Rigorous Estimation of Floating-Point Round-Off Errors with Symbolic Taylor Expansions

ALEXEY SOLOVYEV, MAREK S. BARANOWSKI, IAN BRIGGS, CHARLES JACOBSEN, ZVONIMIR RAKAMARIĆ, and GANESH GOPALAKRISHNAN, School of Computing, University of Utah, Salt Lake City, UT, USA

Rigorous estimation of maximum floating-point round-off errors is an important capability central to many formal verification tools. Unfortunately, available techniques for this task often provide very pessimistic overestimates, causing unnecessary verification failure. We have developed a new approach called *Symbolic Taylor Expansions* that avoids these problems, and implemented a new tool called FPTaylor embodying this approach. Key to our approach is the use of rigorous global optimization, instead of the more familiar interval arithmetic, affine arithmetic, and/or SMT solvers. FPTaylor emits per-instance analysis certificates in the form of HOL Light proofs that can be machine checked.

In this article, we present the basic ideas behind Symbolic Taylor Expansions in detail. We also survey as well as thoroughly evaluate six tool families, namely, Gappa (two tool options studied), Fluctuat, PRECiSA, Real2Float, Rosa, and FPTaylor (two tool options studied) on 24 examples, running on the same machine, and taking care to find the best options for running each of these tools. This study demonstrates that FPTaylor estimates round-off errors within much tighter bounds compared to other tools on a significant number of case studies. We also release FPTaylor along with our benchmarks, thus contributing to future studies and tool development in this area.

CCS Concepts: • **Software and its engineering → Software verification**; **Compilers**; • **Mathematics of computing → Mathematical optimization**; *Numerical analysis*;

Additional Key Words and Phrases: Floating-point arithmetic, IEEE floating-point standard, mixed-precision arithmetic, round-off error, global optimization, formal verification

ACM Reference format:
Alexey Solovyev, Marek S. Baranowski, Ian Briggs, Charles Jacobsen, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. 2018. Rigorous Estimation of Floating-Point Round-Off Errors with Symbolic Taylor Expansions. *ACM Trans. Program. Lang. Syst.* 41, 1, Article 2 (December 2018), 39 pages.
https://doi.org/10.1145/3230733

1 INTRODUCTION

The floating-point number representation is foundational to computing, playing a central role in the representation and manipulation of real numbers. Unfortunately, floating-point arithmetic suffers from error-inducing rounding operations. That is, after each calculation of a sub-expression,

This work was supported by the National Science Foundation awards CCF 1535032, 1552975, 1643056, and 1704715.
Author's addresses: A. Solovyev, M. S. Baranowski, I. Briggs, C. Jacobsen, Z. Rakamarić, and G. Gopalakrishnan, School of Computing, University of Utah, 50 South Central Campus Drive, Salt Lake City, 84112-9205, Utah; emails: {monad, baranows}@cs.utah.edu, {ianbriggsutah, charlie.jacobsen}@gmail.com, {zvonimir, ganesh}@cs.utah.edu.

0164-0925/2018/12-ART2 $15.00
https://doi.org/10.1145/3230733

the result must be snapped (rounded) to the nearest representable number before the whole expression can be evaluated. The nearest representable number may be half a *unit in the last place* or ulp (in the worst case) away from the calculated (intermediate) result, and this distance is called *round-off error*. The value of the ulp is proportional to the result exponent, making errors value dependent. Given all these subtleties, one seldom analyzes computations directly in floating-point. Instead, one conceives and analyzes computations in the realm of reals, and then ensures that the amount of discrepancy—round-off error—is "small." The manner in which round-off errors are examined and accepted varies from application to application: the more exacting the application, the more rigorous the methods employed must be. The IEEE standard (IEEE 754 2008) was a landmark achievement in computer arithmetic, standardizing the meaning of floating-point operations and the notion of correct rounding. Even systems that abide by the IEEE standard can violate axioms that one takes for granted in the space of real numbers. One such axiom might be $(b \neq c) \Rightarrow ((a - b) \neq (a - c))$. The following session in Python (version 3.5.0) produces these results, where one can take $b = .333333333333333$ and $c = .333333333333334$.

```
>>> (100+(1.0/3)) - .333333333333333
100.0
>>> (100+(1.0/3)) - .333333333333334
100.0

>>> (10+(1.0/3)) - .333333333333333
10.000000000000002
>>> (10+(1.0/3)) - .333333333333334
10.0
```

Very few practitioners have the time or wherewithal to dig into the underlying reasons (e.g., it is known that even the process of printing an answer (Andrysco et al. 2016) introduces rounding).

Some everyday tools can be more befuddling. Excel (version 15.56) produces these answers (the first two cases are discussed in Kahan (2006)) with the rounding rules involved discussed tangentially (Support 2018):

$= (4/3 - 1) * 3 - 1$ prints 0,

$= ((4/3 - 1) * 3 - 1)$ prints -2.22045E-16,

$= (4/3 - 1) * 3 - 1 + 0$ prints -2.22045E-16.

Google sheets used to behave similar to Excel, but now prints -2.22045E-16 even for the first case.

Even in IEEE-standard abiding systems, floating-point arithmetic presents numerous difficulties. Floating-point error analysis is non-compositional. As an example, Kahan (2006) has pointed out that there exist input intervals in which $(e^x - 1)/log(e^x)$ exhibits smaller error than $(e^x - 1)/x$ even though clearly x always exhibits smaller error than $log(e^x)$. In practice, all these boil down to two vexing features of floating-point. First, an entire given expression must be analyzed to find out its maximum round-off error; we cannot, in general, learn much by analyzing the subexpressions. Second, identities true in real arithmetic do not apply to floating-point.

Our Focus. This article develops rigorous methods to estimate the maximum floating-point round-off error in straight-line code sequences. Understanding error analysis properly in this setting is essential before we can meaningfully consider programs with conditionals and loops. Moreover, there are only a handful of tools that conduct error estimation on codes with conditionals and loops, and very few are available for external evaluation. The approaches employed to

handle loops in these tools also differ significantly. These make such comparisons practically difficult.[1] Given that the community has established widely understood criteria for measuring errors in straight-line codes, that will be the focus of our detailed study in this article.

We thoroughly evaluate six tool families, namely, Gappa (two tool options studied), Fluctuat, PRECiSA, Real2Float, Rosa, and FPTaylor (two tool options studied) on 24 examples, running on the same machine. We are also careful to find the best options for running each of these tools, and fully document all these options plus our evaluation platforms. We believe that this is a timely contribution in the light of the recently growing interest in rigorous floating-point analysis.

Currently available techniques for computing round-off errors of even simple straight-line programs often return quite pessimistic overestimates. A typical cause for such poor behavior are the chosen underlying abstractions that lose correlations between round-off errors generated by subexpressions. This can lead to unnecessary verification failures in contexts where one is attempting to prove upper bounds of round-off error. Another limitation of available rigorous techniques is that most of them do not handle transcendental functions other than by analyzing truncated series expansions thereof.

In this article, we present a new approach called *Symbolic Taylor Expansions* that avoids these problems, and we implement it in a tool called FPTaylor. A key feature of our approach is that it employs rigorous global optimization, instead of the more familiar interval arithmetic, affine arithmetic, and/or SMT solvers for error estimation. In addition to providing far tighter upper bounds of round-off error in a vast majority of cases, FPTaylor also emits per-instance analysis certificates in the form of HOL Light proofs that can be machine checked. We note that this ability to check proof certificates is possessed by Gappa, Real2Float (for polynomial programs), and PRECiSA.

Simple Illustrative Example. Suppose one is asked to calculate the maximum round-off error produced by $t/(t+1)$ for $t \in [0, 999]$ being an IEEE-defined 32-bit floating-point number (IEEE 754 2008). This can be achieved in practice in a few minutes by running through all $\sim 2^{32}$ cases by (1) instantiating this expression in two precisions, namely, the requested 32-bit precision, and a much higher precision (say, 128-bit) serving as an approximation for reals, and (2) comparing the answers over the 2^{32} values of the requested floating-point precision. Clearly, this brute-force approach will not scale for more than one variable or even a single double-precision (64-bit) variable t.

While SMT solvers can be used for small problems (Rümmer and Wahl 2010; Haller et al. 2012), the need to scale encourages the use of various abstract interpretation methods (Cousot and Cousot 1977), the most popular choices being interval (Moore 1966) or affine (Stolfi and de Figueiredo 2003) arithmetic. Unfortunately, both these popular methods, while often fast, produce grossly exaggerated error estimates. In many contexts, such overestimates cause unnecessary verification failures or result in programmers over-optimizing code.

For example, suppose one sets about to verify using interval analysis that the error in calculating $t/(t+1)$ in single-precision is less than 0.001. We show in Section 3 that interval analysis would produce an estimate of error equal to 0.030517578125. Affine arithmetic also does not fare well as it is well-known to produce exaggerated answers in the presence of non-linear operators. In reaction to such high error, a programmer might decide to instantiate t in double-precision, and reapply interval analysis, resulting in an error equal to $5.6843418860808015e{-}11$. However, by employing better error analysis, one can avoid un-necessarily switching over to higher precision. For example, using the methods we propose, we can show that the error in $t/(t+1)$ can be rigorously bounded by $1.1920928955078125e{-}07$ even for single-precision.

[1] In Section 6.3, we do provide some preliminary comparative results pertaining to conditionals.

Contributions. Now we summarize our key contributions:

- We describe all the details of our rigorous floating-point round-off error estimation approach based on Symbolic Taylor Expansions and rigorous global optimization, which allows us to reduce the dimensionality of the problem while maintaining critical correlations between round-off errors.
- We release an open source version of our tool FPTaylor.[2] FPTaylor handles all basic floating-point operations and all the binary floating-point formats defined in IEEE 754. It supports transcendental and mixed-precision expressions, uncertainties in input variables, and estimation of relative and absolute round-off errors;[3] it also provides a rigorous treatment of subnormal numbers.
- For the same problem complexity (i.e., number of input variables and expression size), FPTaylor obtains tighter bounds than state-of-the-art tools in most cases, while incurring comparable runtimes. We also empirically verify that our overapproximations are within a factor of 1.9 of the corresponding underapproximations computed using a recent dynamic tool (Chiang et al. 2014).
- FPTaylor has a mode in which it produces HOL Light proof scripts. This facility actually helped us find a bug in our initial tool version. This experience underscores the importance of built-in consistency checking mechanisms, especially bridging tool versions.
- Many tools in this space including FPTaylor are based on back-end global optimizers. We provide a thorough evaluation of FPTaylor on our examples across three different optimizers. Our studies demonstrate the importance of supporting multiple optimizer types that perform differently on different types of examples.

Roadmap. We first provide the necessary background in Section 2 and present a brief overview of our approach and compare it against other existing methods in Section 3. Symbolic Taylor Expansions are presented in Section 4, implementation details in Sections 5 and 6, evaluation in Section 7, related work in Section 8, and conclusions in Section 9. In our Appendix, we provide the following details: (A): a comprehensive results table listing all experimental results, including Fluctuat with and without subdivisions, and FPTaylor under 10 different combinations of rounding models, backend optimizer selections, and optimization problem selections (standard rounding model versus improved rounding model); and (B): an evaluation of the performance of FPTaylor's backend optimizers on several harder benchmarks.

2 BACKGROUND

Floating-Point Arithmetic. The IEEE 754 standard (IEEE 754 2008), concisely formalized in a related article (Goualard 2014), defines a binary floating-point number as a triple of sign (0 or 1), significand, and exponent, i.e., (sgn, sig, exp), with numerical value $(-1)^{sgn} \times sig \times 2^{exp}$. The standard defines four general binary formats with sizes of 16, 32, 64, and 128 bits, varying in constraints on the sizes of sig and exp. The standard also defines special values such as infinities and NaN (not a number). We do not distinguish these values in our work and report them as potential errors. Rounding plays a central role in defining the semantics of floating-point arithmetic. Denote the set of floating-point numbers (in some fixed format) as $\mathbb{F}$. A rounding operator $\mathrm{rnd} : \mathbb{R} \to \mathbb{F}$ is a function which takes a real number and returns a floating-point number which is closest to the

[2] Available at https://github.com/soarlab/FPTaylor.

[3] These terms are defined later, but at a high level the absolute error in evaluating an expression E is the actual difference between the true (real-valued) answer and the floating-point answer, while the relative error divides the absolute error with the true answer.

Table 1. Rounding to Nearest Operator Parameters

Precision (bits)	ϵ	δ
half (16)	2^{-11}	2^{-25}
single (32)	2^{-24}	2^{-150}
double (64)	2^{-53}	$2^{-1,075}$
quad. (128)	2^{-113}	$2^{-16,495}$

input real number and has some special properties defined by the rounding operator. Common rounding operators are rounding to nearest (ties to even), toward zero, and toward $\pm\infty$. A simple model of rounding is given by the following formula (Goldberg 1991; Goualard 2014)

$$\mathrm{rnd}(x) = x(1+e) + d, \tag{1}$$

where $|e| \leq \epsilon$, $|d| \leq \delta$, and $e \times d = 0$. If x is a symbolic expression, then exact numerical values of e and d are not explicitly defined in most cases. (Values of e and d may be known in some cases; for instance, if we know that x is a sufficiently small integer, then $\mathrm{rnd}(x) = x$ and thus $e = d = 0$.) The parameter ϵ specifies the maximal relative error introduced by the given rounding operator. The parameter δ gives the maximal *absolute* error for numbers which are very close to zero (relative error estimation does not work for these small numbers called subnormals). Table 1 shows values of ϵ and δ for the rounding to nearest operator of different floating-point formats. Parameters for other rounding operators can be obtained from Table 1 by multiplying all entries by 2, while Equation (1) applies both for rounding toward zero and toward infinity (by suitably adjusting the ranges of e and d).

The standard precisely defines the behavior of several basic floating-point arithmetic operations. Suppose $op : \mathbb{R}^k \to \mathbb{R}$ is an operation. Let op_{fp} be the corresponding floating-point operation. Then the operation op_{fp} is correctly rounded if the following equation holds for all floating-point values $x_1, \ldots, x_k$:

$$op_{\mathrm{fp}}(x_1, \ldots, x_k) = \mathrm{rnd}(op(x_1, \ldots, x_k)). \tag{2}$$

The following operations must be correctly rounded according to the standard: $+, -, \times, /, \sqrt{}$, fma. (Here, $\mathrm{fma}(a, b, c)$ is a ternary *fused multiply-add* operation that computes $a \times b + c$ with a single rounding.)

Combining Equations (1) and (2), we get a simple model of floating-point arithmetic which is valid in the absence of overflows and invalid operations:

$$op_{\mathrm{fp}}(x_1, \ldots, x_k) = op(x_1, \ldots, x_k)(1+e) + d. \tag{3}$$

There are some special cases where the model given by Equation (3) can be improved. For instance, if op is "$-$" or "$+$," then $d = 0$ (Goualard 2014). Also, if op is "$\times$" and one of the arguments is a non-negative power of 2, then $e = d = 0$. These and several other special cases are implemented in FPTaylor to improve the quality of the error analysis.

Equation (3) can be used even with operations that are not correctly rounded. For example, most implementations of floating-point transcendental functions are not correctly rounded but they yield results which are very close to correctly rounded results (Harrison 2000). As another example, the technique introduced by Bingham and Leslie-Hurd (Bingham and Leslie-Hurd 2014) can verify relative error bounds of hardware implementations of transcendental functions. In all such cases, we can still use Equation (3) to model transcendental functions, but we need to increase values of ϵ and δ appropriately. In addition, there exist software libraries that compute

correctly rounded values of transcendental functions (Daramy et al. 2003; Fousse et al. 2007). For such libraries, Equation (3) can be applied without any changes to the values of ϵ and δ.

