspline0	1.46e-15	4.12e-13	4.26e-13	5.11e+02	7.44e-14	**3.00e-15**	**3.00e-15**	**3.00e-15**	**3.00e-15**
bspline1	7.91e-16	**2.54e-15**	3.32e-15	4.16e-01	5.32e-15	3.22e-15	3.22e-15	3.22e-15	3.22e-15
bspline2	2.74e-16	1.11e-15	1.16e-15	5.22e-01	1.61e-15	**8.92e-16**	9.76e-16	**8.92e-16**	**8.92e-16**
bspline3	5.49e-16	2.46e-10	3.07e-10	5.12e+05	5.23e-11	**6.66e-16**	**6.66e-16**	**6.66e-16**	**6.66e-16**
sine	2.84e-16	8.94e-16	8.27e-16	4.45e-01	1.39e-15	**7.66e-16**	**7.66e-16**	**7.66e-16**	**7.66e-16**
sineOrder3	3.65e-16	1.04e-15	1.10e-15	1.39e-01	1.99e-15	**8.94e-16**	**8.94e-16**	**8.94e-16**	**8.94e-16**
sqroot	4.01e-16	1.04e-15	1.21e-15	1.02e+00	2.20e-15	**1.02e-15**	**1.02e-15**	**1.02e-15**	**1.02e-15**
Multivariate benchmarks									
doppler	1.06e-15	2.08e-04	6.13e-07	2.09e+08	2.60e-05	**1.93e-13**	**1.94e-13**	**1.93e-13**	**1.94e-13**
himmilbeau	8.46e-16	6.55e-13	7.89e-13	6.69e+02	9.81e-15	6.54e-13	1.98e-15	7.05e-15	**1.99e-15**
invPendulum	3.74e-16	2.09e-11	2.48e-11	1.64e+00	1.22e-11	**1.21e-15**	**1.35e-15**	**1.21e-15**	**1.52e-15**
jet	1.45e-15	9.26e-15	7.53e-15	3.87e+00	1.40e-13	**4.47e-15**	5.12e-15	6.03e-15	6.51e-15
kepler0	4.39e-16	1.31e-12	1.64e-12	2.16e+03	3.63e-12	3.97e-12	2.39e-15	**1.63e-15**	2.64e-15
kepler1	7.22e-16	2.17e-11	2.59e-11	7.93e+04	8.70e-13	3.80e-11	**1.29e-15**	2.85e-13	1.71e-15
kepler2	5.28e-16	4.01e-10	5.65e-15	4.09e+05	1.35e-11	4.56e-10	2.42e-15	8.58e-12	**2.26e-15**
rigidBody1	4.49e-16	8.77e-11	1.14e-10	1.55e+00	2.50e-11	**9.78e-16**	1.27e-15	**9.78e-16**	1.46e-15
rigidBody2	5.48e-16	3.91e-12	4.73e-12	5.14e+03	1.77e-12	**2.21e-15**	2.33e-15	**2.21e-15**	2.96e-15
traincar_state8	2.72e-15	2.16e-13	2.69e-13	2.91e+02	2.16e-13	**7.67e-14**	2.72e-13	**7.67e-14**	2.50e-13
traincar_state9	8.11e-16	3.44e-13	4.31e-13	3.47e+02	1.91e-13	**3.45e-14**	4.15e-13	**3.45e-14**	2.38e-13
turbine1	5.79e-16	6.47e-13	1.48e-13	4.16e+02	6.81e-13	**2.06e-15**	3.07e-15	**2.06e-15**	3.90e-15
turbine2	1.03e-15	5.26e-15	4.25e-15	4.81e+00	1.66e-13	**4.12e-15**	4.30e-15	**4.12e-15**	4.33e-15
turbine3	7.41e-16	3.52e-13	7.43e-14	2.13e+02	3.91e-13	**1.91e-14**	**1.92e-14**	**1.91e-14**	**1.93e-14**

TABLE 5.3: Relative error bounds computed by different techniques

Benchmark	Daisy	FPTaylor	Naive approach	fwd analysis + subdiv	Opt.-based approach for direct relative error. comp. Z3	dReal	Z3 + subdiv	dReal + subdiv
Univariate benchmarks								
bsplines	6s	13s	13m 25s	0.34s	20s	25s	27s	30s
sines	5s	8s	13m 45s	0.42s	1m 4s	1m 21s	1m 8s	1m 9s
sqroot	3s	6s	6m 4s	0.15s	14s	12s	14s	14s
Multivariate benchmarks								
doppler	5s	2m 11s	2m 14s	1s	1m 59s	2m 35s	2m 58s	7m 28s
himmilbeau	9s	4s	5m 30s	0.36s	1m 50s	1m 16s	6m 15s	8m 5s
invPendulum	3s	5s	1m 31s	0.15s	7s	37s	25s	3m 54s
jet	20s	17s	19m 35s	7s	30m 40s	32m 24s	45m 31s	2 h 20m 49s
kepler	37s	39s	14m 41s	1s	3m 27s	16m 29s	12m 20s	27m 56s
rigidBody	11s	8s	10m 4s	0.39s	30s	1m 18s	1m 26s	8m 37s
traincar	10s	42s	8m 15s	1s	1m 1s	10m 43s	4m 7s	18m 35s
turbine	11s	28s	17m 25s	2s	5m 29s	11m 28s	12m 30s	42m 36s
total	1m 60s	5m 1s	1h 52m 28s	13s	46m 42s	1h 18m 45s	1h 27m 22s	4h 19m 53s

TABLE 5.4: Running times for computing the relative error bound using different techniques

above). Here we observe that unlike for the directly computed relative errors, interval subdivision is mostly beneficial. However, even with benefits from subdivision relative errors in this column are less accurate than the relative error bounds computed with the optimization-based approach. We also noticed that running times for the forward dataflow analysis in combination with subdivision are significantly less than for any other configuration. Thus, we can potentially improve the accuracy by increasing subdivision parameters m and p, while running times would be comparable to the running times of other techniques.

The tightest bounds for each benchmark were computed using one of the configurations of the optimization-based approach, and the difference between error bounds for different configurations is rather small. The most significant difference between configurations is in their running times, thus we choose the one with the smallest running time. We conclude, that the best trade-off between accuracy and performance has been shown by the optimization-based approach without subdivision and using the Z3 solver.

5.4.3 Effect of Denormals

In the floating-point approximation $\tilde{f}(x, e, d)$, we have two arguments that capture the roundoff error: e for normal numbers and d for denormals. The latter are required for complete soundness, but using them is expensive and their effect is usually small. There may be applications, where using denormals is not really necessary. We investigate the effect of the error term d in the abstraction. For that we compare the error bounds computed using the abstraction $\tilde{f}(x, e, d)$ with and $\tilde{f}(x, e)$ without denormals.

We have implemented the optimization based approach for both the absolute and relative error bounds estimation in Daisy, and compare the effect of denormals in both cases. As we defined in the previous section subsection 5.4.2 a good trade-off between accuracy and performance is achieved with the optimization-based approach without subdivisions using the Z3 solver. We use this configuration for all experiments, the timeout for Z3 is set to 1 second.

Table 5.5 shows absolute error bounds for the abstraction with and without the term d (standing for denormals) and running times respectively. The column 'Absolute error bounds with denormals' presents results for absolute error bounds, computed in the presence of error terms d in the abstraction $\tilde{f}(x, e, d)$, the column 'without' denotes relative errors, computed for the abstraction $\tilde{f}(x, e)$ without terms d. Columns 'Absolute times with denormals' and 'without' denote running times for error computations with abstractions $\tilde{f}(x, e, d)$ and $\tilde{f}(x, e)$ respectively.