Taylor Expansion. A Taylor expansion is a well-known formula for approximating an arbitrary sufficiently smooth function with a polynomial expression. In this work, we use the first-order Taylor approximation with the second-order error term. Higher-order Taylor approximations are possible but they lead to complex expressions for second- and higher-order derivatives and do not give much better approximation results (Neumaier 2003). Suppose $f(x_1, \ldots, x_k)$ is a twice continuously differentiable multivariate function on an open convex domain $D \subset \mathbb{R}^k$. Note that the open convex domain restriction is only required for the general Taylor's theorem. We apply this theorem to elementary functions to derive our rules described in Section 5. As a consequence, our rules are in general not restricted to any domain, except when we, for example, divide by zero or take *arcsin* of an argument outside $[-1, 1]$. For any fixed point $\mathbf{a} \in D$ (we use bold symbols to represent vectors), the following formula holds (e.g., see Theorem 3.3.1 in Mikusinski and Taylor (2002)):

$$f(\mathbf{x}) = f(\mathbf{a}) + \sum_{i=1}^{k} \frac{\partial f}{\partial x_i}(\mathbf{a})(x_i - a_i) + \frac{1}{2} \sum_{i,j=1}^{k} \frac{\partial^2 f}{\partial x_i \partial x_j}(\mathbf{p})(x_i - a_i)(x_j - a_j). \tag{4}$$

Here, $\mathbf{p} \in D$ is a point which depends on $\mathbf{x}$ and $\mathbf{a}$.

Later we will consider functions with arguments $\mathbf{x}$ and $\mathbf{e}$ defined by $f(\mathbf{x}, \mathbf{e}) = f(x_1, \ldots, x_n, e_1, \ldots, e_k)$. We will derive Taylor expansions of these functions with respect to variables $e_1, \ldots, e_k$:

$$f(\mathbf{x}, \mathbf{e}) = f(\mathbf{x}, \mathbf{a}) + \sum_{i=1}^{k} \frac{\partial f}{\partial e_i}(\mathbf{x}, \mathbf{a})(e_i - a_i) + R_2(\mathbf{x}, \mathbf{e}). \tag{5}$$

In this expansion, variables $x_1, \ldots, x_n$ appear in coefficients $\frac{\partial f}{\partial e_i}$ thereby producing Taylor expansions with symbolic coefficients.

3 OVERVIEW OF OUR APPROACH

We now detail the simple example from Section 1 on estimating the worst case absolute round-off error in the expression $t/(t+1)$. Our goal is to illustrate the difficulties faced by interval and affine methods used by themselves, and to bring out many of the key ideas underlying our work. In our example, $t \in [0, 999]$ is a floating-point number, and *absolute round-off error* is defined as $err_{\text{abs}} = |\tilde{v} - v|$, where $\tilde{v}$ is the result of floating-point computations and v is the result of corresponding exact mathematical computations. Let $\oslash$ and $\oplus$ denote floating-point operations corresponding to "/" and "+."

Suppose interval abstraction were used to analyze this example. The round-off error of $t \oplus 1$ can be estimated by 512ϵ where ϵ is the machine epsilon (which bounds the maximum relative error of basic floating-point operations such as $\oplus$ and $\oslash$) and the number $512 = 2^9$ is the largest power of 2 which is less than $1,000 = 999 + 1$. Interval abstraction replaces the expression $d = t \oplus 1$ with the abstract pair $([1, 1,000], 512\epsilon)$ where the first component is the interval of all possible values of d and 512ϵ is the associated round-off error. Now we need to calculate the round-off error of $t \oslash d$. It can be shown that one of the primary sources of errors in this expression is attributable to the propagation of error in $t \oplus 1$ into the division operator. The propagated error is computed by multiplying the error in $t \oplus 1$ by $\frac{t}{d^2}$.[4] At this point, interval abstraction does not yield a satisfactory

[4]Ignoring the round-off division error, one can view $t \oslash d$ as $t/(d_{\text{exact}} + \delta)$ where δ is the round-off error in d. Apply Taylor approximation which yields as the first two terms $(t/d_{\text{exact}}) - (t/(d^2_{\text{exact}}))\delta$.

result since it computes $\frac{t}{d^2}$ by setting the numerator t to 999 and the denominator d to 1. Therefore, the total error bound is computed as $999 \times 512\epsilon \approx 512{,}000\epsilon$. This works out to be 0.030517578125 for single precision and $5.6843418860808015e{-}11$ for double precision (see Table 1).

The main weakness of the interval abstraction is that it does not preserve variable relationships (e.g., the two t's may be independently set to 999 and 0). In the example above, the abstract representation of d was too coarse to yield a good final error bound (we suffer from eager composition of abstractions). While affine arithmetic is more precise since it remembers linear dependencies between variables, it still does not handle our example well as it contains division, a nonlinear operator (for which affine arithmetic is known to be a poor fit).

A better approach is to *model the error at each subexpression position* and *globally solve for maximal error*—as opposed to merging the worst-cases of local abstractions, as happens in the interval abstraction usage above. Following this approach, a simple way to get a much better error estimate is the following. Consider a simple model for floating-point arithmetic. Write $t \oplus 1 = (t+1)(1+\epsilon_1)$ and $t \oslash (t \oplus 1) = (t/(t \oplus 1))(1+\epsilon_2)$ with $|\epsilon_1| \leq \epsilon$ and $|\epsilon_2| \leq \epsilon$. Now, compute the first-order Taylor approximation of our expression with respect to ϵ_1 and ϵ_2 by taking ϵ_1 and ϵ_2 as the perturbations around t, and computing partial derivatives with respect to them (see Equations (4) and (5) for a recap):

$$t \oslash (t \oplus 1) = \frac{t(1+\epsilon_2)}{(t+1)(1+\epsilon_1)} = \frac{t}{t+1} - \frac{t}{t+1}\epsilon_1 + \frac{t}{t+1}\epsilon_2 + O(\epsilon^2).$$

(Here $t \in [0, 999]$ is fixed and hence we do not divide by zero.) It is important to keep all coefficients in the above Taylor expansion as symbolic expressions depending on the input variable t. The difference between $t/(t+1)$ and $t \oslash (t \oplus 1)$ can be easily estimated (we ignore the term $O(\epsilon^2)$ in this motivating example but later in Section 4 we demonstrate how rigorous upper bounds are derived for all error terms):

$$\left|-\frac{t}{t+1}\epsilon_1 + \frac{t}{t+1}\epsilon_2\right| \leq \left|\frac{t}{t+1}\right||\epsilon_1| + \left|\frac{t}{t+1}\right||\epsilon_2| \leq 2\left|\frac{t}{t+1}\right|\epsilon.$$

The only remaining task now is finding a bound for the expression $t/(t+1)$ for all $t \in [0, 999]$. Simple interval computations as above yield $t/(t+1) \in [0, 999]$. The error can now be estimated by 1998ϵ, which is already a much better bound than before. We go even further and apply a global optimization procedure to maximize $t/(t+1)$ and compute an even better bound, i.e., $t/(t+1) \leq 1$ for all $t \in [0, 999]$. Thus, the error is bounded by 2ϵ. This works out to be $1.1920928955078125e{-}07$. The combination of Taylor expansion with symbolic coefficients and global optimization yields an error bound which is $512{,}000/2 = 256{,}000$ times better than a naive error estimation technique implemented in many other tools for floating-point analysis. Our error estimation approach has the added advantage of avoiding the explicit modeling of the operators involved in the problem being analyzed ("/" and "+" in our example); functions underlying these operators are handled by the backend global optimizer.

4 SYMBOLIC TAYLOR EXPANSIONS

In this section, we present Symbolic Taylor Expansions at a high level, and then discuss how error estimation is regarded as an optimization problem (Section 4.1), how relative errors are computed (Section 4.2), mixed-precision support (Section 4.3), and FPTaylor's improved rounding model (Section 4.4). A deep-dive into how exactly Symbolic Taylor Forms are derived is then provided in Section 5.

Given a function $f : \mathbb{R}^n \to \mathbb{R}$, the goal of the Symbolic Taylor Expansions approach is to estimate the round-off error when f is realized in floating-point. We assume that the arguments of the function belong to a bounded domain I, i.e., $\mathbf{x} \in I$. In general, the domain I can be quite arbitrary;

the only requirement is that it is bounded and the function f is defined everywhere on this domain. In FPTaylor, the domain I is defined with inequalities over input variables. In our benchmarks as well as our implementation of FPTaylor presented later, we have $a_i \le x_i \le b_i$ for all $i = 1, \ldots, n$. In this case, $I = [a_1, b_1] \times \cdots \times [a_n, b_n]$ is a product of intervals.

Let $\mathrm{fp}(f) : \mathbb{R}^n \to \mathbb{F}$ be a function derived from f where all operations, variables, and constants are replaced with the corresponding floating-point operations, variables, and constants. Our goal is to compute the following round-off error:

$$\mathrm{err}_{\mathrm{fp}}(f, I) = \max_{\mathbf{x} \in I} |\mathrm{fp}(f)(\mathbf{x}) - f(\mathbf{x})|. \tag{6}$$

The optimization problem in Equation (6) is computationally hard and not supported by most classical optimization methods as it involves a highly irregular and discontinuous function $\mathrm{fp}(f)$. The most common way of overcoming such difficulties is to consider abstract models of floating-point arithmetic that approximate floating-point results with real numbers. Section 2 presented the following model of floating-point arithmetic (see Equation (3)):

$$op_{\mathrm{fp}}(x_1, \ldots, x_n) = op(x_1, \ldots, x_n)(1 + e) + d.$$

Values of e and d depend on the rounding mode and the operation itself. Special care must be taken in case of exceptions (overflows or invalid operations). Our tool can detect and report such exceptions.

First, we replace all floating-point operations in the function $\mathrm{fp}(f)$ with the right-hand side of Equation (3). Constants and variables also need to be replaced with rounded values, unless they can be exactly represented with floating-point numbers. We get a new function $\tilde{f}(\mathbf{x}, \mathbf{e}, \mathbf{d})$ which has all the original arguments $\mathbf{x} = (x_1, \ldots, x_n) \in I$, but also the additional arguments $\mathbf{e} = (e_1, \ldots, e_k)$ and $\mathbf{d} = (d_1, \ldots, d_k)$ where k is the number of potentially inexact floating-point operations (plus constants and variables) in $\mathrm{fp}(f)$. Note that $\tilde{f}(\mathbf{x}, \mathbf{0}, \mathbf{0}) = f(\mathbf{x})$. Also, $\tilde{f}(\mathbf{x}, \mathbf{e}, \mathbf{d}) = \mathrm{fp}(f)(\mathbf{x})$ for some choice of $\mathbf{e}$ and $\mathbf{d}$. Now, the difficult optimization problem in Equation (6) can be replaced with the following simpler optimization problem that overapproximates it:

$$\mathrm{err}_{\mathrm{overapprox}}(\tilde{f}, I) = \max_{\mathbf{x} \in I, |e_i| \le \epsilon, |d_i| \le \delta} |\tilde{f}(\mathbf{x}, \mathbf{e}, \mathbf{d}) - f(\mathbf{x})|. \tag{7}$$

Note that for any I, $\mathrm{err}_{\mathrm{fp}}(f, I) \le \mathrm{err}_{\mathrm{overapprox}}(\tilde{f}, I)$. However, even this optimization problem is still hard because we have $2k$ new variables e_i and d_i for (inexact) floating-point operations in $\mathrm{fp}(f)$. We further simplify the optimization problem using Taylor expansion.

We know that $|e_i| \le \epsilon$, $|d_i| \le \delta$, and ϵ, δ are small. Define $y_1 = e_1, \ldots, y_k = e_k, y_{k+1} = d_1, \ldots, y_{2k} = d_k$. Consider the Taylor formula (see Equation (5)) with the second-order error term of $\tilde{f}(\mathbf{x}, \mathbf{e}, \mathbf{d})$ with respect to $e_1, \ldots, e_k, d_1, \ldots, d_k$.

$$\tilde{f}(\mathbf{x}, \mathbf{e}, \mathbf{d}) = \tilde{f}(\mathbf{x}, \mathbf{0}, \mathbf{0}) + \sum_{i=1}^{k} \frac{\partial \tilde{f}}{\partial e_i}(\mathbf{x}, \mathbf{0}, \mathbf{0}) e_i + R_2(\mathbf{x}, \mathbf{e}, \mathbf{d}) \tag{8}$$

with

$$R_2(\mathbf{x}, \mathbf{e}, \mathbf{d}) = \frac{1}{2} \sum_{i,j=1}^{2k} \frac{\partial^2 \tilde{f}}{\partial y_i \partial y_j}(\mathbf{x}, \mathbf{p}) y_i y_j + \sum_{i=1}^{k} \frac{\partial \tilde{f}}{\partial d_i}(\mathbf{x}, \mathbf{0}, \mathbf{0}) d_i$$

for some $\mathbf{p} \in \mathbb{R}^{2k}$ such that $|p_i| \le \epsilon$ for $i = 1, \ldots, k$ and $|p_i| \le \delta$ for $i = k + 1, \ldots, 2k$. Note that we added first-order terms $\frac{\partial \tilde{f}}{\partial d_i}(\mathbf{x}, \mathbf{0}, \mathbf{0}) d_i$ to the error term R_2 because $\delta = O(\epsilon^2)$ (see Table 1; in fact, δ is much smaller than ϵ^2).

We have $\tilde{f}(\mathbf{x}, \mathbf{0}, \mathbf{0}) = f(\mathbf{x})$ and hence the error from Equation (7) can be determined as follows:

$$\mathrm{err}_{\mathrm{overapprox}}(\tilde{f}, I) \leq \max_{\mathbf{x} \in I, |e_i| \leq \epsilon} \left| \sum_{i=1}^{k} \frac{\partial \tilde{f}}{\partial e_i}(\mathbf{x}, \mathbf{0}, \mathbf{0}) e_i \right| + M_2, \tag{9}$$

where M_2 is an upper bound for the error term $R_2(\mathbf{x}, \mathbf{e}, \mathbf{d})$. In our work, we use simple methods to estimate the value of M_2, such as interval arithmetic or several iterations of a global optimization algorithm. We always derive a rigorous bound of $R_2(\mathbf{x}, \mathbf{e}, \mathbf{d})$ and this bound is small in general since it contains an ϵ^2 factor. Large values of M_2 (relative to the first term in Equation (9)) may indicate serious stability problems—for instance, the denominator of some expression is very close to zero. Our tool issues a warning if the computed value of M_2 is large.

Next, we note that in Equation (9) the maximized expression depends on e_i linearly and it achieves its maximum value when $e_i = \pm\epsilon$. Therefore, the expression attains its maximum when the sign of e_i is the same as the sign of the corresponding partial derivative, and we transform the maximized expression into the sum of absolute values of partial derivatives. Finally, we get the following optimization problem:

$$\mathrm{err}_{\mathrm{fp}}(f, I) \leq \mathrm{err}_{\mathrm{overapprox}}(\tilde{f}, I) \leq M_2 + \epsilon \max_{\mathbf{x} \in I} \sum_{i=1}^{k} \left| \frac{\partial \tilde{f}}{\partial e_i}(\mathbf{x}, \mathbf{0}, \mathbf{0}) \right|. \tag{10}$$

The solution of our original, almost intractable problem (i.e., estimation of the floating-point error $\mathrm{err}_{\mathrm{fp}}(f, I)$ mentioned in Equation (6)) is reduced to the following two much simpler subproblems: (i) compute all expressions and constants involved in the optimization problem in Equation (10), and (ii) solve the optimization problem in Equation (10).

In our implementation, we do not compute partial derivatives directly. Instead, we use special rules that produce the final symbolic expressions as described in detail in Section 5. There are two advantages to having these rules. First, the rules help systematically derive the partial derivatives. Second, our tool supports an improved rounding model (see Section 4.4) that introduces discontinuous functions for which partial derivatives cannot be computed; our special rules help overcome this difficulty.

4.1 Solving Optimization Problems

We compute error bounds using rigorous global optimization techniques (Neumaier 2004). In general, it is not possible to find an exact optimal value of a given real-valued function. The main property of rigorous global optimization methods is that they always return a rigorous bound for a given optimization problem (some conditions on the optimized function are necessary such as continuity or differentiability). These methods can also balance between accuracy and performance. They can either return an estimation of the optimal value with the given tolerance or return a rigorous upper bound after a specific amount of time (iterations).

It is also important to note that we are optimizing real-valued expressions, not floating-point ones. A particular global optimizer can work with floating-point numbers internally but it must return a rigorous result (i.e., one that overapproximates the optimum). For instance, the optimal maximal floating-point value of the function $f(x) = 0.3$ is not 0.3 (since this constant is not exactly FP-representable); instead it is the smallest floating-point number r which is greater than 0.3. This ensures that the real-valued bound is below the given answer. It is known that global optimization is a hard problem. But note that abstraction techniques based on interval or affine arithmetic can be considered as primitive (and generally overly conservative) global optimization methods. FPTaylor can use any existing global optimization method to derive rigorous bounds of error expressions,

and hence it is possible to run it with very conservative but fast global optimization technique if necessary (Table 3, in fact, lists three optimizers that FPTaylor supports).

The optimization problem in Equation (10) depends only on input variables of the function f, but it also contains a sum of absolute values of functions. Hence, it is not trivial—some global optimizers may not accept absolute values since they are not smooth functions. In addition, even if an optimizer accepts absolute values, they make the optimization problem considerably harder.

There is a naive approach to simplify and solve this optimization problem. Find minimum (y_i) and maximum (z_i) values for each term $\frac{\partial \tilde{f}}{\partial e_i}(\mathbf{x}, \mathbf{0}, \mathbf{0})$ separately. Let $s_i(\mathbf{x}) = \frac{\partial \tilde{f}}{\partial e_i}(\mathbf{x}, \mathbf{0}, \mathbf{0})$. Now, imagine computing the following:

$$\max_{\mathbf{x} \in I} \sum_{i=1}^{k} |s_i(\mathbf{x})| \leq \sum_{i=1}^{k} \max_{\mathbf{x} \in I} |s_i(\mathbf{x})| = \sum_{i=1}^{k} \max\{-y_i, z_i\}. \tag{11}$$

This result can sometimes be overly conservative, but in most cases it is close to the optimal result as our experimental results demonstrate (see Section 7). This leads to the two approaches (`decomposed` versus `monolithic`) discussed in Table 3 (the `decomposed` approach allows for the separate maximizations suggested in Equation (11)).

We also apply global optimization to compute a conservative range of the expression for which we estimate the round-off error (i.e., the range of the function f). By combining this range information with the bound of the absolute round-off error computed from Equation (10), we can get a rigorous estimation of the range of $\mathrm{fp}(f)$. The range of $\mathrm{fp}(f)$ is useful for verification of program assertions and proving the absence of floating-point exceptions such as overflows or divisions by zero. In addition, FPTaylor computes ranges of intermediate expressions. By default, these ranges are computed with simple interval arithmetic, but there is also an option to compute them with global optimization backends. In FPTaylor, potential runtime errors are checked with simple interval arithmetic before the construction of Taylor forms. However, FPTaylor does not check for all runtime errors accurately (this is not the main goal of FPTaylor).

4.2 Relative Error

It is easy to derive the relative error estimation method from our formulas for absolute errors. The relative error is computed as

$$\mathrm{err}_{\mathrm{rel\_fp}}(f, I) = \max_{\mathbf{x} \in I} \left| \frac{\mathrm{fp}(f)(\mathbf{x}) - f(\mathbf{x})}{f(\mathbf{x})} \right|. \tag{12}$$

Replace $\mathrm{fp}(f)(\mathbf{x})$ with $\tilde{f}(\mathbf{x}, \mathbf{e}, \mathbf{d})$ and use Equation (8) to get the following overapproximation of the relative error:

$$\mathrm{err}_{\mathrm{rel\_overapprox}}(\tilde{f}, I) \leq \max_{\mathbf{x} \in I, |e_i| \leq \epsilon} \left| \sum_{i=1}^{k} \left(\frac{\partial \tilde{f}}{\partial e_i}(\mathbf{x}, \mathbf{0}, \mathbf{0}) \Big/ f(\mathbf{x}) \right) e_i \right| + \max_{\mathbf{x} \in I} \frac{M_2}{|f(\mathbf{x})|}. \tag{13}$$

Here, M_2 is exactly the same as in Equation (9). The final optimization problem for the relative error is

$$\mathrm{err}_{\mathrm{rel\_fp}}(f, I) \leq \mathrm{err}_{\mathrm{rel\_overapprox}}(\tilde{f}, I) \leq \max_{\mathbf{x} \in I} \frac{M_2}{|f(\mathbf{x})|} + \epsilon \max_{\mathbf{x} \in I} \sum_{i=1}^{k} \left| \frac{\partial \tilde{f}}{\partial e_i}(\mathbf{x}, \mathbf{0}, \mathbf{0}) \Big/ f(\mathbf{x}) \right|. \tag{14}$$

We can also derive a simplified optimization problem similar to Equation (11). Both Equation (14) and the corresponding simplified optimization problems are implemented in FPTaylor.

Note that the relative error can be estimated only for functions which are not equal to 0 for all input arguments.

4.3 Mixed Precision

We derived Equation (10) under the assumption that all rounding operations have the same precision. It is easy to derive a general optimization problem for mixed precision computations.

Suppose that each error variable e_i is bounded by ϵ_i: $|e_i| \leq \epsilon_i$. Without loss of generality, assume that $\epsilon_1 = \min\{\epsilon_1, \ldots, \epsilon_k\}$. Then the optimization problem given by Equation (10) can be rewritten in the following way:

$$\mathrm{err}_{\mathrm{fp}}(f, I) \leq M_2 + \epsilon_1 \max_{\mathbf{x} \in I} \sum_{i=1}^{k} \frac{\epsilon_i}{\epsilon_1} \left| \frac{\partial \tilde{f}}{\partial e_i}(\mathbf{x}, \mathbf{0}, \mathbf{0}) \right|. \tag{15}$$

FPTaylor has been used to verify results of the mixed precision synthesis tool FPTuner (Chiang et al. 2017).

4.4 Improved Rounding Model

The rounding model described by Equations (1) and (3) is not tight. For example, if we round a real number $x \in (8, 16]$, then Equation (1) yields $\mathrm{rnd}(x) = x + xe$ with $|e| \leq \epsilon$. A tighter bound for the same e would be $\mathrm{rnd}(x) = x + 8e$. This more precise rounding model follows from the fact that floating-point numbers have the same distance between each other in the interval $[2^n, 2^{n+1}]$ for integer n. These lead to the options `standard` versus `improved` discussed in Table 3.

We now show how to implement this improved rounding model. Define $p_2(x) = \max_{n \in \mathbb{Z}}\{2^n \mid 2^n < x\}$ for $x > 0$, $p_2(0) = 0$, and $p_2(x) = -p_2(-x)$ for $x < 0$. Now we can rewrite Equations (1) and (3) as

$$\begin{aligned} \mathrm{rnd}(x) &= x + p_2(x)e + d, \\ op_{\mathrm{fp}}(x_1, \ldots, x_k) &= op(x_1, \ldots, x_k) + p_2(op(x_1, \ldots, x_k))e + d. \end{aligned} \tag{16}$$

The function p_2 is piecewise constant. The improved model yields optimization problems with discontinuous functions p_2. These problems are harder than optimization problems for the original rounding model and can be solved with branch and bound algorithms based on rigorous interval arithmetic (see Section 6.1).

5 DERIVING SYMBOLIC TAYLOR FORMS

We now present the technical details of deriving symbolic Taylor forms and the accompanying correctness proofs.

Definitions. We want to estimate the round-off error in computation of a function $f : \mathbb{R}^n \to \mathbb{R}$ on a domain $I \subset \mathbb{R}^n$. The round-off error at a point $\mathbf{x} \in I$ is defined as the difference $\mathrm{fp}(f)(\mathbf{x}) - f(\mathbf{x})$ and $\mathrm{fp}(f)$ is the function f where all operations (constants, variable, respectively) are replaced with floating-point operations (constants, variables, respectively). Inductive rules which define $\mathrm{fp}(f)$ are the following:

$$\begin{aligned} \mathrm{fp}(x) &= x, \ x \text{ is a floating-point variable or constant}, \\ \mathrm{fp}(x) &= \mathrm{rnd}(x), \ x \text{ is a real variable or constant}, \\ \mathrm{fp}(op(f_1, \ldots, f_r)) &= \mathrm{rnd}(op(\mathrm{fp}(f_1), \ldots, \mathrm{fp}(f_r))), \\ &\quad \text{where } op \text{ is } +, -, \times, /, \sqrt{}, \mathrm{fma}. \end{aligned} \tag{17}$$

The definition of $\mathrm{fp}(\sin(f))$ and other transcendental functions is implementation dependent and it is not defined by the IEEE 754 standard. Nevertheless, it is possible to consider the same approximation model of $\mathrm{fp}(\sin(f))$ as in Equation (3) with slightly larger bounds for e and d.

$$\mathsf{CONST}\ \frac{c}{\langle c, []\rangle} \qquad \mathsf{CONST_{RND}}\ \frac{\mathrm{rnd}(c)}{\langle c, [f_{\mathrm{err}}(c)]\rangle}$$

$$\mathsf{VAR}\ \frac{x}{\langle x, []\rangle} \qquad \mathsf{VAR_{RND}}\ \frac{\mathrm{rnd}(x)}{\langle x, [f_{\mathrm{err}}(x)]\rangle}$$

$$\mathsf{RND}\ \frac{\langle f, \mathsf{s}\rangle}{\left\langle f, [f] \mathbin{@} \mathsf{s} \mathbin{@} [\epsilon M_2 + \frac{\delta}{\epsilon}]\right\rangle,\ \textbf{where } M_2 \geq \max\limits_{\mathbf{x}\in I}\left(\sum_i |s_i(\mathbf{x})|\right)}$$

$$\mathsf{ADD}\ \frac{\langle f, \mathsf{s}\rangle, \langle g, \mathsf{t}\rangle}{\langle f + g, \mathsf{s} \mathbin{@} \mathsf{t}\rangle} \qquad \mathsf{SUB}\ \frac{\langle f, \mathsf{s}\rangle, \langle g, \mathsf{t}\rangle}{\langle f - g, \mathsf{s} \mathbin{@} [-t_j]_j\rangle}$$

$$\mathsf{MUL}\ \frac{\langle f, \mathsf{s}\rangle, \quad \langle g, \mathsf{t}\rangle}{\langle f \times g, [f \times t_j]_j \mathbin{@} [g \times s_i]_i \mathbin{@} [\epsilon M_2]\rangle,\ \textbf{where } M_2 \geq \max\limits_{\mathbf{x}\in I}\left(\sum\limits_{i,j}|t_j(\mathbf{x})s_i(\mathbf{x})|\right)}$$

$$\mathsf{INV}\ \frac{\langle f, \mathsf{s}\rangle}{\left\langle \frac{1}{f}, [-\frac{s_i}{f^2}]_i \mathbin{@} [\epsilon M_2]\right\rangle,\ \textbf{where } M_2 \geq \max\limits_{\mathbf{x}\in I, |e_i|\leq\epsilon}\left(\sum\limits_{i,j}\left|\frac{s_i(\mathbf{x})s_j(\mathbf{x})}{(f(\mathbf{x})+\sum\limits_k s_k(\mathbf{x})e_k)^3}\right|\right)}$$

$$\mathsf{SQRT}\ \frac{\langle f, \mathsf{s}\rangle}{\left\langle \sqrt{f}, [\frac{s_i}{2\sqrt{f}}]_i \mathbin{@} [\epsilon M_2]\right\rangle,\ \textbf{where } M_2 \geq \max\limits_{\mathbf{x}\in I, |e_i|\leq\epsilon}\left(\frac{1}{8}\sum\limits_{i,j}\left|\frac{s_i(\mathbf{x})s_j(\mathbf{x})}{(f(\mathbf{x})+\sum\limits_k s_k(\mathbf{x})e_k)^{3/2}}\right|\right)}$$

$$\mathsf{SIN}\ \frac{\langle f, \mathsf{s}\rangle}{\langle \sin f, [s_i \cos f]_i \mathbin{@} [\epsilon M_2]\rangle,\ \textbf{where } M_2 \geq \max\limits_{\mathbf{x}\in I, |e_i|\leq\epsilon}\left(\frac{1}{2}\sum\limits_{i,j}\left|\sin(f(\mathbf{x})+\sum\limits_k s_k(\mathbf{x})e_k)s_i(\mathbf{x})s_j(\mathbf{x})\right|\right)}$$

Fig. 1. Derivation rules of symbolic Taylor forms.

Use Equation (1) to construct a function $\tilde{f}(\mathbf{x}, \mathbf{e}, \mathbf{d})$ from $\mathrm{fp}(f)$. The function $\tilde{f}$ approximates $\mathrm{fp}(f)$ in the following precise sense:

$$\forall \mathbf{x} \in I,\ \exists \mathbf{e} \in D_\epsilon, \mathbf{d} \in D_\delta,\ \mathrm{fp}(f)(\mathbf{x}) = \tilde{f}(\mathbf{x}, \mathbf{e}, \mathbf{d}), \tag{18}$$

where ϵ and δ are upper bounds of the corresponding error terms in the model in Equation (1). Here, $D_\alpha = \{\mathbf{y} \mid |y_i| \leq \alpha\}$, i.e., $\mathbf{e} \in D_\epsilon$ means $|e_i| \leq \epsilon$ for all i; likewise, $\mathbf{d} \in D_\delta$ means $|d_j| \leq \delta$ for all j.

We have the following Taylor expansion of $\tilde{f}(\mathbf{x}, \mathbf{e}, \mathbf{d})$:

$$\tilde{f}(\mathbf{x}, \mathbf{e}, \mathbf{d}) = f(\mathbf{x}) + \sum_{i=1}^{k} s_i(\mathbf{x})e_i + R_2(\mathbf{x}, \mathbf{e}, \mathbf{d}). \tag{19}$$

Here we denote $s_i = \frac{\partial \tilde{f}}{\partial e_i}$. We also include the effect of subnormal computations captured by $\mathbf{d}$ in the second-order error term. We can include all variables d_j in $R_2(\mathbf{x}, \mathbf{e}, \mathbf{d})$ since $\delta = O(\epsilon^2)$ (in fact, δ is much smaller than ϵ^2). Rules for computing a rigorous upper bound of $R_2(\mathbf{x}, \mathbf{e}, \mathbf{d})$ are presented in Figure 1.

Equation (19) is inconvenient from the point of view of Taylor expansion derivation as it differentiates between first- and second-order error terms. Let $M_2 \in \mathbb{R}$ be such that $|R_2(\mathbf{x}, \mathbf{e}, \mathbf{d})| \leq M_2$ for all $\mathbf{x} \in I$, $\mathbf{e} \in D_\epsilon$, and $\mathbf{d} \in D_\delta$. In practice, we estimate M_2 using interval arithmetic by default, but one can select global optimization as well. Define $s_{k+1}(\mathbf{x}) = \frac{M_2}{\epsilon}$. Then the following formula holds:

$$\forall \mathbf{x} \in I, \mathbf{e} \in D_\epsilon, \mathbf{d} \in D_\delta,\ \exists e_{k+1}, |e_{k+1}| \leq \epsilon \wedge R_2(\mathbf{x}, \mathbf{e}, \mathbf{d}) = s_{k+1}(\mathbf{x}) e_{k+1}. \tag{20}$$

This formula follows from the simple fact that $|\frac{R_2(\mathbf{x},\mathbf{e},\mathbf{d})}{s_{k+1}(\mathbf{x})}| \leq \frac{M_2}{s_{k+1}(\mathbf{x})} = \epsilon$. Next, we substitute Equation (19) into Equation (18), find e_{k+1} from Equation (20), and replace $R_2(\mathbf{x}, \mathbf{e}, \mathbf{d})$ with $s_{k+1}(\mathbf{x}) e_{k+1}$. We get the following identity:

$$\forall \mathbf{x} \in I,\ \exists e_1, \ldots, e_{k+1},\ |e_i| \leq \epsilon \wedge \mathrm{fp}(f)(\mathbf{x}) = f(\mathbf{x}) + \sum_{i=1}^{k+1} s_i(\mathbf{x}) e_i. \tag{21}$$

The identity in Equation (21) does not include variables $\mathbf{d}$. The effect of these variables is accounted for in the expression $s_{k+1}(\mathbf{x}) e_{k+1}$.