We observe that the absolute error bounds are exactly the same for abstractions $\tilde{f}(x, e, d)$ and $\tilde{f}(x, e)$ up to the fourth digit in the fractional part for all benchmarks except *kepler2*. The running times for $\tilde{f}(x, e)$, perhaps surprisingly, are only slightly smaller than for the abstraction $\tilde{f}(x, e, d)$. For the three benchmarks in *kepler*, the total running time without denormals is even larger than with denormals in the

Benchmark	Absolute error bounds		Running times	
	with denormals	**without**	**with denormals**	**without**
Univariate benchmarks				
bspline0	1.1926e-16	1.1926e-16		
bspline1	6.5077e-16	6.5077e-16		
bspline2	5.8233e-16	5.8233e-16		
bspline3	1.1102e-16	1.1102e-16		
bsplines			*29.81s*	*31.12s*
sine	6.5413e-16	6.5413e-16		
sineOrder3	7.9985e-16	7.9985e-16		
sine			*45.13s*	*45.13s*
sqroot	3.9725e-13	3.9725e-13	*19.01s*	*18.83s*
Multivariate benchmarks				
doppler	1.7404e-13	1.7404e-13	*2m 33s*	*1m 25s*
himmilbeau	1.4211e-12	1.4211e-12	*35.70s*	*34.60s*
invPendulum	4.4371e-14	4.4371e-14	*5.78s*	*6.07s*
jet	2.4939e-11	2.4939e-11	*9m 41s*	*9m 9s*
kepler0	1.1515e-13	1.1515e-13		
kepler1	4.9898e-13	4.9898e-13		
kepler2	2.2777e-12	2.2241e-12		
kepler			*5m 41s*	*6m 35s*
rigidBody1	5.0793e-13	5.0793e-13		
rigidBody2	6.4752e-11	6.4752e-11		
rigidBody			*49.80s*	*55.70s*
traincar_state8	1.3323e-14	1.3323e-14		
traincar_state9	1.1991e-14	1.1991e-14		
traincar			*32.92s*	*27.70s*
turbine1	2.8023e-14	2.8023e-14		
turbine2	3.6663e-14	3.6663e-14		
turbine3	1.6540e-14	1.6540e-14		
turbine			*1m 50s*	*1m 36s*
		total	23m 23s	22m 23s

TABLE 5.5: Absolute errors for different abstractions (with/without denormals)

abstraction. This is a special case and we suspect, that for at least one of the *kepler* benchmarks adding d-s to the abstraction makes it easier for Z3 to process the query.

The results show that for absolute errors the difference (in both accuracy and performance) between using the abstraction $\tilde{f}(x, e, d)$ or $\tilde{f}(x, e)$ is rather insignificant.

We then investigate the effect of denormals on relative errors. Relative error bounds and running times are presented in Table 5.6. Similarly to our comparison for absolute errors for relative errors columns 'with denormals' and 'without' denote computations for the abstractions $\tilde{f}(x, e, d)$ and $\tilde{f}(x, e)$.

We noticed that for all benchmarks except for *kepler0* relative errors computed with and without denormals are exactly the same up to the fourth digit in fractional part. Comparing running times for *kepler0* we see that the abstraction $\tilde{f}(x, e)$ without denormals has been actually slower than the $\tilde{f}(x, e, d)$ with denormals, that means

Benchmark	Relative error bounds		Running times	
	with denormals	**without**	**with denormals**	**without**
Univariate benchmarks				
bspline0	2.9976e-15	2.9976e-15	5.09s	2.77s
bspline1	3.2197e-15	3.2197e-15	6.26s	4.02s
bspline2	8.9220e-16	8.9220e-16	7.60s	3.54s
bspline3	6.6614e-16	6.6614e-16	1.45s	0.13s
sine	7.6590e-16	7.6593e-16	1m 1.26s	12.15s
sineOrder3	8.9422e-16	8.9423e-16	3.13s	2.53s
sqroot	1.0214e-15	1.0214e-15	14.30s	8.90s
Multivariate benchmarks				
doppler	1.9312e-13	1.9310e-13	1m 59s	15.44s
himmilbeau	6.5417e-13	6.5417e-13	1m 50s	1m 35s
invPendulum	1.2124e-15	1.2124e-15	6.92s	6.53s
jet	4.4738e-15	4.4738e-15	30m 40s	2m 3s
kepler0	3.9676e-12	1.9846e-12	28.22s	46.66s
kepler1	3.7956e-11	3.7956e-11	1m 44s	1m 25s
kepler2	4.5606e-10	4.5606e-10	1m 16s	38.12s
rigidBody1	9.7794e-16	9.7794e-16	7.48s	6.55s
rigidBody2	2.2135e-15	2.2135e-15	22.99s	10.87s
traincar_state8	7.6668e-14	7.6668e-14	32.38s	23.67s
traincar_state9	3.4476e-14	3.4476e-14	28.33s	21.99s
turbine1	2.0616e-15	2.0616e-15	1m 41s	13.73s
turbine2	4.1184e-15	4.1184e-15	2m 14s	1m 24s
turbine3	1.9150e-14	1.9150e-14	1m 34s	20.14s
		total	46m 4s	10m 2s

TABLE 5.6: Relative errors for different abstractions (with/without denormals)

the solver timed out (more often) for $\tilde{f}(x, e)$. Similarly to the absolute error comparison results, we conclude that including terms d made it easier for the solver to process the query for this particular benchmark. For the rest of benchmarks the running times confirm our hypothesis: computing relative errors using abstraction $\tilde{f}(x, e)$ (without denormals) is significantly faster (the difference between running times is up to the factor of 5 for *sine* and *jet*).

The analysis using the abstraction $\tilde{f}(x, e,)$ runs faster without loss of accuracy, however, we should not forget that the error bounds computed using $\tilde{f}(x, e)$ are not sound. Thus, using the abstraction $\tilde{f}(x, e)$ without denormals may be a good alternative if complete soundness is not required.

5.4.4 Scalability of Relative Errors

The magnitude of roundoff errors (both absolute and relative) depends on the magnitude of input values. Larger input domains cause larger roundoff errors, but also the over-approximation of the errors grows together with the size of input domain. Ideally, we want this over-approximation to grow as slowly as possible. In this section, we explore the scalability of the two approaches for computing relative error

Benchmark	via absolute errors			directly		
	small	large	ratio	small	large	ratio
Univariate benchmarks						
bspline0	6.44e-15	4.12e-13	*64*	9.99e-16	3.00e-15	*3*
bspline1	1.57e-15	2.54e-15	*2*	2.07e-15	3.22e-15	*2*
bspline2	6.71e-16	1.11e-15	*2*	6.75e-16	8.92e-16	*1.32*
bspline3	3.27e-13	2.46e-10	*750.96*	6.66e-16	6.66e-16	*1*
sine	7.44e-16	8.94e-16	*1.20*	6.77e-16	7.66e-16	*1.13*
sineOrder3	5.70e-16	1.04e-15	*2*	4.81e-16	8.94e-16	*2*
sqroot	6.49e-16	1.04e-15	*1.61*	5.65e-16	1.02e-15	*1.81*
Multivariate benchmarks						
doppler	1.48e-11	2.08e-04	*1.40e+07*	1.26e-15	1.93e-13	*153.48*
himmilbeau	1.21e-15	6.55e-13	*541.15*	7.78e-16	6.54e-13	*841.07*
invPendulum	2.96e-13	2.09e-11	*70.74*	1.21e-15	1.21e-15	*1*
jet	9.05e-15	9.26e-15	*1.02*	4.60e-15	4.47e-15	*0.97*
kepler0	1.40e-15	1.31e-12	*934.97*	1.17e-15	3.97e-12	*3.39e+03*
kepler1	1.47e-15	2.17e-11	*1.47e+04*	3.06e-15	3.80e-11	*1.24e+04*
kepler2	4.28e-15	4.01e-10	*9.37e+04*	7.42e-15	4.56e-10	*6.15e+04*
rigidBody1	1.40e-12	8.77e-11	*62.66*	9.75e-16	9.78e-16	*1*
rigidBody2	2.00e-15	3.91e-12	*1.95e+03*	1.16e-15	2.21e-15	*1.90*
traincar_state8	6.93e-15	2.16e-13	*31.18*	1.64e-15	7.67e-14	*46.85*
traincar_state9	4.61e-15	3.44e-13	*74.67*	1.73e-15	3.45e-14	*19.96*
turbine1	4.46e-14	6.47e-13	*14.50*	1.75e-15	2.06e-15	*1.18*
turbine2	6.94e-16	5.26e-15	*7.57*	6.91e-16	4.12e-15	*5.96*
turbine3	1.10e-13	3.52e-13	*3.20*	6.50e-15	1.91e-14	*2.94*

TABLE 5.7: Relative error scalability with respect to the size of the input domain

bounds: the direct computation of relative errors and computing via absolute error with respect to the size of the input domain.

For the experiments in previous sections, we use as *large* input domains as possible without introducing result ranges which include zero. Now for our scalability comparison we also compute relative errors on *small* input domains. For that we modify the standard input domains of the benchmarks such that the width of input intervals is reduced, while the function ranges still do not contain zero. All experiments are performed with the Z3 solver with a timeout set to 1 second. The relative errors are computed without interval subdivision, since we noticed that it has a limited effect on accuracy while increasing running times.

Table 5.7 presents relative error bounds computed for smaller and larger input domains. Columns 'small' and 'large' show relative errors computed on smaller and larger input domains respectively. Column 'ratio' presents a relation between the values for the large and small domains. This relation characterizes the scalability of the approach, the smaller, the better.

Comparing the numbers from the 'ratio' columns we notice that for direct computation ratio is significantly smaller than for the computation via absolute errors. This means that the over-approximation committed by the direct computation is smaller than the over-approximation committed by the relative error computation

via absolute errors. The most prominent example is the *doppler* benchmark, where the directly computed error relative grew for the larger domain by two orders of magnitude, while the relative error computed via absolute grew by seven orders of magnitude. Based on these results we conclude that relative errors computed directly scale better than relative errors computed via absolute with respect to the size of the input domain.

Chapter 6

Division by zero

An important challenge arising while computing relative errors is how to handle potential divisions by zero. State-of-the-art tools today simply do not report any error at all whenever the function range contains zero. Unfortunately, this is not a rare occurrence; on a standard benchmark set for floating-point verification, many functions exhibit division by zero (see Table 6.2 for our experiments).

Note that these divisions by zero are *inherent* to the expression which we consider and are usually not due to over-approximations in the analysis. Thus, we can only *reduce* the effect of division by zero, but we cannot eliminate it entirely. Here, we aim to reduce the domain for which we cannot compute relative errors. Similar to how relative and absolute errors are combined in the IEEE 754 floating-point model (Equation 2.2), we want to identify parts of the input domain on which relative error computation is not possible due to division by zero and compute absolute errors. For the remainder of the input domain, we compute relative errors as before.

We use interval subdivision (section 5.1), attempting to compute relative errors (with one of the methods described before) on every sub-domain. Where the computation fails due to (potential) division by zero, we compute the absolute error and report both to the user:

```
relError: 6.6614143807162e-16
On several sub-intervals relative error cannot be computed.
Computing absolute error on these sub-intervals.
For intervals (u -> [0.875,1.0]), absError: 9.66746937132909e-19
```

While the reported relative error bound is only sound for parts of the domain, we believe that this information is nonetheless more informative than either no result at all or only an absolute error bound, which today's tools report and which may suffer from unnecessary over-approximations.

6.1 Possible Improvements

We realize that while this approach provides a practical solution, it is still preliminary and can be improved in several ways.

First, a smarter subdivision strategy would be beneficial. Currently, we divide the domain into equal-width intervals, and vary only their number. The more fine-grained the subdivision, the bigger part of the domain can be covered by relative error computations, however the running time increases correspondingly. Ideally, we could exclude from the relative error computation only a small domain around the zeros of the function f. While for univariate functions, this approach is straight-forward (zeros can be, for instance, obtained as models from a nonlinear decision procedure), for multivariate functions this is challenging, as the zeros are not simple points but curves.

Secondly, subdivision could only be used for determining which sub-domains exhibit potential division by zero. The actual relative error bound computation can then be performed on the remainder of the input domain without subdividing it. This would lead to performance improvements, even though the 'guaranteed-no-zero' domain can still consist of several disconnected parts. Again, for univariate functions this is a straight-forward extension, but non-trivial for multivariate ones.

Finally, whenever the function evaluates to zero $f(x) = 0$ we can replace its value with some small ϵ. Thus, for the sub-domains where division by zero occurs, we would compute:

$$err_{rel\_approx\_i} = \left| \frac{f(x) - \tilde{f}(x, e, d)}{f(x) + \epsilon} \right| \tag{6.1}$$

Note, that such an approximation is a common approach in scientific computing, even though Equation 6.1 does not compute the same relative error as we did before. For this reason, for now we do not follow this approach.

6.2 Experimental Evaluation

To evaluate whether interval subdivision is helpful when dealing with inherent division by zero, we now consider the standard benchmark set with standard input domains (as used in [2, 4], presented in Appendix A). We note that for these benchmarks division by zero indeed does occur quite often, that can be seen from our evaluation. We first find a default configuration for subdivision based on experiments and then compare relative errors computed with this configuration against relative errors computed with state-of-the-art tools.

6.2.1 Finding Default Configuration

Recall, the interval subdivision is parametric in the amount of subdivisions for each interval m and the total amount of optimizations p. The parameter m regulates how

Benchmark	# of vars	$p = 50$			$p = 100$		
		$m = 4$	$m = 6$	$m = 8$	$m = 4$	$m = 6$	$m = 8$
Univariate benchmarks							
bspline0	1	1 (4)	1 (6)	1 (8)	1 (4)	1 (6)	1 (8)
bspline1	1	1 (4)	1 (6)	0 (8)	1 (4)	1 (6)	0 (8)
bspline2	1	0 (4)	0 (6)	0 (8)	0 (4)	0 (6)	0 (8)
bspline3	1	1 (4)	1 (6)	1 (8)	1 (4)	1 (6)	1 (8)
sine	1	2 (4)	2 (6)	2 (8)	2 (4)	2 (6)	2 (8)
sineOrder3	1	-	2 (6)	2 (8)	-	2 (6)	2 (8)
sqroot	1	3 (4)	2 (6)	3 (8)	3 (4)	2 (6)	3 (8)
Multivariate benchmarks							
doppler	3	0 (16)	0 (36)	0 (8)	0 (64)	0 (36)	0 (64)
himmilbeau	2	12 (16)	17 (36)	-	12 (16)	17 (36)	15 (64)
invPendulum	4	12 (16)	22 (36)	6 (8)	32 (64)	22 (36)	34 (64)
jet	2	15 (16)	33 (36)	-	15 (16)	33 (36)	56 (64)
kepler0	6	-	-	-	49 (64)	-	-
kepler1	4	-	-	-	63 (64)	-	-
kepler2	6	-	-	-	-	-	-
rigidBody1	3	-	-	-	46 (64)	-	-
rigidBody2	3	-	-	-	50 (64)	-	-
traincar_state8	14	-	-	-	-	-	-
traincar_state9	14	-	-	-	-	-	-
turbine1	3	0 (16)	0 (36)	0 (8)	0 (64)	0 (36)	0 (64)
turbine2	3	10 (16)	17 (36)	6 (8)	25 (64)	17 (36)	25 (64)
turbine3	3	0 (16)	0 (36)	0 (8)	0 (64)	0 (36)	0 (64)

The first number stands for the amount of sub-domains where computation of relative errors failed, the second is the total amount of sub-domains. For example $1(4)$ means that on one of the four sub-domains division by zero occurred.

TABLE 6.1: Comparison of different configurations for subdivision

fine-grained the subdivision of each input interval should be, while the parameter p is intended to limit the running time. p bounds not only the total amount of optimization runs, but also regulates for how many variables the input intervals are subdivided. This balance between more *subdivisions for one* interval and more input *intervals being subdivided* may change if we have to deal with potential division by zero. Thus, we do not reuse the configuration found in subsection 5.4.1, and perform the comparison for several values of m and p. We compare results for $m = 4, 6$ and 8 and $p = 50, 100$.

The focus of this comparison is to find a configuration which computes relative errors for as many benchmarks as possible and for as big part of the input domain as possible. Therefore, for different configurations we compare the amount of sub-domains where computations failed because of division by zero. Table 6.1 summarizes our results. Columns $m = 4$, $m = 6$ and $m = 8$ show the amount of

sub-domains where division by zero occurred for 4, 6 and 8 subdivisions for each input interval with upper limit for total amount of optimizations $p = 50$ and 100 respectively. The result consists of two values: the first value is the amount of sub-domains where computation of relative errors failed, the second is the total amount of sub-domains, e.g. $0(4)$ means that relative error has been successfully computed on all sub-domains. Underline marks the results, for which relative error computation failed (due to division by zero) on more than 80% of sub-domains. We consider such results to be impractical and thus will not report error bounds for these cases. Whenever we report '-' in the table, this means that relative error computations reported division by zero for all sub-domains.