We introduce the following data structure and notation. Let $\langle f, \mathrm{s} \rangle$ be a pair of a symbolic expression f (we do not distinguish between a function f and its symbolic expression) and a list $\mathrm{s} = [s_1; \ldots; s_r]$ of symbolic expressions s_i. We call the pair $\langle f, \mathrm{s} \rangle$ a *Taylor form*. We also use capital letters to denote Taylor forms, for example, $F = \langle f, \mathrm{s} \rangle$. For any function $h(\mathbf{x})$, we write $h \sim \langle f, \mathrm{s} \rangle$ if and only if

$$\forall \mathbf{x} \in I,\ \exists \mathbf{e} \in D_\epsilon,\ h(\mathbf{x}) = f(\mathbf{x}) + \sum_{i=1}^{r} s_i(\mathbf{x}) e_i. \tag{22}$$

If $h \sim \langle f, \mathrm{s} \rangle$, we say that $\langle f, \mathrm{s} \rangle$ corresponds to h. We are interested in Taylor forms $\langle f, \mathrm{s} \rangle$ corresponding to $\mathrm{fp}(f)$. Note that the expression on the right-hand side of Equation (22) is similar to an affine form where all coefficients are symbolic expressions and the noise symbols e_i are restricted to the interval $[-\epsilon, \epsilon]$.

Rules. Our goal is to derive a Taylor form F corresponding to $\mathrm{fp}(f)$ from the symbolic expression of $\mathrm{fp}(f)$. This derivation is done by induction on the structure of $\mathrm{fp}(f)$. Figure 1 shows main derivation rules of Taylor forms. In this figure, the operation @ concatenates two lists and [] denotes the empty list. The notation $[-t_j]_j$ means $[-t_1; \ldots; -t_r]$ where r is the length of the corresponding list.

Consider a simple example illustrating these rules. Let $f(x, y) = 1.0/(x + y)$ and $x, y \in [0.5, 1.0]$. From Equation (17) we get $\mathrm{fp}(f)(x, y) = \mathrm{rnd}(1.0/\,\mathrm{rnd}(\mathrm{rnd}(x) + \mathrm{rnd}(y)))$. (Note that x and y are real variables so they must be rounded.) We take the rules CONST and $\mathsf{VAR}_{\mathsf{RND}}$ and apply them to corresponding subexpressions of $\mathrm{fp}(f)$:

$$\begin{aligned} \mathsf{CONST}(1.0) &= \langle 1.0, [] \rangle, \\ \mathsf{VAR}_{\mathsf{RND}}(\mathrm{rnd}(x)) &= \langle x, [f_{\mathrm{err}}(x)] \rangle = \langle x, [x] \rangle, \\ \mathsf{VAR}_{\mathsf{RND}}(\mathrm{rnd}(y)) &= \langle y, [f_{\mathrm{err}}(y)] \rangle = \langle y, [y] \rangle. \end{aligned}$$

Here, the function $f_{\mathrm{err}} : \mathbb{R} \to \mathbb{R}$ estimates the rounding error of a given value. We used the simplest definition of this function: $f_{\mathrm{err}}(c) = c$. But it is also possible to define f_{err} in a more precise way and get better error bounds for constants and variables. The rule $\mathsf{CONST}_{\mathsf{RND}}$ ($\mathsf{VAR}_{\mathsf{RND}}$, respecetively) may yield better results than application of rules CONST (VAR, respectively) and RND in sequence.

We present the remainder of the Taylor form construction across four steps, naming the intermediate results A through D. Applying the rule ADD to the Taylor forms of $\mathrm{rnd}(x)$ and $\mathrm{rnd}(y)$:

$$A = \mathsf{ADD}(\langle x, [x] \rangle, \langle y, [y] \rangle) = \langle x + y, [x] \mathbin{@} [y] \rangle.$$

We now apply the RND rule to A to get

$$B = \mathsf{RND}(A) = \left\langle x + y, [x + y] \ @ \ [x; y] \ @ \left[\epsilon M_2 + \frac{\delta}{\epsilon}\right]\right\rangle,$$

where $M_2 \geq \max_{x,y\in[0.5,1.0]}(|x| + |y|)$ is found to be 2 internally via interval arithmetic. For the sake of illustration, let us replace $\epsilon M_2 + \frac{\delta}{\epsilon}$ by the overapproximation 2.1ϵ:

$$B = \langle x + y, [x + y; x; y; 2.1\epsilon]\rangle .$$

Then we apply the rule INV to B (with some algebraic simplification):

$$C = \mathsf{INV}(B) = \left\langle \frac{1}{x+y}, \left[\frac{-1}{x+y}; \frac{-x}{(x+y)^2}; \frac{-y}{(x+y)^2}; \frac{-2.1\epsilon}{(x+y)^2}\right] \ @ \ [\epsilon M_2]\right\rangle,$$

where M_2 is computed using the formula in the INV rule. In this case, $M_2 = 16.1$, which is computed internally in FPTaylor using interval arithmetic. The next step is the multiplication rule applied to the Taylor form corresponding to the constant 1.0 and C: $\mathsf{MUL}(\langle 1.0, []\rangle, C) = \langle f_C, s_C \ @ \ [\epsilon \cdot 0]\rangle$. Finally, we apply the rule RND to $\mathsf{MUL}(\langle 1.0, []\rangle, C)$:

$$D = \left\langle \frac{1}{x+y}, \left[\frac{1}{x+y}\right] \ @ \left[\frac{-1}{x+y}; \frac{-x}{(x+y)^2}; \frac{-y}{(x+y)^2}; \frac{-2.1\epsilon}{(x+y)^2}; 16.1\epsilon; 0\right] \ @ \left[\epsilon M_2 + \frac{\delta}{\epsilon}\right]\right\rangle.$$

As before, we compute $\epsilon M_2 + \frac{\delta}{\epsilon} \leq 4.1$ and get the final Taylor form corresponding to our example (a complete treatment of all expression types is given under the proof of Theorem 5.1):

$$D = \left\langle \frac{1}{x+y}, \left[\frac{1}{x+y}; \frac{-1}{x+y}; \frac{-x}{(x+y)^2}; \frac{-y}{(x+y)^2}; \frac{-2.1\epsilon}{(x+y)^2}; 16.1\epsilon; 0; 4.1\epsilon\right]\right\rangle.$$

The main property of rules in Figure 1 is given by the following theorem.

Theorem 5.1. *Suppose* RULE *is one of the derivation rules in Figure 1 with k arguments and op is the corresponding mathematical operation. Let $F_1, \ldots, F_k$ be Taylor forms such that $h_1 \sim F_1, \ldots, h_k \sim F_k$ for some functions $h_1, \ldots, h_k$. Then we have*

$$op(h_1, \ldots, h_k) \sim \mathsf{RULE}(F_1, \ldots, F_k).$$

Proof. We prove the property in turn for all rules defined in Figure 1.

CONST. Let $c \in R$ be a constant. Then the corresponding Taylor form is $\langle c, []\rangle$. The proof of the fact that $c \sim \langle c, []\rangle$ is trivial. We have another rule for constants. If the symbolic expression of $\mathrm{fp}(f)$ contains the term $\mathrm{rnd}(c)$ (i.e., c cannot be exactly represented with a floating-point number), then the rule $\mathsf{CONST}_{\mathsf{RND}}$ is applied and the form $\langle c, [f_{\mathrm{err}}(c)]\rangle$ is derived. There are different ways to define the function $f_{\mathrm{err}}(c)$. The simplest definition is $f_{\mathrm{err}}(c) = c$. In this case, the fact $\mathrm{rnd}(c) \sim \langle c, [c]\rangle$ follows from Equation (1): $\mathrm{rnd}(c) = c(1 + e) = c + ce$ with $|e| \leq \epsilon$. (We need to make an additional assumption that $\mathrm{rnd}(c)$ is not in the subnormal range of floating-point numbers, i.e., $d = 0$ in Equation (1); it is usually the case, but if it is a subnormal number, then we still can construct a correct Taylor form as $\langle c, [\delta/\epsilon]\rangle$.) It is possible to construct a more precise Taylor form of c when $\mathrm{rnd}(c) \neq c$. We can always compute a precise value $f_{\mathrm{err}}(c) = (\mathrm{rnd}(c) - c)/\epsilon$ and the corresponding Taylor form.

VAR. The rules for variables are analogous to rules for constants.

RND. Given a Taylor form $\langle f, \mathrm{s}\rangle$, the rounding rule RND returns another Taylor form which corresponds to a rounding operator applied to the expression defined by $\langle f, \mathrm{s}\rangle$. We need to prove that $h \sim \langle f, \mathrm{s}\rangle$ implies $\mathrm{rnd}(h) \sim \mathsf{RND}(\langle f, \mathrm{s}\rangle)$ (here, $\mathrm{rnd}(h)$ is a function defined by $\mathrm{rnd}(h)(\mathbf{x}) = \mathrm{rnd}(h(\mathbf{x}))$). Fix $\mathbf{x}$. The assumption $h \sim \langle f, \mathrm{s}\rangle$ means that we can find $e_1, \ldots, e_k$ with $|e_i| \leq \epsilon$ such

that $h(\mathbf{x}) = f(\mathbf{x}) + \sum_{i=1}^{k} s_i(\mathbf{x})e_i$ (see Equation (22)). Equation (1) allows us to find e_{k+1} and d with $|e_{k+1}| \le \epsilon$, $|d| \le \delta$ such that

$$\begin{aligned}\mathrm{rnd}(h(\mathbf{x})) &= \left(f(\mathbf{x}) + \sum_{i=1}^{k} s_i(\mathbf{x})e_i\right)(1 + e_{k+1}) + d \\ &= f(\mathbf{x}) + \sum_{i=1}^{k} s_i(\mathbf{x})e_i + f(\mathbf{x})e_{k+1} + \left(d + e_{k+1}\sum_{i=1}^{k} s_i(\mathbf{x})e_i\right).\end{aligned}$$

Define $s_{k+1} = f$ and find M_2 such that $M_2 \ge \max_{\mathbf{x}\in I}(\sum_{i=1}^{k} |s_i(\mathbf{x})|)$. Define $s_{k+2} = \epsilon M_2 + \frac{\delta}{\epsilon}$. We get $d + e_{k+1}\sum_{i=1}^{k} s_i(\mathbf{x})e_i = s_{k+2}e_{k+2}$ for some e_{k+2}. Moreover, it is not difficult to see that $|e_{k+2}| \le \epsilon$. We can write

$$\exists e_1, \ldots, e_k, e_{k+1}, e_{k+2},\ |e_i| \le \epsilon \wedge \mathrm{rnd}(h(\mathbf{x})) = f(\mathbf{x}) + \sum_{i=1}^{k+2} s_i(\mathbf{x})e_i.$$

Compare the definitions of s_{k+1} and s_{k+2} with the result of the rule RND and conclude that $\mathrm{rnd}(h) \sim \mathrm{RND}(\langle f, \mathrm{s}\rangle)$.

SUB (ADD). Consider the subtraction rule (the addition rule is analogous). Suppose $h_1 \sim \langle f, \mathrm{s}\rangle$ and $h_2 \sim \langle g, \mathrm{t}\rangle$. Show that $h_1 - h_2 \sim \mathrm{SUB}(\langle f, \mathrm{s}\rangle, \langle g, \mathrm{t}\rangle)$. We can find $e_1, \ldots, e_k$ and $v_1, \ldots, v_r$, $|e_i| \le \epsilon$, $|v_j| \le \epsilon$, such that

$$\begin{aligned}h_1(\mathbf{x}) - h_2(\mathbf{x}) &= \left(f(\mathbf{x}) + \sum_{i=1}^{k} s_i(\mathbf{x})e_i\right) - \left(g(\mathbf{x}) + \sum_{j=1}^{r} t_j(\mathbf{x})v_j\right) \\ &= f(\mathbf{x}) - g(\mathbf{x}) + \left(\sum_{i=1}^{k} s_i(\mathbf{x})e_i + \sum_{j=1}^{r}(-t_j(\mathbf{x}))v_j\right).\end{aligned}$$

Hence the result follows.

MUL. Suppose that $h_1 \sim \langle f, \mathrm{s}\rangle$ and $h_2 \sim \langle g, \mathrm{t}\rangle$. Fix $\mathbf{x} \in I$, then by Equation (22) we have

$$\begin{aligned}h_1(\mathbf{x}) &= f(\mathbf{x}) + \sum_{i=1}^{t} s_i(\mathbf{x})e_i,\ \text{for some } e_1, \ldots, e_t, |e_i| \le \epsilon, \\ h_2(\mathbf{x}) &= g(\mathbf{x}) + \sum_{j=1}^{r} t_j(\mathbf{x})v_j,\ \text{for some } v_1, \ldots, v_r, |v_j| \le \epsilon.\end{aligned}$$

Compute the product of $h_1(\mathbf{x})$ and $h_2(\mathbf{x})$:

$$\begin{aligned}h_1(\mathbf{x})h_2(\mathbf{x}) &= \left(f(\mathbf{x}) + \sum_{i=1}^{t} s_i(\mathbf{x})e_i\right)\left(g(\mathbf{x}) + \sum_{j=1}^{r} t_j(\mathbf{x})v_j\right) \\ &= f(\mathbf{x})g(\mathbf{x}) + \sum_{j=1}^{r} f(\mathbf{x})t_j(\mathbf{x})v_j + \sum_{i=1}^{t} g(\mathbf{x})s_i(\mathbf{x})e_i + R_2(\mathbf{x}),\end{aligned}$$

where $R_2(\mathbf{x}) = \sum_{i=1,j=1}^{t,r} s_i(\mathbf{x})t_j(\mathbf{x})e_iv_j$. Find a constant M_2 such that $M_2 \ge \max_{\mathbf{x}\in I}(\sum_{i=1,j=1}^{t,r} |s_i(\mathbf{x})t_j(\mathbf{x})|)$. We have $M_2\epsilon^2 \ge |R_2(\mathbf{x})|$ for all $\mathbf{x} \in I$. Hence, for any $\mathbf{x}$ we can find $w = w(\mathbf{x})$, $|w| \le \epsilon$,

such that $R_2(\mathbf{x}) = \epsilon M_2 w$. Therefore,

$$h_1(\mathbf{x})h_2(\mathbf{x}) = f(\mathbf{x})g(\mathbf{x}) + \sum_{j=1}^{r} f(\mathbf{x})t_j + \sum_{i=1}^{t} g(\mathbf{x})s_i + (\epsilon M_2)w.$$

This equation holds for any $\mathbf{x} \in I$. Compare the right-hand side of this equation with the definition of the rule MUL and we get $h_1 h_2 \sim \mathsf{MUL}(\langle f, \mathrm{s}\rangle, \langle g, \mathrm{t}\rangle)$.