For univariate benchmarks we see for almost all benchmarks all configurations provided relative error bounds. Only for *sineOrder* subdivision of $m = 4$ sub-intervals is not sufficient to obtain a result. Interestingly, for *bspline1* computations reported division by zero for $m = 4$ and $m = 6$, but for the more fine-grained subdivision $m = 8$ no division by zero occurred, and it was possible to compute relative error bound for all sub-domains. Since the configuration $m = 8$ $p = 50$ allows to compute relative error for all univariate benchmarks in our set, we choose it as a default for univariate benchmarks. Note that for univariate benchmarks it does not play a role whether we take $p = 50$ or $p = 100$, as the amount of sub-domains for all tested values of m is lower than 50 and 100.

We noticed that for multivariate benchmarks there is one configuration that allowed to compute relative error for most of the benchmarks, that is $m = 4$, $p = 100$. That means that for this set of multivariate benchmarks it is beneficial to subdivide each individual interval less, while having intervals for more variables subdivided. We choose this configuration as default for multivariate benchmarks.

For some benchmarks, however, independent from the subdivision parameters it was still not possible to compute any estimate of relative error, or the computed error bound is valid for only a small sub-domain (*jet* and *kepler1*).

6.2.2 Comparison with State-of-the-Art Tools

We compare the results of relative error computations of state-of-the-art tools with the default configuration for the interval subdivision combined with the forward analysis and the optimization-based approach.

Table 6.2 summarizes our results. We report '-' in the table whenever the tools could not compute any relative error estimate or the computed error is valid on less than 20% of sub-domains. Columns 'Daisy' and 'FPTaylor' show relative error bounds computed via absolute errors by Daisy(before this project) and FPTaylor tool. Last three columns show our results when using interval subdivision. Column 'fwd analysis+subdiv' presents relative errors computed using the combination of interval subdivision with forward analysis. Columns 'Z3 + subdiv' and 'dReal + subdiv' show relative errors computed combining the interval subdivision with the optimization based approach using the solvers Z3 and dReal respectively.

We observe that while interval subdivision does not provide us with a result for all benchmarks, it nonetheless computes estimates for more benchmarks than state-of-the-art tools.

We also report running times for all benchmarks in Table 6.3. We present it independently of whether an error estimate could be computed or not. Note that running times are cumulated for benchmarks in one input program, e.g. times for *sine* and *sineOrder3* are summed and presented under *sines*.

We notice that running times for the optimization-based approach with both Z3 and dReal are significantly greater than for state-of-the-art tools. This is caused by the fact that we run the analysis as many times as we have sub-domains. This means, that we evaluate every term of the Taylor approximation Equation 5.4, including the first order derivatives $\frac{\partial \tilde{g}}{\partial e_i}(x, 0, 0)e_i$ and the second order derivatives summed in M_R, multiple times. On multivariate benchmarks this inevitably introduces a noticeable slowdown, which is especially unwanted when no relative error is reported in the end due to division by zero for every sub-domain. However, there is a practical solution to avoid long waiting times if no result can be computed. We noticed, that whenever the combination of interval subdivision and the optimization-based approach can compute relative error, the combination of interval subdivision with forward analysis ('fwd analysis+subdiv') also can, while running times for the latter combination of techniques are significantly less. Thus, we suggest to run 'fwd analysis+subdiv' configuration to obtain preliminary results and refine them when necessary by applying the combination of the optimization based approach and interval subdivision.

Benchmark	Daisy	FPTaylor	fwd analysis+subdiv	Optimization-based approach Z3 + subdiv	Optimization-based approach dReal + subdiv
Univariate benchmarks					
bspline0	-	-	1.58e-01	3.00e-15	3.00e-15
bspline1	-	3.32e-15	2.80e-13	3.22e-15	3.22e-15
bspline2	-	3.50e-15	9.20e-16	8.92e-16	8.92e-16
bspline3	-	-	1.31e-14	6.66e-16	6.66e-16
sine	-	-	1.07e-15	7.02e-16	7.02e-16
sineOrder3	-	-	2.29e-15	8.94e-16	8.94e-16
sqroot	-	-	7.09e-15	1.92e-15	1.92e-15
Multivariate benchmarks					
doppler	1.48e-11	4.99e-12	8.95e-13	1.26e-15	1.35e-15
himmilbeau	-	-	3.75e-14	2.57e-14	2.84e-15
invPendulum	-	-	4.94e-15	2.82e-15	3.08e-15
jet	-	-	-	-	-
kepler0	4.35e-15	4.57e-15	2.38e-13	2.16e-15	3.88e-15
kepler1	1.33e-14	1.17e-14	-	-	-
kepler2	-	4.21e-14	-	-	-
rigidBody1	-	-	2.29e-14	1.07e-15	1.78e-15
rigidBody2	-	-	2.65e-12	1.67e-15	3.80e-15
traincar_state8	-	-	-	-	-
traincar_state9	-	-	-	-	-
turbine1	6.12e-14	1.18e-14	6.03e-15	1.75e-15	5.21e-15
turbine2	-	-	5.64e-14	2.74e-15	6.97e-14
turbine3	1.52e-13	2.21e-14	2.77e-14	6.50e-15	6.71e-15

TABLE 6.2: Relative error bounds computed by different techniques on standard benchmarks (with potential division by zero)

Benchmark	Daisy	FPTaylor	fwd analysis+subdiv	Optimization-based approach Z3 + subdiv	Optimization-based approach dReal + subdiv
Univariate benchmarks					
bsplines	7.50s	16.91s	0.58s	2m 39s	1m 40s
sines	8.26s	10.89s	1.51s	8m 11s	7m 31s
sqroot	5.20s	5.48s	0.29s	56.78s	52.22s
Multivariate benchmarks					
doppler	5.87s	10.00s	7.09s	2h 32m 37s	3h 15m 2s
himmilbeau	6.63s	8.77s	0.48s	5m 12s	3m 51s
invPendulum	3.35s	6.25s	0.45s	2m 42s	8m 33s
jet	48.44s	18.87s	12.00s	5h 28m 30s	7h 53m 45s
kepler	52.38s	2m 28s	3.56s	1h 13m 2s	12h 8m 52s
rigidBody	13.06s	10.47s	1.22s	11m 10s	1h 41m 16s
traincar	5.92s	54.97s	4.36s	16m 7s	1h 54m 16s
turbine	12.20s	31.44s	9.12s	4h 34m 52s	8h 45m 14s
total	2m 49s	5m 22s	41s	14h 35m 59s	36h 53s

TABLE 6.3: Running times for relative error computations for different techniques on standard benchmarks (with potential division by zero)

Chapter 7

Related Work

Th goal of this work is to find an automated and sound static analysis technique for computing tight relative error bounds for floating-point arithmetic. The most related work are current static analysis tools for computing absolute roundoff error bounds [2–4, 16] which we have already reviewed.

Another closely related tool is Gappa [13]. Gappa is designed as a helper tool for verifying the correctness of numerical programs in Coq, that provides a strong guarantee for the computed bound. It appears relative errors can be both computed directly or via absolute errors. The *automated* error computation inside Gappa uses forward dataflow analysis and interval arithmetic. Thus, computation of relative errors via absolute errors is less accurate than what Daisy performs, since Daisy additionally refines intervals using a nonlinear decision procedure. Even if relative errors are computed directly (is not clear if this done automatically), this amounts to the naive approach, which we have implemented and showed that it works poorly. Gappa also provides the possibility to apply user hints, which can potentially help to compute better bounds, however, this approach is then comparable to interactive theorem proving and not fully automated techniques.

The only direct relative error computation that we are aware of was used in the context of verifying bit-shifting manipulations of floating-point numbers [20]. The approach is specific to low-level bit operations and includes only polynomials. To compute roundoff error bounds, the authors use the optimization based approach similar to FPTaylor's, but instead of using Taylor theorem the expression to be maximized is simplified by omitting sufficiently small polynomial terms. However, tight error bounds are not the focus of the paper, and authors report only one of roundoff errors, absolute or relative, whichever one is better. To summarize, while many tools compute relative errors we are not aware of any systematic evaluation of different approaches for sound relative error bounds.

7.1 Abstract Interpretation-Based Analysis

More broadly related are abstract interpretation-based static analyses which are sound with respect to floating-point arithmetic. The analyses utilize various abstract domains: intervals [21–23], octagons [24, 25], polyhedra [23, 26] for which there exist

implementations provided with a uniform API [22] and some of the domains have been formalized in Coq [21]. These abstract domains are applied to track possible values for floating-point variables and analyze whether they can overflow, turn to infinity or throw any other exceptions specified in IEEE 754. They do not quantify the difference between real and floating-point semantics, thus they only can prove the absence of run-time errors but cannot report any roundoff errors.

7.2 Theorem Provers

Floating-point arithmetic has also been formalized in interactive theorem provers, e.g. in Coq [27] and HOL-Light [28]. Verification conditions for numerical programs can be generated automatically [29, 30], including reasoning about ranges in interval [13] and affine arithmetic [31]. Using an existing formalization, verification conditions can be discharged interactively with proof assistants to verify numerical programs. Entire numerical programs implemented in C have been proven correct [32, 33] including to errors introduced by rounding. For some specific programs (such as x^n) a very tight relative error bound can be proved [34]. However, these methods are to a large part manual and require substantial user expertise in both floating-point numbers and theorem provers. Our work is not program specific and it aims at a different level of generality and thus does not need user's expertise in floating-point arithmetic as it is fully automated.

7.3 SMT-Lib Standard

Another formalization for theory of floating-point arithmetic is proposed in SMT-Lib [35]. The formalization also supports all special values of the IEEE 754, arithmetic and comparison operations and thus enables SMT solvers that follow the SMT-Lib standard to handle floating-point arithmetic. Another decision procedure for the theory of floating-point arithmetic is proposed by Brain et al. [36]. Authors present a natural-domain SMT solver that combines interval abstraction for floating-points with satisfiability algorithms. These are, however, not suitable for roundoff error computation, as this would require to combine the theory of reals with the theory of floating-points. Such a combination would currently be at the propositional level only and thus does not lead to any useful results.

7.4 Dynamic Program Analysis

One more related approach is dynamic program analysis. Often a higher-precision program is executed alongside the original one and thus by testing analyzers obtain absolute [37] or relative roundoff errors [38], or detect digit cancellation events [39]. These methods include both error estimation based on maintaining shadow values at higher precision and heuristic search guided testing and aim to obtain a lower

bound on roundoff error, while we (and other static analysis tools) compute sound upper bounds. The tool Herbie [40] not only computes the absolute roundoff error, but suggests how to improve accuracy of the program by reordering arithmetic operations preserving their real-valued semantic. It first identifies which operations contribute the most to the absolute roundoff error. For that on several sampled inputs it compares the result of the floating-point operation with the result of the operation in arbitrary precision. For the operations with the highest error Herbie searches possible improvements in a database of rewriting rules and generates alternative operation. However, Herbie performs its analysis for randomly sampled inputs and does not statically analyze programs. Thus, it cannot provide worst-case error bound guarantees.

7.5 Mixed-Precision Optimization

Testing has also been used for optimizing mixed-precision computations in different tools. Precimonious [41] uses the delta-debugging algorithm [42] to obtain possible data type assignments for floating-point variables and executes a program with proposed configurations in order to determine whether a newly proposed configuration provides performance improvements. For this procedure it requires a representative set of training inputs. Lam's et al. mixed-precision tuning algorithm [43] modifies the binary code of the original program and automatically searches the configuration, which promises the greatest improvement in running time and memory bandwidth usage. These tools, however, are based on testing and random input sampling or only valid for a limited training input data set, and thus cannot provide sound error bounds.

For mixed-precision optimization there have recently appeared static analysis tools that not only improve running times but guarantee that a specified accuracy bound will be satisfied by the new data type assignment for all inputs from the defined input range. Today's tools use absolute roundoff error to define the accuracy bound. One of the tools is called FPTuner [44]; it is a follow-up of FPTaylor, which additionally to absolute roundoff error computations solves the second optimization problem with performance as the objective. The second tool allowing automated sound mixed-precision tuning is the new feature of Daisy [45]. Additionally to a mixed-precision assignment, Daisy also performs rewriting, i.e. re-ordering of computations preserving real arithmetic semantics. Our approach is also applicable to mixed-precision (without significant modifications), and thus could be used for mixed-precision tuning with the relative error as a measure of accuracy.

Chapter 8

Conclusion

Automated tools for sound absolute error analysis are intended to help developers write more reliable code with respect to numerical errors. However, absolute errors are not always a good estimate for the result's quality. Our goal was to explore today's techniques for computing the relative roundoff errors and by combining different strategies to find a fully automated technique which computes sound and tight relative error bounds.

We have presented the first experimental investigation into the suitability of different static analysis techniques for sound accurate relative error estimation. Provided that the function range does not include zero, computing relative errors *directly* usually yields error bounds which are (orders of magnitude) more accurate than if relative errors are computed via absolute errors (as is the current state-of-the-art). Surprising to us, while interval subdivision is beneficial for absolute error estimation, when applied to the direct relative error computation it most often does not have a significant effect on accuracy.

Additionally, we found that interval subdivision helps to alleviate the effect of the inherent division by zero issue in relative error computation. Nonetheless it still remains an open challenge.

Our experimental evaluation has also shown that the direct computation of relative errors scales better with respect to the input domains of a different sizes compared to the relative error computation via absolute errors. We also demonstrate that omitting error terms for denormals in relative error computations usually does not effect accuracy, but gain performance. Thus, using the abstraction for floating-point operation which does not include denormals may be a good alternative, if complete soundness is not necessary.

During our experiments, we used the SMT solvers as part of an interval refining procedure. We noticed that while these tools are in general rigorous, they might time out unpredictably. As a result, timeouts might lead to the premature stop of the refinement procedure and, thus, coarser bounds. Therefore, we note that today's rigorous optimization tools could be improved in terms of both reliability and scalability.

We believe that the results we obtained are encouraging and that our work is a successful step towards improving techniques for computing relative error, which may lead to better accuracy specifications in the future.

Bibliography

[1] R.E. Moore. *Interval Analysis*. Prentice-Hall, 1966.

[2] Eva Darulova and Viktor Kuncak. "Sound Compilation of Reals". In: *POPL*. 2014.

[3] Eric Goubault and Sylvie Putot. "Static Analysis of Finite Precision Computations". In: *VMCAI*. 2011.

[4] Alexey Solovyev et al. "Rigorous Estimation of Floating-Point Round-off Errors with Symbolic Taylor Expansions". In: *FM*. 2015.

[5] MPI-SWS. *Daisy - a framework for accuracy analysis and synthesis of numerical programs*. Version 0.1. https://gitlab.mpi-sws.org/AVA/daisy-public. 2017.

[6] "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 754-2008* (Aug. 2008).

[7] David Goldberg. "What Every Computer Scientist Should Know About Floating-point Arithmetic". In: *ACM Comput. Surv.* 23.1 (Mar. 1991), pp. 5–48. ISSN: 0360-0300.

[8] L. H. de Figueiredo and J. Stolfi. "Affine Arithmetic: Concepts and Applications". In: *Numerical Algorithms* 37.1-4 (2004).

[9] Eva Darulova and Viktor Kuncak. "Trustworthy Numerical Computation in Scala". In: *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '11. Portland, Oregon, USA: ACM, 2011, pp. 325–344. ISBN: 978-1-4503-0940-0.

[10] Eva Darulova and Viktor Kuncak. "Towards a Compiler for Reals". In: *ACM TOPLAS* 39.2 (2017).

[11] Leonardo De Moura and Nikolaj Bjørner. "Z3: an efficient SMT solver". In: *TACAS*. 2008.

[12] Dejan Jovanović and Leonardo de Moura. "Solving Non-linear Arithmetic". In: *IJCAR*. 2012.

[13] Marc Daumas and Guillaume Melquiond. "Certification of Bounds on Expressions Involving Rounded Operators". In: *ACM Trans. Math. Softw.* 37.1 (Jan. 2010), 2:1–2:20. ISSN: 0098-3500.

[14] Mark S. Baranowski and Ian Briggs. *Gelpia - Global Extrema Locator Parallelization for Interval Arithmetic*. https://github.com/soarlab/gelpia.

[15] S.G. Johnson. *The NLopt nonlinear-optimization package*. http://ab-initio.mit.edu/nlopt.

[16] Victor Magron, George Constantinides, and Alastair Donaldson. "Certified Roundoff Error Bounds Using Semidefinite Programming". In: *ACM Trans. Math. Softw.* 43.4 (Jan. 2017), 34:1–34:31. ISSN: 0098-3500.

[17] Sicun Gao, Soonho Kong, and Edmund M. Clarke. "dReal: An SMT Solver for Nonlinear Theories over the Reals". In: *CADE*. 2013.

[18] Eva Darulova et al. "Synthesis of Fixed-point Programs". In: *EMSOFT*. 2013.

[19] *MPFR Java Bindings*. https://github.com/kframework/mpfr-java.