INV. The proof of this rule follows from the following Taylor expansion:

$$\frac{1}{f + \sum_k s_k e_k} = \frac{1}{f} - \sum_i \frac{s_i}{f^2} e_i + \sum_{i,j} \frac{s_i s_j}{(f + \sum_k s_k \theta_k)^3} e_i e_j,$$

where $|\theta_k| \le |e_k| \le \epsilon$. Replace the last sum in this expansion with its upper bound $M_2\epsilon$ and we get the rule INV.

SQRT. The proof of this rule follows from the following Taylor expansion:

$$\sqrt{f + \sum_k s_k e_k} = \sqrt{f} + \sum_i \frac{s_i}{2\sqrt{f}} e_i - \frac{1}{8} \sum_{i,j} \frac{s_i s_j}{(f + \sum_k s_k \theta_k)^{3/2}} e_i e_j,$$

where $|\theta_k| \le |e_k| \le \epsilon$. Replace the last sum in this expansion with its upper bound $M_2\epsilon$ and we get the rule SQRT.

SIN. The proof of this rule follows from the following Taylor expansion:

$$\sin\left(f + \sum_k s_k e_k\right) = \sin f + \sum_i s_i \cos(f) e_i - \frac{1}{2} \sum_{i,j} \sin\left(f + \sum_k s_k \theta_k\right) s_i s_j e_i e_j,$$

where $|\theta_k| \le |e_k| \le \epsilon$. Replace the last sum in this expansion with its upper bound $M_2\epsilon$ and we get the rule SIN. □

The next theorem summarizes the main result of this section.

Theorem 5.2. *For any input function* $\mathrm{fp}(f)$*, the Taylor form constructed with the rules described in Figure 1 corresponds to the function* $\mathrm{fp}(f)$*. That is, if the constructed Taylor form is* $\langle f, \mathrm{s}\rangle$*, then* $\mathrm{fp}(f) \sim \langle f, \mathrm{s}\rangle$ *and the property in Equation (22) holds.*

Proof. We present a sketch of the proof. The proof is by induction on the symbolic expression $\mathrm{fp}(f)$. The base case corresponds to Taylor forms of constants and variables which are derived with rules CONST and VAR. These rules produce correct Taylor forms (see Theorem 5.1). The induction step follows from the identity (here, we give a proof for the multiplication; all other operations are analogous): $\mathrm{fp}(f \times g) = \mathrm{rnd}(\mathrm{fp}(f) \times \mathrm{fp}(g))$. Suppose that $\mathrm{fp}(f) \sim \langle f, \mathrm{s}\rangle = F$ and $\mathrm{fp}(g) \sim \langle g, \mathrm{t}\rangle = G$. Theorem 5.1 implies $h = \mathrm{fp}(f) \times \mathrm{fp}(g) \sim \mathsf{MUL}(F, G) = H$ and $\mathrm{rnd}(h) \sim \mathsf{RND}(H)$. Therefore, $\mathrm{fp}(f \times g) \sim \mathsf{RND}(\mathsf{MUL}(F, G))$ and the result follows by induction. □

6 IMPLEMENTATION

We implemented a prototype tool called FPTaylor for estimating round-off errors in floating-point computations based on our method described in Sections 4 and 5. The tool implements all features described there such as estimation of relative errors (Section 4.2), support for transcendental functions (Section 5), mixed precision floating-point computations (Section 4.3), and the improved rounding model (Section 4.4).

FPTaylor is implemented in OCaml and uses several third-party tools and libraries. An interval arithmetic library (Alliot et al. 2012b) is used for rigorous estimations of floating-point constants and second-order error terms in Taylor expansions. Internally, FPTaylor implements a very simple

```
Variables
 float64 x in [1.001, 2.0],
 float64 y in [1.001, 2.0];
Definitions
 t rnd64= x * y;
// Constraints
//  x + y <= 2;
Expressions
 r rnd64= (t-1)/(t*t-1);
```

Fig. 2. FPTaylor input file example.

branch and bound global optimization technique based on interval arithmetic. The main advantage of this simple optimization method is that it can work even with discontinuous functions which are required by the improved rounding model described in Section 4.4. Our current implementation of the branch and bound method supports only simple interval constraints for input domain specification. FPTaylor also works with several external global optimization tools and libraries, such as NLopt optimization library (Johnson 2017) that implements various global optimization algorithms. The optimization algorithms in NLopt are not rigorous and may produce incorrect results, but they are fast and can be used for obtaining solid preliminary results before applying slower, rigorous optimization techniques. The Z3 SMT solver (de Moura and Bjørner 2008) can also be used as an optimization backend by employing a simple binary search algorithm similar to the one described in related work (Darulova and Kuncak 2014). We use Z3-based optimization to support inequality constraints; however, it does not work with transcendental or discontinuous functions. We also plan to support other free global optimization tools and libraries in FPTaylor such as ICOS (Lebbah 2009), GlobSol (Kearfott 2009), and OpenOpt (OpenOpt 2017). We optionally use the Maxima computer algebra system (Maxima 2013) for performing symbolic simplifications which can improve overall performance.

As input FPTaylor takes a text file describing floating-point computations, and prints out the computed floating-point error bounds as output. Figure 2 demonstrates an example FPTaylor input file. Each input file contains several sections which define variables, constraints (in Figure 2 constraints are not used and are commented out), and expressions. FPTaylor analyzes all expressions in an input file. All operations are assumed to be over real numbers. Floating-point arithmetic is modeled with rounding operators and with initial types of variables. The operator `rnd64=` in the example means that the rounding operator `rnd64` is applied to all operations, variables, and constants on the right-hand side (this notation is borrowed from Gappa (Daumas and Melquiond 2010)). See the FPTaylor user manual distributed with the tool for all usage details.

Implementation Details of Taylor Form Derivation. In Section 5, the definitions of Taylor forms and derivation rules are simplified. Taylor forms which we use in the implementation of our method keep track of error variables e_i explicitly in order to account for possible cancellations. Consider a simple example of computing a Taylor form of $\mathrm{fp}(f)$ where $f(x, y) = xy - xy$ with $x, y \in [0, 1] \cap \mathbb{F}$. It is obvious that $\mathrm{fp}(f)(x, y) = 0$ for all x and y. On the other hand, we have $\mathrm{fp}(f)(x, y) = \mathrm{rnd}(\mathrm{rnd}(xy) - \mathrm{rnd}(xy))$ and if we compute its Taylor form with rules from Figure 1, we get an error which is of order of magnitude of ϵ. The problem in this example is that the rounding error introduced by floating-point computation of xy should always be the same. Our simplified Taylor forms do not explicitly include error terms e_i, which we address with the following easy modification. Let a pair $\langle f, [s_i e_{a_i}]_i \rangle$ be a Taylor form where f, s_i are symbolic expressions and e_{a_i} are symbolic variables. Values of indices a_i can be the same for different values of i (e.g., we can have $a_3 = a_1 = 1$). With this new definition of the Taylor form, the only significant

change must be done in the rounding rule RND. This rule creates the following list of error terms: $[f]$@ s @$[\epsilon M_2 + \frac{\delta}{\epsilon}]$. This list needs to be replaced with the list $[f e_{a_f}]$@ s @$[(\epsilon M_2 + \frac{\delta}{\epsilon}) e_a]$. Here, e_a is a fresh symbolic variable and the index a_f corresponds to the symbolic expression f; a_f should be the same whenever the same expression is rounded.

Explicit error terms also provide the mixed-precision support in FPTaylor. It is done by attaching different bounds (values of ϵ and δ) to different error terms.

Improvements to Taylor Form Derivation. We implemented several other improvements of the derivation rules for obtaining better error bounds:

- Whenever we multiply an expression by a non-negative power of 2 or divide by a negative power of 2, we do not need to round the result.
- If we divide by a non-negative power of 2 or multiply by a negative power of 2, we only need to consider potential subnormal errors (given by the term $\frac{\delta}{\epsilon}$ in the RND rule).
- There are no subnormal errors for rounding after addition, subtraction, or square root (i.e., we do not need to add the term $\frac{\delta}{\epsilon}$ in the RND rule).
- Constants which can be exactly represented by floating-point numbers (of a given precision) are not rounded.
- The rule $\mathsf{CONST}_{\mathsf{RND}}$ has different implementations for standard (Equation (1)) and improved (Equation (16)) rounding models. For the improved rounding model, we have $f_{\mathrm{err}}(c) = (\mathrm{rnd}(c) - c)/\epsilon$ computed with infinite precision rational arithmetic (all input constants are finite decimal numbers in FPTaylor). For the standard rounding model, we have $f_{\mathrm{err}}(c) = p_2(c)$ (see Section 4.4). This definition is formalized in HOL Light. The f_{err} definition for the improved rounding model is better and we plan to use it for the standard rounding model when we formalize it in HOL Light.
- The rule $\mathsf{VAR}_{\mathsf{RND}}$ has two different implementations for the standard rounding model. The default implementation is $f_{\mathrm{err}}(x) = x$. Another implementation is $f_{\mathrm{err}}(x) = \max\{|p_2(a)|, |p_2(b)|\}$ if the variable x belongs to the interval $[a, b]$. This implementation may yield better results when the interval $[a, b]$ is not too wide. Essentially, we overestimate the improved rounding error for a variable. From the optimization point of view, the second implementation is simpler than the default implementation because it replaces some variables with constants. In our experiments, we observed that the second implementation is better for all but one benchmark (t_div_t1; see Section 7) and we will present results for the second implementation only. Whenever the standard rounding model is used, one can always run experiments with both variable rounding rules and select the best results. Note that this rule is only applied to real-valued input variables.

6.1 Rigorous Global Optimizer

We have developed a global optimization tool called Gelpia (Gelpia 2017) to obtain the upper-bounds of round-off errors. In general, finding the maximum value of an n-variable function requires search over the n-dimensional space of its inputs—an n-dimensional rectangle. Given the large number of floating-point n-tuples in such input space, exhaustive search is all but impossible, and sampling can cover only a tiny fraction of possible input tuples. Precision estimation and optimization methods leverage a variety of tools and techniques, including dReal (Gao et al. 2013), semi-definite programming (Magron et al. 2017), SMT (Darulova and Kuncak 2014), and classical tools for interval and affine arithmetic (Daumas and Melquiond 2010; Darulova and Kuncak 2011, 2014). Previous studies (Panchekha et al. 2015; Lee et al. 2016; Solovyev et al. 2015) have shown that using optimization tools in this arena is promising, often proving to be advantageous

```
function IBBA(f, x, x_tol, f_tol)
  f_bestLow ← −∞
  f_bestHigh ← −∞
  Q ← Queue()
  Q.push(x)
  while Q ≠ ∅ do
    x_n ← Q.pop()
    fx_n ← f(x_n)
    f_bestLow ← max(f_bestLow, lower(fx_n))
    if upper(fx_n) < f_bestLow or width(x_n) < x_tol or
       width(fx_n) < f_tol then
      f_bestHigh ← max(f_bestHigh, upper(fx_n))
      continue
    end if
    x_l, x_r ← split(x_n)
    Q.push(x_l)
    Q.push(x_r)
  end while
  return f_bestHigh
end function
```

Fig. 3. Interval branch and bound algorithm (IBBA) underlying Gelpia. Here, f is the function to optimize and x is the input domain (treated as a scalar here, but in general, is an N-dimensional rectangular domain). Arguments x_{tol} and f_{tol} are scalars used to suppress the *split* step when either the input or the output interval width are small.

to more classical (e.g., SMT-based) methods that do not support important classes of functions (e.g., transcendental functions). When the interval functions in question are *monotonic* (for rectangle r_1 contained in rectangle r_2, i.e., $r_1 \sqsubseteq r_2$, the upper-bound calculation respects $f(r_1) \sqsubseteq f(r_2)$), one can perform this search using a combination of heuristics.

The basic heuristics are to split a rectangle along the longest dimension, obtain upper-bounds for each sub-rectangle, and zoom into the most promising sub-rectangle, while also keeping alive a population of postponed rectangles (Alliot et al. 2012a). We have found that performing this split at the point where the exponent increments improves performance. Determining if a sub-rectangle is promising utilizes the derivative of the function—if the derivative spans zero, then a local maxima or minima must be present. A multi-point function estimation is used to further order the intervals. This of course only changes the convergence rate, and not the soundness of the optimizer. This is known as the *interval branch and bound* algorithm, and Figure 3 gives the pseudocode as it is implemented in Gelpia. The arguments x_{tol} and f_{tol} dictate, respectively, the maximum width an input rectangle and output interval are allowed to have before bounding occurs. This allows for full optimization when these arguments are set to zero. We implemented this basic algorithm with a number of improvements in our Gelpia global optimizer.

Gelpia is a *rigorous* global optimizer—it guarantees that the returned upper bound is greater than or equal to the global maximum, and the returned lower bound is less than or equal to the global maximum. Key to its efficiency is its use of GAOL (Goualard 2017), an interval library which uses X86 SIMD instructions to speed up interval arithmetic, and also supports transcendental functions such as sin, cos, tan. GAOL is sound as it satisfies the *inclusion* property for interval arithmetic. For example, if $[a, b] + [c, d] = [a + c, b + d]$, where the addition is computed using real arithmetic, GAOL computes the interval as $[a, b] \bigoplus [c, d] = [\underline{a + c}, \overline{b + d}]$, where $\underline{a + c}$ is the nearest double rounded toward $-\infty$ and $\overline{b + d}$ is the nearest double rounded toward ∞. This guarantees

the interval inclusion property as $[a, b] + [c, d] = [a + c, b + d] \subseteq [\underline{a + c}, \overline{b + d}] = [a, b] \bigoplus [c, d]$. Since we are operating on real intervals, we employ rewriting to improve results. For example, if $x = [-1, 1]$ then, $x \cdot x$ equals $[-1, 1]$ in interval arithmetic; we replace the input sub-expression with x^2 which evaluates to $[0, 1]$. We are also able to determine statically if a division-by-zero error occurs, and emit an appropriate message to the user. We implemented Gelpia as an open source project using the Rust programming language, and we parallelized the search algorithm. We use an *update* thread that periodically synchronizes all solvers to focus their attention on the current most promising sub-rectangle. Additionally, information from other solvers is used to boost the priorities of promising sub-rectangles.

6.2 Formal Verification of FPTaylor Results in HOL Light

We formalized error estimations in Equations (10) and (11) in HOL Light (Harrison 2009). In our formalization we do not prove that the implementation of FPTaylor satisfies a given specification. Instead, we formalized theorems necessary for validating results produced by FPTaylor. The validity of results is checked against specifications of floating-point rounding operations given by Equation (1). We also use Equation (16) to certify error bounds of constants. The general improved rounding model is not fomally verified yet. We chose HOL Light as the tool for our formalization because it includes a procedure for formal verification of nonlinear inequalities (including inequalities with transcendental functions) (Solovyev and Hales 2013). Verification of nonlinear inequalities is necessary since the validity of results of global optimization procedures can be proved with nonlinear inequalities. Proof assistants PVS (Narkawicz and Munoz 2013) and Coq (Martin-Dorel et al. 2013) also include procedures for verification of nonlinear inequalities.

The validation of FPTaylor results is done as follows. First, FPTaylor is executed on a given problem with a special proof saving flag turned on. In this way, FPTaylor computes the round-off errors and produces a proof certificate and saves it in a file. Then a special procedure is executed in HOL Light which reads the produced proof certificate and formally verifies that all steps in this certificate are correct. The final theorem has the following form (for an error bound e computed by FPTaylor):

$$\vdash \forall \mathbf{x} \in I,\ |\,\mathrm{fp}(f)(\mathbf{x}) - f(\mathbf{x})| \leq e.$$

Here, the function $\mathrm{fp}(f)$ is a function where a rounding operator is applied to all operations, variables, and constants. As mentioned above, in our current formalization we define such a rounding operator as any operator satisfying Equations (1) and (16). We also implemented a comprehensive formalization of floating-point arithmetic in HOL Light (Jacobsen et al. 2015); our floating-point formalization is available in the HOL Light distribution. Combining this formalization with theorems produced from FPTaylor certificates, we can get theorems about floating-point computations which do not explicitly contain references to rounding models from Equations (1) and (16).