[20] Wonyeol Lee, Rahul Sharma 0001, and Alex Aiken. "Verifying Bit-Manipulations of Floating-Point." In: *PLDI* (2016).

[21] Jacques-Henri Jourdan et al. "A Formally-Verified C Static Analyzer". In: *POPL*. 2015.

[22] Bertrand Jeannet and Antoine Miné. "Apron: A Library of Numerical Abstract Domains for Static Analysis". In: *CAV*. 2009.

[23] Daniel Kästner et al. *Astrée: Proving the Absence of Runtime Errors*. 2010.

[24] Antoine Miné. "Relational Abstract Domains for the Detection of Floating-Point Run-Time Errors". In: *ESOP 2004, Held as Part of the ETAPS 2004*. Ed. by David Schmidt. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 3–17. ISBN: 978-3-540-24725-8.

[25] Bruno Blanchet et al. "A Static Analyzer for Large Safety-Critical Software". In: *PLDI*. 2003.

[26] Liqian Chen, Antoine Miné, and Patrick Cousot. "A Sound Floating-Point Polyhedra Abstract Domain". In: *APLAS*. 2008.

[27] Sylvie Boldo and Guillaume Melquiond. "Flocq: A Unified Library for Proving Floating-Point Algorithms in Coq". In: *ARITH*. 2011.

[28] Charles Jacobsen, Alexey Solovyev, and Ganesh Gopalakrishnan. "A Parameterized Floating-Point Formalizaton in HOL Light". In: *Electronic Notes in Theoretical Computer Science* 317 (2015), pp. 101–107.

[29] Sylvie Boldo and Jean-Christophe Filliâtre. "Formal Verification of Floating-Point Programs". In: *18th IEEE Symposium on Computer Arithmetic (ARITH-18 2007), 25-27 June 2007, Montpellier, France*. 2007, pp. 187–194.

[30] Ali Ayad and Claude Marché. "Multi-Prover verification of floating-point programs". In: *IJCAR*. 2010.

[31] Michael D. Linderman et al. "Towards program optimization through automated analysis of numerical precision". In: *CGO*. 2010.

[32] Sylvie Boldo et al. "Wave Equation Numerical Resolution: A Comprehensive Mechanized Proof of a C Program". In: *Journal of Automated Reasoning* 50.4 (2013), pp. 423–456.

[33] Tahina Ramananandro et al. "A Unified Coq Framework for Verifying C Programs with Floating-Point Computations". In: *CPP*. 2016.

[34] Stef Graillat, Vincent Lefèvre, and Jean-Michel Muller. *On the maximum relative error when computing x^n in floating-point arithmetic*. Tech. rep. <ensl-00945033v2>. Laboratoire d'Informatique de Paris 6, Inria Grenoble Rhône-Alpes, 2014.

[35] Philipp Rümmer and Thomas Wahl. "An SMT-LIB Theory of Binary Floating-Point Arithmetic". In: *SMT*. 2010.

[36] Martin Brain et al. "Deciding floating-point logic with abstract conflict driven clause learning". In: *Formal Methods in System Design* 45.2 (Dec. 2013), pp. 213–245.

[37] Florian Benz, Andreas Hildebrandt, and Sebastian Hack. "A Dynamic Program Analysis to Find Floating-point Accuracy Problems". In: *PLDI*. 2012.

[38] Wei-Fan Chiang et al. "Efficient Search for Inputs Causing High Floating-point Errors". In: *PPoPP*. 2014.

[39] Michael O. Lam, Jeffrey K. Hollingsworth, and G.W. Stewart. "Dynamic Floating-point Cancellation Detection". In: *Parallel Computing* 39.3 (2013).

[40] Pavel Panchekha et al. "Automatically Improving Accuracy for Floating Point Expressions". In: *PLDI*. 2015.

[41] Cindy Rubio-González et al. "Precimonious: Tuning Assistant for Floating-point Precision". In: *SC*. 2013.

[42] Andreas Zeller and Ralf Hildebrandt. "Simplifying and Isolating Failure-Inducing Input". In: *IEEE Trans. Softw. Eng.* 28.2 (Feb. 2002), pp. 183–200. ISSN: 0098-5589.

[43] Michael O Lam et al. "Automatically Adapting Programs for Mixed-precision Floating-point Computation". In: *ICS*. 2013.

[44] Wei-Fan Chiang et al. "Rigorous Floating-Point Mixed-Precision Tuning". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. ACM, 2017, pp. 300–315.

[45] Eva Darulova, Einar Horn, and Saksham Sharma. "Sound Mixed-Precision Optimization with Rewriting". In: *arXiv:1707.02118*. 2017.

Appendix A

Standard Benchmarks

In this appendix, we present the standard benchmarks with the standard input domains with potential division by zero. These benchmarks are used in the experimental evaluation described in section 6.2.

Univariate benchmarks

– bsplines

```
def bspline0(u: Real): Real = {
   require(0 <= u && u <= 1)
   (1 - u) * (1 - u) * (1 - u) / 6.0
 }

def bspline1(u: Real): Real = {
   require(0 <= u && u <= 1)
   (3 * u*u*u - 6 * u*u + 4) / 6.0
}

def bspline2(u: Real): Real = {
   require(0 <= u && u <= 1)
   (-3 * u*u*u + 3*u*u + 3*u + 1) / 6.0
}

def bspline3(u: Real): Real = {
   require(0 <= u && u <= 1)
   -u*u*u / 6.0
}
```

– sine

```
def sine(x: Real): Real = {
   require(x > -1.57079632679 && x < 1.57079632679)
   x - (x*x*x)/6.0 + (x*x*x*x*x)/120.0 - (x*x*x*x*x*x*x)/5040.0
}

def sineOrder3(x: Real): Real = {
   require(-2.0 < x && x < 2.0)
   0.954929658551372 * x - 0.12900613773279798*(x*x*x)
}
```

– sqroot

```
def sqroot(x: Real): Real = {
   require(x >= 0.0 && x < 10.0)
   1.0 + 0.5*x - 0.125*x*x + 0.0625*x*x*x - 0.0390625*x*x*x*x
}
```

Multivariate benchmarks

– doppler

```
def doppler(u: Real, v: Real, T: Real): Real = {
   require(-100.0 <= u && u <= 100 && 20 <= v && v <= 20000 &&
             -30 <= T && T <= 50)

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

– himmilbeau

```
def himmilbeau(x1: Real, x2: Real) = {
   require(-5 <= x1 && x1 <= 5 && -5 <= x2 && x2 <= 5)

   (x1 * x1 + x2 - 11)*(x1 * x1 + x2 - 11) +
      (x1 + x2*x2 - 7)*(x1 + x2*x2 - 7)

}
```

– invertedPendulum

```
def invPendulum(s1: Real, s2: Real, s3: Real, s4: Real) = {
   require(-50 <= s1 && s1 <= 50 && -10 <= s2 && s2 <= 10 &&
      -0.785 <= s3 && s3 <= 0.785 && -0.785 <= s4 && s4 <= 0.785)

   1.0000 * s1 + 1.6567 * s2 + (-18.6854) * s3 + (-3.4594) * s4
 }
```

– jetEngine

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

   x1 + ((2 * x1 * ((3 * x1 * x1 + 2 * x2 - x1)/(x1 * x1 + 1))*
     ((3 * x1 * x1 + 2 * x2 - x1)/(x1 * x1 + 1) - 3) +
     x1 * x1 * (4 * ((3 * x1 * x1 + 2 * x2 - x1)/(x1 * x1 + 1))-6)) *
     (x1 * x1 + 1) + 3 * x1 * x1 *
     ((3 * x1 * x1 + 2 * x2 - x1)/(x1 * x1 + 1)) +
     x1 * x1 * x1 + x1 + 3 * ((3 * x1 * x1 + 2 * x2 -x1)/(x1 * x1 + 1)))
}
```