The formalization of FPTaylor helped us to find a critical bug in our implementation. We have an option to use an external tool for algebraic simplifications of internal expressions in FPTaylor (see Section 6 for more details). All expressions are passed as strings to this tool. Constants in FPTaylor are represented with rational numbers and they are printed as fractions. We forgot to put parentheses around these fractions and in some rare cases it resulted in wrong expressions passed to and from the simplification tool. For instance, if $c = 111/100$ and we had the expression $1/c$, then it would be given to the simplification tool as $1/111/100$. We discovered this associativity-related bug when formal validation failed on one of our test examples. The main takeaway is that the maintenance and enhancement of FPTaylor (and other tools that generate proof-certificates) is greatly facilitated by such safety-nets.

All limitations of our current formalization are limitations of the tool for verification of nonlinear inequalities in HOL Light. In order to get a verification of all features of FPTaylor, it is necessary to be able to verify nonlinear inequalities containing the discontinuous function $p_2(x)$ defined in Section 4.4. We are working on improvements of the inequality verification tool which will include this function. Nevertheless, we already can automatically verify interesting results which are much better than results produced by Gappa, another tool which can produce formal proofs in the Coq proof assistant (Coq 2016).

6.3 Handling of Conditionals

FPTaylor is not a tool for general-purpose floating-point program analysis. It cannot handle conditionals and loops directly, but can be used as an external decision procedure for program verification tools (e.g., Frama-C (2017) and Rakamarić and Emmi (2014)).

Conditional expressions can be verified in FPTaylor in the same way as it is done in Rosa (Darulova and Kuncak 2014). Consider a simple real-valued expression

$$f(x) = \text{if } c(x) < 0 \text{ then } f_1(x) \text{ else } f_2(x).$$

The corresponding floating-point expression is the following:

$$\tilde{f}(x) = \text{if } \tilde{c}(x) < 0 \text{ then } \tilde{f}_1(x) \text{ else } \tilde{f}_2(x),$$

where $\tilde{c}(x) = c(x) + e_c(x)$, $\tilde{f}_1(x) = f_1(x) + e_1(x)$, and $\tilde{f}_2(x) = f_2(x) + e_2(x)$. Our goal is to compute a bound E of the error $e(x) = \tilde{f}(x) - f(x)$.

First of all, we estimate the error $e_c(x)$. Suppose it is bounded by a constant E_c: $|e_c(x)| < E_c$. Now we need to consider four cases: two cases when both $f(x)$ and $\tilde{f}(x)$ take the same path, and two cases when they take different paths:

(1) Find E_1 such that $c(x) < 0 \implies |\tilde{f}_1(x) - f_1(x)| \le E_1$.
(2) Find E_2 such that $c(x) \ge 0 \implies |\tilde{f}_2(x) - f_2(x)| \le E_2$.
(3) Find E_3 such that $-E_c < c(x) < 0 \implies |\tilde{f}_2(x) - f_1(x)| \le E_3$.
(4) Find E_4 such that $0 \le c(x) < E_c \implies |\tilde{f}_1(x) - f_2(x)| \le E_4$.

Finally, we take $E = \max\{E_1, E_2, E_3, E_4\}$. Problems (1)–(4) can be solved in FPTaylor. Indeed, FPTaylor can handle additional constraints given in these problems ($c(x) < 0$, and so forth; currently, constraints are supported by the Z3-based optimization procedure only) and it can directly compute bounds of errors $|\tilde{f}_i(x) - f_i(x)|$, $i = 1, 2$. The value of E_3 can be determined from the following inequality:

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

It is enough to bound the range of the real-valued function $f_2(x) - f_1(x)$ which FPTaylor can do. We can find E_4 in the same way.

The procedure described above is partially implemented in FPTaylor and we already can handle some examples with conditionals in a semi-automatic way (we need to prepare separate input files for each case described above).

Consider a simple example which demonstrates that automatic handling of conditionals in FPTaylor is a promising research direction. Figure 4 presents a simple Fluctuat (Delmas et al. 2009) example with two floating-point variables a and b such that $a, b \in [0, 100]$. We want to measure the round-off error in the result r. We prepared corresponding input files for Rosa and FPTaylor. Table 2 shows results obtained with Fluctuat (version 3.1071), Rosa (version from May 2014), and FPTaylor on this simple example. We can see that Fluctuat (even with manual subdivisions) failed

```
int main(void) {
  double a, b, r;
  a = __BUILTIN_DAED_DBETWEEN(0.0, 100.0);
  b = __BUILTIN_DAED_DBETWEEN(0.0, 100.0);
  if (b >= a) {
    r = b / (b - a + 0.5);
  } else {
    r = b / 0.5;
  }
  DSENSITIVITY(r);
  return 0;
}
```

Fig. 4. A simple Fluctuat example with a conditional expression.

Table 2. Round-Off Error Estimation Results for the Example in Figure 4

Fluctuat	Fluctuat (subdiv.)	Rosa	FPTaylor
∞	∞	1.8e-11	5.8e-12

to find any error bound in this example. Results of FPTaylor are about three times better than Rosa's results.

7 EXPERIMENTAL RESULTS

In this section, we describe our extensive empirical evaluation of FPTaylor, including a detailed comparison of various tools and configurations in terms of the computed round-off errors and performance.

7.1 Tools and Benchmarks

We have conducted a detailed comparative study of FPTaylor against Gappa (version 1.3.1) (Daumas and Melquiond 2010), Fluctuat (version 3.1384) (Delmas et al. 2009), PRECiSA (Titolo 2017), Real2Float (version 0.7), and the Rosa compiler for reals (we report results from the version of this tool from the *opt* branch dated October 2015, as this yields much better results overall) (Darulova and Kuncak 2014). See Section 8 for more information on these tools. Table 3 lists FPTaylor configurations we experimented with, where "Simple bb" denotes a simple implementation of the branch and bound algorithm used in the FPTaylor paper (Solovyev et al. 2015). As an example, *FPTaylor-f* computes results using the standard rounding approach (that supports HOL Light proof generation), decomposed optimization problem (Equation (11)), and Gelpia optimizer. On the other hand, *FPTaylor-h* computes results using the improved rounding model (for which our HOL Light proof generation is not available) combined with the monolithic optimization problem (Equations (16) and (10)). Table 4 lists the configurations of related tools we experimented with. *Gappa (hints)* and *Fluctuat (subdiv.)* compute results using manually provided subdivision hints. More precisely, Gappa and Fluctuat are instructed to subdivide intervals of input variables into a given number of smaller pieces. The main drawback of these manually provided hints is that it is not always clear which variable intervals should be subdivided and how many pieces are required. It is very easy to make Gappa and Fluctuat very slow by subdividing intervals into too many pieces (even 100 pieces are enough in some cases).[5] Note that we selected the results of only the most

[5]In one experiment, we provided Gappa with hints of the form $x; $y; and that provided results in-between in quality between Gappa and Gappa (hints).

Table 3. Configurations of FPTaylor Used in the Experiments

Configuration	Rounding model	Optimizer	Optimization type
FPTaylor_a	standard	Simple bb	decomposed
FPTaylor_b	standard	Simple bb	monolithic
FPTaylor_c	improved	Simple bb	decomposed
FPTaylor_d	improved	Simple bb	monolithic
FPTaylor_e	standard	Gelpia	decomposed
FPTaylor_f	standard	Gelpia	monolithic
FPTaylor_g	improved	Gelpia	decomposed
FPTaylor_h	improved	Gelpia	monolithic
FPTaylor_i	standard	Z3	decomposed
FPTaylor_j	standard	Z3	monolithic

Table 4. Configurations of Related Tools Used in the Experiments

Tool	Version and notes
Fluctuat	fluctuat_v3.1384
Fluctuat subdiv.	fluctuat_v3.1384 with 20 subdivisions per input variable
Gappa	Gappa 1.3.1
Gappa simple	Gappa 1.3.1 with simple hints
Gappa hints	Gappa 1.3.1 with advanced hints
PRECiSA	master branch (commit 19f5089ca9ef) with args. 52 5 2 False 40
Real2Float	version 0.7
Rosa	master branch (commit 1f9e9d2fc1dc)
Rosa opt	opt branch (commit d1360b85a563)

interesting configurations to be included in the article. We provide all results in the accompanying spreadsheet at Solovyev (2017).

Table 5 lists all of our benchmarks and provides a comparison with the dynamic underapproximation tool S3FP (Chiang et al. 2014). We used 24 benchmarks in our empirical evaluation, most of which were obtained from related work (Darulova and Kuncak 2014; Magron et al. 2017); we added several more benchmarks that include transcendental functions.[6] For all benchmarks, input values are assumed to be real numbers, which is how Rosa treats input values, and hence we always need to consider uncertainties in inputs. Benchmarks *sine*, *sqroot*, and *sineOrder3* are different polynomial approximations of sine and square root. The benchmark *t_div_t1* is the $t/(t+1)$ example presented in Section 1. Benchmarks *carbonGas*, *rigidBody1*, *rigidBody2*, *doppler1*, *doppler2*, and *doppler3* are nonlinear expressions used in physics. Benchmarks *verhulst* and *predPrey* are from biological modeling. Benchmarks *turbine1*, *turbine2*, *turbine3*, and *jetEngine* are from control theory. Benchmarks *kepler0*, *kepler1*, *kepler2*, *himmilbeau*, and *hartman3* are from mathematical problems. Benchmark *logExp* is from the Gappa++ paper (Linderman et al. 2010) and it estimates the error in $log(1 + exp(x))$ for $x \in [-8, 8]$. Benchmarks *sphere* and *azimuth* are taken from NASA World Wind Java SDK (NASA 2017), which is a popular open-source 3D interactive world viewer with many users ranging from US Army and Air Force to European Space Agency. An example application that leverages World Wind is a critical component of the

[6]Our benchmarks are available at https://github.com/soarlab/FPTaylor/tree/develop/benchmarks/toplas.

Table 5. Benchmarks, Their Characteristics, and Comparison with the Dynamic Underapproximation Tool S3FP

Benchmark	Vars	Ops	Trans	FPTaylor-h	S3FP	Ratio
Univariate polynomial approximations						
sine	1	18	0	4.5e-16	2.9e-16	1.51
sineOrder3	1	5	0	6.0e-16	4.1e-16	1.45
sqroot	1	14	0	5.1e-16	4.7e-16	1.08
Rational functions with 1 to 6 variables						
t_div_t1	1	2	0	2.3e-16	1.6e-16	1.38
carbonGas	1	11	0	6.0e-9	4.2e-9	1.40
doppler1	3	8	0	1.3e-13	1.0e-13	1.24
doppler2	3	8	0	2.3e-13	1.9e-13	1.20
doppler3	3	8	0	6.7e-14	5.7e-14	1.16
himmilbeau	2	14	0	1.1e-12	7.5e-13	1.45
jetEngine	2	48	0	1.1e-11	7.1e-12	1.53
kepler0	6	15	0	7.5e-14	5.3e-14	1.40
kepler1	4	24	0	2.9e-13	1.6e-13	1.72
kepler2	6	36	0	1.6e-12	8.4e-13	1.90
predPrey	1	7	0	1.6e-16	1.5e-16	1.04
rigidBody1	3	7	0	3.0e-13	2.7e-13	1.09
rigidBody2	3	14	0	3.7e-11	3.0e-11	1.21
turbine1	3	14	0	1.7e-14	1.1e-14	1.45
turbine2	3	10	0	2.0e-14	1.4e-14	1.39
turbine3	3	14	0	9.6e-15	6.2e-15	1.54
verhulst	1	4	0	2.5e-16	2.4e-16	1.02
Transcendental functions with 1 to 4 variables						
azimuth	4	14	7	8.9e-15	6.6e-15	1.35
hartman3	3	72	4	4.6e-15	2.4e-15	1.89
logexp	1	3	2	2.0e-15	1.4e-15	1.43
sphere	4	5	2	8.4e-15	6.4e-15	1.29

Columns *Vars*, *Ops*, and *Trans* show the numbers of variables, floating-point operations, and transcendental operations in each benchmark, respectively; column *FPTaylor-h* shows error bounds computed by FPTaylor; column *S3FP* shows lower bounds of errors estimated with S3FP; column *Ratio* gives ratios of overapproximations computed with FPTaylor-h and underapproximations computed with S3FP.

Next Generation Air Transportation System (NextGen) called AutoResolver, whose task is to provide separation assurance for airplanes (Giannakopoulou et al. 2014).

We first compare the results of FPTaylor with lower bounds of errors estimated with a state-of-the-art dynamic underapproximation tool S3FP (Chiang et al. 2014) in Table 5. All FPTaylor results are only 1.0–1.9 times worse than the estimated lower bounds for polynomial and rational benchmarks and 1.3–1.9 times worse for transcendental tests. This indicates that the overapproximations of round-off errors produced by FPTaylor are typically close to the actual errors. We also conducted experiments in which SMT solvers that support floating-point reasoning (Cimatti et al. 2013; de Moura and Bjørner 2008) were directly used to measure roundoff error. This was done by expressing the function of interest in the theory of real numbers as well as in the theory of floating-point numbers, and then showing that their difference lies within a threshold. This approach was unable to produce results even on simple examples in a reasonable amount of time (we

set timeout to 30 minutes). We performed all experiments on an Intel Xeon E5-2680 machine with 126GB of RAM.

7.2 Comparison of Computed Round-Off Errors

Table 6 gives the main results of our experimental comparison. All results are given for double precision floating-point numbers and we ran Gappa, Fluctuat, Real2Float, and Rosa with standard settings. We now detail aspects specific to various tools and discuss the results they produce.

For PRECiSA, a cursory pass was made in which we varied the required configuration variables over multiple runs. Based on this "training" phase, a final configuration to run PRECiSA was chosen based on the best average answer for the benchmarks while not exceeding maximum runtime of the existing tools. Times for PRECiSA are based on the tool-reported runtime. We should observe that this tool-reported runtime actually excludes a setup time that the tool incurs when run in a batch mode. The authors of PRECiSA (correctly) argue that that this setup time will be incurred only once when using the tool interactively, and so it makes sense to ignore it.

We used a simple branch and bound optimization method in FPTaylor since it works better than Z3-based optimization on most benchmarks. For transcendental functions, we employ increased values of ϵ and δ with the 1.5 coefficient: $\epsilon = 1.5 \cdot 2^{-53}$ and $\delta = 1.5 \cdot 2^{-1075}$. It is well known that error models for transcendental functions are implementation dependent. The standard recommends (but does not require) to have correctly rounded basic transcendental functions. This can be achieved with CRLibm or analogous libraries. Java documentation requires 1 ulp error for transcendental functions, which is equivalent to the coefficient of 2.0 in FPTaylor. However, this bound appears to be too pessimistic: we ran several simple experiments and never observed a round-off error larger than 0.6 ulp for exp, sin, cos, and log (for double-precision arguments). There are some formal results for particular implementations. For example, Harrison gives the bound of 0.57341 ulp for cos on IA-64 (Harrison 2000). We chose the coefficient of 1.5 as a compromise between actual implementations (the coefficient of about $0.6/0.5 = 1.2$) and a pessimistic safe choice (the coefficient of 2.0).

Gappa computed best results in 3 out of 20 benchmarks (we do not count the last 4 benchmarks with transcendental functions). FPTaylor computed best results in 23 benchmarks, except on the *himmilbeau* benchmar.[7] Gappa without hints was able to find a result better than or equivalent to FPTaylor-h with respect to the *himmilbeau*, *rigidBody1*, and *rigidBody2* benchmarks. On the other hand, in several benchmarks (*t_div_t1*, *jetEngine*, and *turbine3*), Gappa (even with hints) computed very pessimistic results.