– kepler

```
def kepler0(x1: Real, x2: Real, x3: Real, x4: Real,
                                      x5: Real, x6: Real): Real = {
   require(4 <= x1 && x1 <= 6.36 && 4 <= x2 && x2 <= 6.36 &&
               4 <= x3 && x3 <= 6.36 && 4 <= x4 && x4 <= 6.36 &&
               4 <= x5 && x5 <= 6.36 && 4 <= x6 && x6 <= 6.36)

   x2 * x5 + x3 * x6 - x2 * x3 - x5 * x6 +
            x1 * (-x1 + x2 + x3 - x4 + x5 + x6)

}

def kepler1(x1: Real, x2: Real, x3: Real, x4: Real): Real = {
   require(4 <= x1 && x1 <= 6.36 && 4 <= x2 && x2 <= 6.36 &&
             4 <= x3 && x3 <= 6.36 && 4 <= x4 && x4 <= 6.36)

   x1 * x4 * (-x1 + x2 + x3 - x4) + x2 * (x1 - x2 + x3 + x4) +
    x3 * (x1 + x2 - x3 + x4) - x2 * x3 * x4 - x1 * x3 - x1 * x2 - x4

}

def kepler2(x1: Real, x2: Real, x3: Real, x4: Real,
                                      x5: Real, x6: Real): Real = {
   require(4 <= x1 && x1 <= 6.36 && 4 <= x2 && x2 <= 6.36 &&
               4 <= x3 && x3 <= 6.36 && 4 <= x4 && x4 <= 6.36 &&
               4 <= x5 && x5 <= 6.36 && 4 <= x6 && x6 <= 6.36)

    x1 * x4 * (-x1 + x2 + x3 - x4 + x5 + x6) + x2 * x5 *
          (x1 - x2 + x3 + x4 - x5 + x6) + x3* x6 *
                (x1 + x2 - x3 + x4 + x5 - x6) - x2 * x3 * x4 -
                            x1* x3* x5 - x1 * x2 * x6 - x4 * x5 * x6
}
```

– rigidBody

```
def rigidBody1(x1: Real, x2: Real, x3: Real): Real = {
   require(-15.0 <= x1 && x1 <= 15 && -15.0 <= x2 && x2 <= 15.0 &&
             -15.0 <= x3 && x3 <= 15)

   -x1*x2 - 2*x2*x3 - x1 - x3
}

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
}
```

– traincar

```
def state8(s0: Real, s1: Real, s2: Real, s3: Real, s4: Real,
           s5: Real, s6: Real, s7: Real, s8: Real, y0: Real,
           y1: Real, y2: Real, y3: Real, y4: Real) = {
  require(-2.5 <= s0 && s0 <= 6.5 && -2.5 <= s1 && s1 <= 6.5 &&
   -2.5 <= s2 && s2 <= 6.5 && -2.5 <= s3 && s3 <= 6.5 &&
    -2 <= s4 && s4 <= 12 && -2 <= s5 && s5 <= 12 &&
    -2 <= s6 && s6 <= 12 && -2 <= s7 && s7 <= 12 &&
    -2 <= s8 && s8 <= 12 && -2 <= y0 && y0 <= 12 &&
    -2 <= y1 && y1 <= 12 && -2 <= y2 && y2 <= 12 &&
    -2 <= y3 && y3 <= 12 && -2 <= y4 && y4 <= 12)
  (2.5093e-10)*s0 +(9.15884e-10)*s1+ (7.81656e-06)*s2+
  (-7.81701e-06)*s3 + (-6.54335e-07)*s4 + (6.87341e-06)*s5 +
   (1.00368e-05)*s6 + (0.999907)*s7 + (3.32876e-05)*s8 +
   (6.5448232e-07)*y0 + (-6.8708837e-06)*y1 + (-8.9460042e-06)*y2 +
    (9.0317123e-05)*y3+ (-3.2191562e-05)*y4 +
      (-1.8530512e-13)*5.2121094496644555e+03
}

def state9(s0: Real, s1: Real, s2: Real, s3: Real, s4: Real,
                 s5: Real, s6: Real, s7: Real, s8: Real, y0: Real,
                 y1: Real, y2: Real, y3: Real, y4: Real) = {
  require(-2.5 <= s0 && s0 <= 6.5 && -2.5 <= s1 && s1 <= 6.5 &&
              -2.5 <= s2 && s2 <= 6.5 && -2.5 <= s3 && s3 <= 6.5 &&
              -2 <= s4 && s4 <= 12 && -2 <= s5 && s5 <= 12 &&
              -2 <= s6 && s6 <= 12 && -2 <= s7 && s7 <= 12 &&
              -2 <= s8 && s8 <= 12 && -2 <= y0 && y0 <= 12 &&
              -2 <= y1 && y1 <= 12 && -2 <= y2 && y2 <= 12 &&
              -2 <= y3 && y3 <= 12 && -2 <= y4 && y4 <= 12)
  (-1.73572e-09)*s0 +(-6.90441e-09)*s1 + (1.91831e-08)*s2 +
   (7.80416e-06)*s3 + (5.01527e-06)*s4 + (-4.73947e-06)*s5 +
    (4.30545e-07)*s6 +(3.35281e-05)*s7 + (0.999934)*s8 +
          (-5.0163739e-06)*y0 + (4.7201386e-06)*y1 +
           (-4.156438e-07)*y2 + (-3.2406398e-05)*y3 +
            (6.4987306e-05)*y4 +
                  (1.4201936e-12)*5.2121094496644555e+03
    }
```

– turbine

```
def turbine1(v: Real, w: Real, r: Real): Real = {
  require(-4.5 <= v && v <= -0.3 && 0.4 <= w && w <= 0.9 &&
            3.8 <= r && r <= 7.8)
  3 + 2/(r*r) - 0.125*(3-2*v)*(w*w*r*r)/(1-v) - 4.5
}

def turbine2(v: Real, w: Real, r: Real): Real = {
  require(-4.5 <= v && v <= -0.3 && 0.4 <= w && w <= 0.9 &&
            3.8 <= r && r <= 7.8)

  6*v - 0.5 * v * (w*w*r*r) / (1-v) - 2.5
}
```

```
def turbine3(v: Real, w: Real, r: Real): Real = {
   require(-4.5 <= v && v <= -0.3 && 0.4 <= w && w <= 0.9 &&
             3.8 <= r && r <= 7.8)

   3 - 2/(r*r) - 0.125 * (1+2*v) * (w*w*r*r) / (1-v) - 0.5
}
```

Appendix B

Standard Benchmarks with Modified Input Domains

In this appendix, we present the the modified input domains for the standard benchmarks. Since the arithmetic expressions for these benchmarks remain unchanged, we only present input domains. They are modified such that function range does not include zero. We have performed the experiments on both larger and smaller input domains, these are listed below in section B.1 and section B.2 respectively. We use larger domains in the experiments throughout section 5.4 and in subsection 5.4.4 the smaller domains are used additionally.

B.1 Large Input Domains

Univariate benchmarks

– bsplines

```
def bspline0(u: Real): Real = {
   require(0 <= u && u <= 0.875)
... }

def bspline1(u: Real): Real = {
   require(0.875 <= u && u <= 1)
... }

def bspline2(u: Real): Real = {
   require(0.5 <= u && u <= 1)
... }

def bspline3(u: Real): Real = {
   require(0.125 <= u && u <= 10)
   ... }
```

– sine

```
def sine(x: Real): Real = {
   require(x > 0.875 && x < 1.7)
... }
```

```
def sineOrder3(x: Real): Real = {
   require(-2.0 < x && x < -1.125)
... }
```

– sqroot

```
def sqroot(x: Real): Real = {
   require(x >= 0.0 && x < 1.925)
... }
```

Multivariate benchmarks

– doppler

```
def doppler(...): Real = {
   require(-100.0 <= u && u <= 1000 && 2 <= v && v <= 200000 &&
          -300 <= T && T <= 500)
... }
```

– himmilbeau

```
def himmilbeau(...) = {
   require(20 <= x1 && x1 <= 100 && -2<= x2 && x2 <= 20)
... }
```

– invertedPendulum

```
def invPendulum(...) = {
   require(0.005 <= s1 && s1 <= 5000 && 0.005 <= s2 && s2 <= 1000 &&
          -0.785 <= s3 && s3 <= -0.005 && -0.785 <= s4 && s4 <= -0.005)
... }
```

– jetEngine

```
def jetEngine(...): Real = {
   require(4 <= x1 && x1 <= 4.65 && 1 <= x2 && x2 <= 6)
... }
```