Real2Float typically produces error bounds that are more pessimistic than Rosa's. The tool also ran out of memory on the *jetEngine* benchmark, which is consistent with the findings in the paper (Magron et al. 2017). At the same time, Real2Float found good error bounds for all transcendental benchmarks. It also was able to find the second best bound for *t_div_t1*.

Rosa consistently computed decent error bounds, with exceptions for *t_div_t1* and *jetEngine*. We used the opt branch of Rosa due to inproved runtimes and improved answers with one exception being *sine* which has a slightly better answer using the master branch with a computed answer of 5.19e-16 versus the opt branch computing 5.74e-16. FPTaylor-h outperformed Rosa on all benchmarks, while the results with the standard rounding model FPTaylor-f are slightly more mixed (Rosa produces tighter bounds in that case for *sqroot*, *himmilbeau*, *kepler0*, *rigidBody1*, and *rigidBody2*).

[7]While the absolute error changing from (e.g.) 10^{-8} to 10^{-10} does not appear to be large, it is a significant two-order of magnitude difference; for instance, imagine these differences accumulating over 10^4 iterations in a loop.

Table 6. Experimental Results for Absolute Round-Off Error Bounds

Benchmark	Fluctuat	Fluctuat (subdiv.)	Gappa	Gappa (hints)	PRECiSA	R2Float	Rosa (opt)	FPTaylor-f	FPTaylor-h
sine	8.0e-16	7.5e-16	1.2e-15	7.0e-16	6.0e-16	6.1e-16	5.8e-16	5.6e-16	**4.5e-16**
	0.25	0.11	0.08	25.43	11.76	4.95	5.20	1.20	1.14
sineOrder3	1.2e-15	1.1e-15	8.9e-16	6.6e-16	1.2e-15	1.2e-15	1.0e-15	9.6e-16	**6.0e-16**
	0.36	0.09	0.05	2.09	6.11	2.22	3.82	0.98	1.02
sqroot	6.9e-16	6.9e-16	5.8e-16	5.4e-16	6.9e-16	1.3e-15	6.2e-16	7.0e-16	**5.1e-16**
	0.39	0.09	0.11	5.06	8.18	4.23	4.20	1.05	1.02
t_div_t1	*1.2e-10*	*2.8e-12*	*1,000*	*10*	*2.3e-13*	5.5e-16	*5.7e-11*	5.8e-14	**2.3e-16**
	0.31	0.09	0.01	0.18	5.89	125.31	6.55	1.00	0.95
carbonGas	4.6e-8	1.2e-8	2.7e-8	6.1e-9	7.4e-9	2.3e-8	1.6e-8	9.2e-9	**6.0e-9**
	1.55	0.54	0.11	2.35	6.30	4.65	6.03	1.10	1.08
doppler1	4.0e-13	**1.3e-13**	2.1e-13	1.7e-13	2.7e-13	7.7e-12	2.5e-13	1.6e-13	**1.3e-13**
	0.52	6.30	0.05	3.31	16.17	13.20	10.86	2.20	1.97
doppler2	9.8e-13	2.4e-13	4.0e-13	2.9e-13	5.4e-13	1.6e-11	5.7e-13	2.9e-13	**2.3e-13**
	0.53	6.15	0.05	3.37	16.87	13.33	10.58	2.42	2.20
doppler3	1.6e-13	7.2e-14	1.1e-13	8.7e-14	1.4e-13	8.6e-12	1.1e-13	8.3e-14	**6.7e-14**
	0.80	6.46	0.05	3.32	15.65	13.05	9.41	2.15	1.88
himmilbeau	1.1e-12	1.1e-12	1.1e-12	**8.6e-13**	1.1e-12	1.5e-12	1.1e-12	1.4e-12	1.1e-12
	0.08	0.34	0.07	1.86	26.20	0.66	4.73	1.03	1.02
jetEngine	*4.1e-8*	1.1e-10	*8,300,000*	*4,500*	crash	OoM	5.0e-9	1.4e-11	**1.1e-11**
	0.62	1.45	0.20	27.64	N/A	N/A	84.01	4.89	2.84
kepler0	1.2e-13	1.1e-13	1.3e-13	1.1e-13	1.2e-13	1.2e-13	8.3e-14	9.5e-14	**7.5e-14**
	0.12	8.59	0.13	7.33	37.57	0.76	5.70	1.13	1.31
kepler1	5.2e-13	3.6e-13	5.4e-13	4.7e-13	crash	4.7e-13	3.9e-13	3.6e-13	**2.9e-13**
	0.10	157.74	0.23	10.68	N/A	22.53	18.82	1.23	2.08
kepler2	2.7e-12	2.3e-12	2.9e-12	2.4e-12	crash	2.1e-12	2.1e-12	2.0e-12	**1.6e-12**
	0.12	22.41	0.44	24.17	N/A	16.53	19.67	1.87	1.30
predPrey	2.5e-16	2.4e-16	2.1e-16	1.7e-16	1.7e-16	2.6e-16	2.0e-16	1.9e-16	**1.6e-16**
	0.50	0.18	0.04	1.40	8.08	4	11.20	1.11	1.07
rigidBody1	3.3e-13	3.3e-13	**3.0e-13**	**3.0e-13**	**3.0e-13**	5.4e-13	3.3e-13	3.9e-13	**3.0e-13**
	0.40	1.96	0.06	1.42	7.42	3.09	3.48	1.01	0.99
rigidBody2	**3.7e-11**	**3.7e-11**	**3.7e-11**	**3.7e-11**	**3.7e-11**	6.5e-11	**3.7e-11**	5.3e-11	**3.7e-11**
	0.22	3.87	0.09	2.22	10.79	1.08	4.20	1.04	1.02
turbine1	9.3e-14	3.1e-14	8.4e-14	2.5e-14	3.8e-14	*2.5e-11*	6.0e-14	2.4e-14	**1.7e-14**
	0.55	5.05	0.11	5.54	24.35	136.35	9.10	1.15	1.10
turbine2	1.3e-13	2.6e-14	1.3e-13	3.4e-14	3.1e-14	2.1e-12	7.7e-14	2.6e-14	**2.0e-14**
	0.79	3.98	0.08	3.94	19.17	8.30	5.46	1.09	1.17
turbine3	7.0e-14	1.4e-14	*40*	*0.36*	2.3e-14	*1.8e-11*	4.7e-14	1.3e-14	**9.6e-15**
	0.62	5.08	0.11	6.29	24.47	137.36	7.60	1.14	1.21
verhulst	5.6e-16	4.9e-16	4.2e-16	2.9e-16	2.9e-16	4.7e-16	4.7e-16	3.3e-16	**2.5e-16**
	0.26	0.09	0.02	0.41	4.95	2.52	7.15	1.01	0.99
azimuth	–	–	–	–	crash	2.9e-13	–	1.2e-14	**8.9e-15**
	–	–	–	–	N/A	2.30	–	5.22	4.62

(Continued)

Table 6. Continued

Benchmark	Fluctuat	Fluctuat (subdiv.)	Gappa	Gappa (hints)	PRECiSA	R2Float	Rosa (opt)	FPTaylor-f	FPTaylor-h
hartman3	–	–	–	–	–	3.0e-13	–	7.0e-15	**4.6e-15**
	–	–	–	–	–	3.51	–	12.98	12.67
logexp	–	–	–	–	–	2.6e-15	–	2.1e-15	**2.0e-15**
	–	–	–	–	–	0.80	–	0.98	0.96
sphere	–	–	–	–	9.0e-14	1.6e-14	–	1.1e-14	**8.4e-15**
	–	–	–	–	20.58	0.08	–	5.10	5.37

Grey rows show computed round-off errors; white rows show runtimes in seconds; bold font marks best results for each benchmark; italic font marks pessimistic results at least three orders of magnitude worse than the best ones; *OoM* marks the tool running out of memory.

Fluctuat results without subdivisions are worse than Rosa and FPTaylor's results. Fluctuat results with subdivisions are comparable to Rosa's, but they were obtained with carefully chosen subdivisions. It found better results than FPTaylor-f for *sqroot*, *doppler1*, *doppler2*, *doppler3*, *himmilbeau*, *rigidBody1*, and *rigidBody2*. FPTaylor with the improved rounding model outperformed Fluctuat with subdivisions on all benchmarks. Only FPTaylor and Fluctuat with subdivisions found good error bounds for the *jetEngine* benchmark. We find that PRECiSA performs about as well as Rosa on all benchmarks, but takes more time than Rosa.

FPTaylor yields best results with the improved rounding model (Equation (16)) combined with the monolithic optimization problem (Equation (10)). These results are at most 1.6 times better (with an exception for *t_div_t1*) than results computed with the standard rounding model (Equation (3)) combined with the decomposed optimization problem (Equation (11)). The main advantage of the decomposed optimization problem is that it creates multiple simple queries for the underlying optimizer. Hence, it can be applied to more complex problems and used with almost any global optimizer. On the other hand, the monolithic optimization problem uses only one advanced query, which typically yields better precision results. However, the advanced query is not twice differentiable, and often includes discontinuities, both of which violate requirements for many global optimization algorithms.

7.3 Performance and Formal Verification Results

Table 7 compares performance results of different tools on the 17 benchmarks which all tools were able to complete (FPTaylor takes about 24 seconds on four transcendental benchmarks using monolithic optimization). They only provide a rough idea of what to expect performance-wise with these tools, given that performance is largely a function of the component technologies (e.g., external optimizers) that all tools end up using. For example, the performance of FPTaylor using the monolithic optimization problem (FPTaylor-f, FPTaylor-h) is better than using the decomposed problem (FPTaylor-e, FPTaylor-g), which seems counterintuitive. This is because Gelpia is invoked multiple times when the decomposed problem is used, leading to a startup time overhead that on short-running benchmarks overshadows the benefits of having simpler optimization queries.

In our evaluation of PRECiSA, we found that general configurations often yielded either inferior estimates or increased time relative to FPTaylor. The manner in which configurations are to be selected can benefit from more guidelines, as it can dramatically affect result quality as well as execution times. Gappa and Fluctuat (without hints and subdivisions) are considerably faster than Real2Float, Rosa, and FPTaylor. Gappa often fails to produce tight bounds on nonlinear examples as Table 6 demonstrates, and it also cannot handle transcendental functions. Fluctuat

Table 7. Summary of Performance Results (In Seconds)

Tool	Time (s)
Fluctuat	8.26
Fluct.(div.)	48.95
Gappa	1.22
Gappa(hints)	75.53
PRECiSA	247.94
Real2Float	475.20
Rosa opt	115.26
FPTaylor-e	130.24
FPTaylor-f	21.79
FPTaylor-g	130.52
FPTaylor-h	21.15

without subdivisions is also not as good as FPTaylor in terms of bounding error estimates. Rosa is slower than FPTaylor because it relies on an inefficient optimization algorithm implemented with Z3.

We also formally verified all results in the column *FPTaylor-f* of Table 6. For all these results, corresponding HOL Light theorems were automatically produced using our formalization of FPTaylor described in Section 6.2. Verification time for 12 benchmarks, excluding *doppler1-3* and *jetEngine*, was 10.5 minutes. *jetEngine* took 28 minutes, and *doppler1-3* took an average of 37 minutes each. Such performance figures match up with the state of the art (e.g., the Flyspeck project—a formal proof of the Kepler conjecture), considering that even results pertaining to basic arithmetic operations must be formally derived from primitive definitions.

7.4 Comparison of Backend Optimizers

Table 8 gives the experimental results for different backend optimizers. This study was done to highlight the fact that different backend optimizers handle certain benchmarks with different levels of precision and efficiency. In terms of precision, on most benchmarks all the backends produce the same or very similar results. However, Z3 does occasionally produce very pessimistic results and/or is much slower than other backends; it also cannot as of now handle transcendental functions. Simple bb and Gelpia are comparable in their performance and precision on these benchmarks. In Appendix B, we present results of an additional evaluation of different backend optimizers we performed on a set of synthetically created harder benchmarks. This is a preliminary study given the synthetic nature of the used benchmarks. As can be observed in Table 10, Gelpia offers best performance on most harder benchmarks when the monolithic optimization problem is used. Such studies may help improve the heuristics employed in future optimizers, and they also indicate the need for developing advanced optimization backends to be able to efficiently tackle larger examples.

8 RELATED WORK

Taylor Series. Methods based on Taylor series have a rich history in floating-point reasoning, including algorithms for constructing symbolic Taylor series expansions for round-off errors (Miller

Table 8. Experimental Results for Different Backend Optimizers

Benchmark	FPTaylor-b (Simple bb)	FPTaylor-f (Gelpia)	FPTaylor-j (Z3)
sine	**5.6e-16**	**5.6e-16**	**5.6e-16**
	0.88	1.21	0.77
sineOrder3	9.6e-16	9.6e-16	**9.5e-16**
	0.78	0.99	0.61
sqroot	**7.0e-16**	**7.0e-16**	**7.0e-16**
	0.78	1.06	0.57
t_div_t1	5.8e-14	5.8e-14	**5.7e-14**
	0.72	1.00	0.63
carbonGas	9.2e-09	9.2e-09	**9.1e-09**
	0.86	1.10	0.69
doppler1	**1.6e-13**	**1.6e-13**	**1.6e-13**
	1.98	2.20	2.86
doppler2	**2.9e-13**	**2.9e-13**	**2.9e-13**
	2.20	2.42	2.96
doppler3	8.3e-14	8.3e-14	**8.2e-14**
	2.45	2.15	2.51
himmilbeau	**1.4e-12**	**1.4e-12**	**1.4e-12**
	0.78	1.03	2.17
jetEngine	**1.4e-11**	**1.4e-11**	9.2e-08
	2.22	4.90	11.33
kepler0	**9.5e-14**	**9.5e-14**	1.3e-13
	0.91	1.13	10.59
kepler1	**3.6e-13**	**3.6e-13**	6.4e-13
	0.93	1.24	10.66
kepler2	**2.0e-12**	**2.0e-12**	3.2e-12
	1.38	1.87	10.69
predPrey	**1.9e-16**	**1.9e-16**	**1.9e-16**
	0.84	1.11	0.65
rigidBody1	**3.9e-13**	**3.9e-13**	**3.9e-13**
	0.73	1.02	0.58
rigidBody2	**5.3e-11**	**5.3e-11**	**5.3e-11**
	0.78	1.05	10.54
turbine1	**2.4e-14**	**2.4e-14**	**2.4e-14**
	0.88	1.15	2.24
turbine2	**2.6e-14**	**2.6e-14**	1.7e-13
	0.9	1.09	10.59
turbine3	**1.3e-14**	**1.3e-14**	7.5e-14
	0.87	1.15	10.61
verhulst	**3.3e-16**	**3.3e-16**	**3.3e-16**
	0.76	1.01	0.56
azimuth	**1.2e-14**	**1.2e-14**	–
	3.89	5.23	–
hartman3	**5.8e-15**	7.0e-15	–
	36.096	12.99	–

(Continued)

Table 8. Continued

Benchmark	FPTaylor-b (Simple bb)	FPTaylor-f (Gelpia)	FPTaylor-j (Z3)
logexp	**2.1e-15**	**2.1e-15**	–
	0.72	0.98	–
sphere	**1.1e-14**	**1.1e-14**	–
	2.91	5.11	–

Grey rows show computed round-off errors; white rows show runtimes in seconds; bold font marks best results for each benchmark.

1975; Stoutemyer 1977; Gáti 2012; Mutrie et al. 1992), and stability analysis. These works do not cover round-off error estimation. Our key innovations include computation of the second-order error term in Taylor expansions and global optimization of symbolic first-order terms. Taylor expansions are also used to strictly enclose values of floating-point computations (Revol et al. 2005). Note that in this case round-off errors are not computed directly and cannot be extracted from computed enclosures without large overestimations.