– kepler

```
def kepler0(...): Real = {
   require(4 <= x1 && x1 <= 6.36 && 0.0001 <= x2 && x2 <= 0.00015 &&
          4.63 <= x3 && x3 <= 6306 && -10 <= x4 && x4 <= -0.01 &&
          4 <= x5 && x5 <= 6.36 && 4 <= x6 && x6 <= 6.36)
... }

def kepler1(...): Real = {
   require(4 <= x1 && x1 <= 6.36 && 0.04 <= x2 && x2 <= 0.0636 &&
          40 <= x3 && x3 <= 6300.6 && 0.001 <= x4 && x4 <= 0.015)
... }
```

```
def kepler2(...): Real = {
   require(4 <= x1 && x1 <= 6.36 && 0 <= x2 && x2 <= 0 &&
          40 <= x3 && x3 <= 63.6 && 0 <= x4 && x4 <= 0 &&
          4 <= x5 && x5 <= 6.36 && 4 <= x6 && x6 <= 6000.36)
... }
```

– rigidBody

```
def rigidBody1(...): Real = {
   require(-1500.0 <= x1 && x1 <= -0.0001 && 0.1 <= x2 && x2 <= 15.0 &&
          -15.0 <= x3 && x3 <= -0.1)
... }

def rigidBody2(...): Real = {
   require(-1500.0 <= x1 && x1 <= -1.125 && -15.0 <= x2 && x2 <= -11.25 &&
          -15.0 <= x3 && x3 <= -11.25)
... }
```

– traincar

```
def state8(...) = {
   require(25 <= s0 && s0 <= 6500 && -2.5 <= s1 && s1 <= 6.5 &&
          -2.5 <= s2 && s2 <= 6.5 && -2.5 <= s3 && s3 <= 6.5 &&
          -2 <= s4 && s4 <= 2750000 && -2 <= s5 && s5 <= 12 &&
          -2 <= s6 && s6 <= 12 && -2 <= s7 && s7 <= 12 &&
          -2 <= s8 && s8 <= 12 && -2 <= y0 && y0 <= 12 &&
          -2 <= y1 && y1 <= 12 && -2 <= y2 && y2 <= 12 &&
          -2 <= y3 && y3 <= 12 && -200000 <= y4 && y4 <= -120000)
... }

def state9(...) = {
   require(-250 <= s0 && s0 <= -0.5 && -2.5 <= s1 && s1 <= 6.5 &&
          -2.5 <= s2 && s2 <= 6.5 && -2.5 <= s3 && s3 <= 6.5 &&
          -900000 <= s4 && s4 <= 2750000 &&
          -2000000 <= s5 && s5 <= -1200000 &&
          -2 <= s6 && s6 <= 12 && -2 <= s7 && s7 <= 12 &&
          -2 <= s8 && s8 <= 12 && -200 <= y0 && y0 <= -120 &&
          200000 <= y1 && y1 <= 1200000 && -2 <= y2 && y2 <= 12 &&
          -2 <= y3 && y3 <= 12 && -2 <= y4 && y4 <= 12)
... }
```

– turbine

```
def turbine1(...): Real = {
   require(-5.5 <= v && v <= -0.3 && 0.001 <= w && w <= 1.9 &&
          3.8 <= r && r <= 10.8)
... }

def turbine2(...): Real = {
   require(-4.5 <= v && v <= -3.3 && -0.4 <= w && w <= -0.01 &&
          3.8 <= r && r <= 16.25)
... }
```

```
def turbine3(...): Real = {
   require(-4.5 <= v && v <= -0.3 && 0.4 <= w && w <= 0.9 &&
          2.85 <= r && r <= 8.5)
... }
```

B.2 Small Input Domains

Univariate benchmarks

– bsplines

```
def bspline0(u: Real): Real = {
   require(0 <= u && u <= 0.5)
... }

def bspline1(u: Real): Real = {
   require(0.875 <= u && u <= 0.9)
... }

def bspline2(u: Real): Real = {
   require(0.5 <= u && u <= 0.75)
... }

def bspline3(u: Real): Real = {
   require(0.125 <= u && u <= 1)
... }
```

– sine

```
def sine(x: Real): Real = {
   require(x > 0.875 && x < 1.57079632679)
... }

def sineOrder3(x: Real): Real = {
   require(-1.5 < x && x < -1.125)
... }
```

– sqroot

```
def sqroot(x: Real): Real = {
   require(x >= 0.0 && x < 1)
... }
```

Multivariate benchmarks

– doppler

```
def doppler(...): Real = {
   require(-100.0 <= u && u <= 100 && 20 <= v && v <= 20000 &&
          -30 <= T && T <= 50)
... }
```

– himmilbeau

```
def himmilbeau(...) = {
   require(0.1 <= x1 && x1 <= 0.5 && -2<= x2 && x2 <= 2)
... }
```

– invertedPendulum

```
def invPendulum(...) = {
   require(0.005 <= s1 && s1 <= 50 && 0.005 <= s2 && s2 <= 10 &&
          -0.785 <= s3 && s3 <= -0.005 && -0.785 <= s4 && s4 <= -0.005)
... }
```

– jetEngine

```
def jetEngine(...): Real = {
   require(4 <= x1 && x1 <= 4.65 && 1 <= x2 && x2 <= 5)
... }
```

– kepler

```
def kepler0(...): Real = {
   require(4 <= x1 && x1 <= 6.36 && 0.0001 <= x2 && x2 <= 0.00011 &&
          40 <= x3 && x3 <= 63.6 && -6.36 <= x4 && x4 <= -4 &&
          4 <= x5 && x5 <= 6.36 && 4 <= x6 && x6 <= 6.36)
... }

def kepler1(...): Real = {
   require(4 <= x1 && x1 <= 6.36 && 0.04 <= x2 && x2 <= 0.0636 &&
          40 <= x3 && x3 <= 63.6 && -6.36 <= x4 && x4 <= -4)
... }

def kepler2(...): Real = {
   require(4 <= x1 && x1 <= 6.36 && 0.0001 <= x2 && x2 <= 0.00011 &&
          40 <= x3 && x3 <= 63.6 && -6.36 <= x4 && x4 <= -4 &&
          4 <= x5 && x5 <= 6.36 && 4 <= x6 && x6 <= 6.36)
... }
```

– rigidBody

```
def rigidBody1(...): Real = {
   require(-15.0 <= x1 && x1 <= -0.1 && 0.1 <= x2 && x2 <= 15.0 &&
          -15.0 <= x3 && x3 <= -0.1)
... }

def rigidBody2(...): Real = {
   require(-15.0 <= x1 && x1 <= -11.25 && -15.0 <= x2 && x2 <= -11.25 &&
          -15.0 <= x3 && x3 <= -11.25)
... }
```

– traincar

```
def state8(...) = {
   require(250 <= s0 && s0 <= 650 && -2.5 <= s1 && s1 <= 6.5 &&
          -2.5 <= s2 && s2 <= 6.5 && -2.5 <= s3 && s3 <= 6.5 &&
           -2 <= s4 && s4 <= 12 && -2 <= s5 && s5 <= 12 &&
           -2 <= s6 && s6 <= 12 && -2 <= s7 && s7 <= 12 &&
           -2 <= s8 && s8 <= 12 && -2 <= y0 && y0 <= 12 &&
           -2 <= y1 && y1 <= 12 && -2 <= y2 && y2 <= 12 &&
           -2 <= y3 && y3 <= 12 && -200000 <= y4 && y4 <= -120000)
... }

def state9(...) = {
   require(-250 <= s0 && s0 <= -0.5 && -2.5 <= s1 && s1 <= 6.5 &&
          -2.5 <= s2 && s2 <= 6.5 && -2.5 <= s3 && s3 <= 6.5 &&
          -2 <= s4 && s4 <= 12 && -2000000 <= s5 && s5 <= -1200000 &&
          -2 <= s6 && s6 <= 12 && -2 <= s7 && s7 <= 12 &&
          -2 <= s8 && s8 <= 12 && -200 <= y0 && y0 <= -120 &&
          200000 <= y1 && y1 <= 1200000 && -2 <= y2 && y2 <= 12 &&
          -2 <= y3 && y3 <= 12 && -2 <= y4 && y4 <= 12)
... }
```

– turbine

```
def turbine1(...): Real = {
   require(-4.5 <= v && v <= -0.3 && 0.4 <= w && w <= 0.9 &&
          3.8 <= r && r <= 7.8)
... }

def turbine2(...): Real = {
   require(-4.5 <= v && v <= -3.3 && -0.4 <= w && w <= -0.1 &&
          3.8 <= r && r <= 7.8)
... }

def turbine3(...): Real = {
   require(-4.5 <= v && v <= -0.3 && 0.4 <= w && w <= 0.9 &&
          3.8 <= r && r <= 7.8)
... }
```