Abstraction-Based Methods. Various abstraction-based methods (including abstract interpretation (Cousot and Cousot 1977)) are widely used for analysis of floating-point computations. Abstract domains for floating-point values include intervals (Moore 1966), affine forms (Stolfi and de Figueiredo 2003), and general polyhedra (Chen et al. 2008). There exist different tools based on these abstract domains. Gappa (Daumas and Melquiond 2010) is a tool for checking different aspects of floating-point programs, and is used in the Frama-C verifier (Frama-C 2017). Gappa works with interval abstractions of floating-point numbers and applies rewriting rules for improving computed results. Gappa++ (Linderman et al. 2010) is an improvement of Gappa that extends it with affine arithmetic (Stolfi and de Figueiredo 2003). It also provides definitions and rules for some transcendental functions. Gappa++ is currently not supported and does not run on modern operating systems. SmartFloat (Darulova and Kuncak 2011) is a Scala library which provides an interface for computing with floating-point numbers and for tracking accumulated round-off. It uses affine arithmetic for measuring errors. Fluctuat (Delmas et al. 2009) is a tool for static analysis of floating-point programs written in C. Internally, Fluctuat uses a floating-point abstract domain based on affine arithmetic (Goubault and Putot 2011). Astrée (Cousot et al. 2005) is another static analysis tool which can compute ranges of floating-point expressions and detect floating-point exceptions. A general abstract domain for floating-point computations is described in Martel (2006). Based on this work, a tool called RangeLab is implemented (Martel 2011) and a technique for improving accuracy of floating-point computations is presented (Martel 2009). Ponsini et al. (2014) propose constraint solving techniques for improving the precision of floating-point abstractions. Our results show that interval abstractions and affine arithmetic can yield pessimistic error bounds for nonlinear computations.

PRECiSA (Titolo 2017) is a tool that converts floating-point programs into lemmas for the PVS proof assistant. This is done by representing errors of floating-point operations symbolically in terms of the errors of the operands relative to the real-valued operations. PRECiSA's error formulas also include constraints enabling detection of invalid operator inputs and analysis of branching conditionals. Furthermore, PRECiSA propagates error conditionals into function calls and branches, thereby enabling more precise analysis in some cases. Hence, unlike FPTaylor, PRECiSA is capable of analyzing entire programs, including function calls and divergent branches. However, as our experiments show, FPTaylor greatly outperforms PRECiSA on straight-line floating-point routines both in terms of precision and efficiency.

SMT. The work closest to ours is Rosa (Darulova and Kuncak 2014, 2017) in which they combine affine arithmetic and an optimization method based on an SMT solver for estimating round-off errors. Their tool Rosa keeps the result of a computation in a symbolic form and uses an SMT solver for finding accurate bounds of computed expressions. The main difference from our work is representation of round-off errors with numerical (not symbolic) affine forms in Rosa. For nonlinear arithmetic, this representation leads to overapproximation of error, as it loses vital dependency information between the error terms. Our method keeps track of these dependencies by maintaining symbolic representation of all first-order error terms in the corresponding Taylor series expansion. Another difference is our usage of rigorous global optimization which is more efficient than using SMT-based binary search for optimization.

While abstract interpretation techniques are not designed to prove general bit-precise results, the use of bit-blasting combined with SMT solving is pursued by Brillout et al. (2009). Recently, a preliminary standard for floating-point arithmetic in SMT solvers was developed (Rümmer and Wahl 2010). Z3 (de Moura and Bjørner 2008) and MathSAT 5 (Cimatti et al. 2013) SMT solvers support this standard, but only partially. For example, Z3 lacks support for casts from floating-points into reals and vice versa, which is typically needed for computing errors of floating-point computations. There exist several other tools which use SMT solvers for reasoning about floating-point numbers. FPhile (Paganelli and Ahrendt 2013) verifies stability properties of simple floating-point programs. It translates a program into an SMT formula encoding low- and high-precision versions, and containing an assertion that the two are close enough. FPhile uses Z3 as its backend SMT solver. Leeser et al. (2014) translate a given floating-point formula into a corresponding formula for real numbers with appropriately defined rounding operators. Ariadne (Barr et al. 2013) relies on SMT solving for detecting floating-point exceptions. Haller et al. (2012) lift the conflict analysis algorithm of SMT solvers to abstract domains to improve their efficacy of floating-point reasoning.

In general, the lack of scalability of SMT solvers used by themselves has been observed in other works (Darulova and Kuncak 2014). Since existing SMT solvers do not directly support mixed real/floating-point reasoning as noted above, one must often resort to non-standard approaches for encoding properties of round-off errors in computations (e.g., using low- and high-precision versions of the same computation).

Optimization-Based Methods. Magron et al. (2017) introduce a method for estimating absolute round-off errors in floating-point nonlinear programs based on semidefinite programming. The approach works by decomposing the round-off error into an affine part with respect to the error variable e and a higher-order part. Bounds on the higher-order error part are obtained similar to how it is done in FPTaylor. For the affine part, instead of using global optimization, the authors employ a relaxation procedure based on semidefinite programming. Lee et al. (2016) proposed a verification method that combines instruction rewriting and rigorous precision measurement using global optimization. A distinguishing feature of their work is that they can handle bit-manipulation operations over floating-points.

Proof Assistants. An ultimate way to verify floating-point programs is to give a formal proof of their correctness. To achieve this goal, there exist several formalizations of the floating-point standard in proof assistants (Melquiond 2012; Harrison 2006). Boldo et al. (2013) formalized a nontrivial floating-point program for solving a wave equation. This work partially relies on Gappa, which can also produce formal certificates for verifying floating-point properties in the Coq proof assistant (Coq 2016).

Many tools and frameworks in this space separately handle the concerns of straight-line code analysis and conditional/loop analysis (Boldo et al. 2013; Boldo and Melquiond 2011; Boldo et al. 2009, 2015; Goodloe et al. 2013; Damouche et al. 2017). They typically achieve this by decomposing the overall verification problem down to simpler checks at the straight-line program level, which are then discharged by simpler tools. Our tool contribution should be viewed as a powerful assistant for such more general frameworks.

9 CONCLUSIONS

We presented a new method to estimate round-off errors of floating-point computations called Symbolic Taylor Expansions. We formally establish the correctness of our method, and also describe a tool FPTaylor that implements it. FPTaylor is one of a small number of tools that rigorously handles transcendental functions. It achieves tight overapproximation estimates of errors—especially for nonlinear expressions. FPTaylor is not designed to be a tool for complete analysis of floating-point programs. It cannot handle conditionals and loops directly; instead, it can be used as an external decision procedure for program verification tools such as Frama-C (Frama-C 2017) or SMACK (Rakamarić and Emmi 2014), or within rigorous floating-point optimization and synthesis tools such as FPTuner (Chiang et al. 2017). Conditional expressions can be verified in FPTaylor in the same way as it is done in Rosa (Darulova and Kuncak 2014) (see Section 6.3 for details).

In addition to experimenting with more examples, a promising application of FPTaylor is in error analysis of algorithms that can benefit from reduced or mixed-precision computations. Another potential application of FPTaylor is its integration with a recently released tool Herbie (Panchekha et al. 2015) that improves the accuracy of numerical programs. Herbie relies on testing for round-off error estimations. FPTaylor can provide strong guarantees for numerical expressions produced by Herbie. Ideas presented in this article can be directly incorporated into existing tools. For instance, an implementation similar to Gappa++ (Linderman et al. 2010) can be achieved by incorporating our error estimation method inside Gappa (Daumas and Melquiond 2010); the Rosa compiler (Darulova and Kuncak 2014) can also be easily extended with our technique.

APPENDICES

A ADDITIONAL EXPERIMENTAL RESULTS

Table 9 presents a summary of all our experimental results.

B GLOBAL OPTIMIZER EVALUATION ON HARD EXAMPLES

Table 10 presents a study of various backend optimizers on some hard examples from the area of optimization (Surjanovic and Bingham 2017; Weisstein 2017a, 2017b). This study demonstrates the need to invest in good backend optimization approaches, in addition to developing improved error estimation approaches.

Table 9. Experimental Results for Absolute Round-Off Error Bounds

Benchmark	Fluctuat	Fluctuat (subdiv.)	FPT-a	FPT-b	FPT-c	FPT-d	FPT-e	FPT-f	FPT-g	FPT-h	FPT-i	FPT-j	Gappa (hints)	Gappa	Gappa (simple hints)	PRECiSA	R2Float	Rosa	Rosa (opt)
carbonGas	4.5E-08	1.2E-08	1.0E-08	9.1E-09	7.0E-09	**5.9E-09**	1.0E-08	9.1E-09	7.0E-09	**5.9E-09**	1.0E-08	9.0E-09	6.0E-09	2.6E-08	8.8E-09	7.3E-09	2.2E-08	1.6E-08	1.6E-08
	1.56	0.54	5.69	0.86	4.57	0.85	9.13	1.10	7.39	1.08	6.65	0.69	2.35	0.12	0.18	5.89	4.66	34.74	6.03
doppler1	3.9E-13	1.3E-13	1.6E-13	1.6E-13	1.3E-13	**1.2E-13**	1.6E-13	1.6E-13	1.3E-13	**1.2E-13**	1.6E-13	1.6E-13	1.6E-13	2.0E-13	1.8E-13	2.6E-13	7.6E-12	2.7E-13	2.4E-13
	0.52	6.30	8.92	1.98	13.95	1.77	9.18	2.20	9.61	1.98	6.78	2.86	3.31	0.05	0.10	17.43	13.20	23.34	10.86
doppler2	9.8E-13	2.4E-13	2.9E-13	2.9E-13	2.3E-13	**2.2E-13**	2.9E-13	2.9E-13	2.3E-13	**2.2E-13**	2.9E-13	2.9E-13	2.9E-13	3.9E-13	3.2E-13	5.4E-13	1.5E-11	6.5E-13	5.6E-13
	0.53	6.15	9.63	2.20	13.31	2.01	10.02	2.42	9.19	2.21	6.87	2.96	3.37	0.05	0.11	17.08	13.34	23.69	10.58
doppler3	1.6E-13	7.1E-14	8.2E-14	8.2E-14	6.7E-14	**6.6E-14**	8.2E-14	8.2E-14	6.7E-14	**6.6E-14**	8.2E-14	8.2E-14	8.7E-14	1.1E-13	9.2E-14	1.3E-13	8.5E-12	1.0E-13	1.0E-13
	0.81	6.46	11.68	2.45	14.52	1.95	9.93	2.15	8.94	1.89	6.50	2.51	3.33	0.05	0.09	17.09	13.06	32.15	9.41
jet	*4.1E-08*	1.1E-10	1.5E-11	1.3E-11	1.2E-11	**1.0E-11**	1.5E-11	1.3E-11	1.2E-11	**1.0E-11**	1.3E-09	*9.1E-08*	*4.4E+03*	*8.2E+06*	*1.3E+05*	crash	OoM	4.9E-09	4.9E-09
	0.63	1.45	14.46	2.22	17.53	2.23	19.52	4.90	21.58	2.85	193.61	11.33	27.64	0.21	0.34	crash	OoM	268.50	84.01
predPrey	2.5E-16	2.4E-16	1.9E-16	1.8E-16	**1.6E-16**	**1.6E-16**	1.9E-16	1.8E-16	**1.6E-16**	**1.6E-16**	1.8E-16	1.8E-16	1.7E-16	2.0E-16	1.9E-16	1.7E-16	2.5E-16	2.0E-16	2.0E-16
	0.51	0.19	2.86	0.84	2.87	0.83	4.68	1.11	4.63	1.08	3.42	0.65	1.40	0.04	0.06	6.48	4.00	31.81	11.21
rigidBody1	3.2E-13	3.2E-13	3.9E-13	3.9E-13	**2.9E-13**	**2.9E-13**	3.9E-13	3.9E-13	**2.9E-13**	**2.9E-13**	3.9E-13	3.9E-13	**2.9E-13**	**2.9E-13**	**2.9E-13**	**2.9E-13**	5.3E-13	3.2E-13	3.2E-13
	0.40	1.97	3.16	0.73	3.14	0.73	5.09	1.02	5.18	0.99	4.16	0.58	1.43	0.07	0.11	6.35	3.09	4.70	3.49
rigidBody2	**3.6E-11**	**3.6E-11**	5.2E-11	5.2E-11	**3.6E-11**	**3.6E-11**	5.2E-11	5.2E-11	**3.6E-11**	**3.6E-11**	5.2E-11	5.2E-11	**3.6E-11**	**3.6E-11**	**3.6E-11**	**3.6E-11**	6.5E-11	**3.6E-11**	**3.6E-11**
	0.23	3.87	5.28	0.78	5.23	0.79	8.52	1.05	8.50	1.02	7.49	10.54	2.22	0.10	0.15	11.40	1.09	7.59	4.21
sine	8.0E-16	7.4E-16	6.8E-16	5.5E-16	5.3E-16	**4.4E-16**	6.7E-16	5.6E-16	5.3E-16	**4.4E-16**	6.7E-16	5.5E-16	6.9E-16	1.1E-15	8.0E-16	8.0E-16	6.0E-16	5.2E-16	5.7E-16
	0.25	0.12	5.00	0.88	5.00	0.91	8.09	1.21	8.09	1.15	6.34	0.77	25.44	0.09	0.14	12.84	4.96	9.69	5.20
sineOrder3	1.2E-15	1.1E-15	1.0E-15	9.5E-16	6.7E-16	**5.9E-16**	1.0E-15	9.5E-16	6.7E-16	**5.9E-16**	1.0E-15	9.4E-16	6.5E-16	8.9E-16	7.6E-16	1.2E-15	1.2E-15	1.0E-15	1.0E-15
	0.37	0.09	3.14	0.78	2.88	0.75	5.12	0.99	4.56	1.02	4.07	0.61	2.09	0.05	0.06	6.38	2.23	5.44	3.82
sqroot	6.8E-16	6.8E-16	7.1E-16	7.0E-16	5.1E-16	**5.0E-16**	7.1E-16	7.0E-16	5.1E-16	**5.0E-16**	7.1E-16	7.0E-16	5.4E-16	5.7E-16	5.4E-16	6.8E-16	1.3E-15	6.2E-16	6.2E-16
	0.39	0.10	4.61	0.78	4.55	0.78	7.46	1.06	7.36	1.02	5.03	0.57	5.07	0.12	0.17	9.14	4.23	6.43	4.20
t_div_t1	*1.1E-10*	*2.8E-12*	5.7E-14	5.7E-14	**2.2E-16**	**2.2E-16**	5.7E-14	5.7E-14	**2.2E-16**	**2.2E-16**	5.7E-14	5.7E-14	*1.0E+01*	*1.0E+03*	*2.5E+02*	*2.3E-13*	5.5E-16	*5.7E-11*	*5.7E-11*
	0.31	0.09	1.42	0.72	1.45	0.74	2.28	1.00	2.32	0.96	2.30	0.63	0.19	0.02	0.02	6.33	125.31	10.92	6.56
turbine1	9.2E-14	3.1E-14	2.3E-14	2.3E-14	1.8E-14	**1.7E-14**	2.3E-14	2.3E-14	1.8E-14	**1.7E-14**	2.3E-14	2.3E-14	2.4E-14	8.4E-14	3.9E-14	3.8E-14	*2.5E-11*	6.0E-14	6.0E-14
	0.56	5.05	6.75	0.88	7.58	0.96	9.77	1.15	9.81	1.11	7.97	2.24	5.54	0.11	0.23	23.86	136.35	24.52	9.10
turbine2	1.3E-13	2.6E-14	3.1E-14	2.5E-14	2.4E-14	**2.0E-14**	3.1E-14	2.6E-14	2.4E-14	**2.0E-14**	3.1E-14	1.6E-13	3.3E-14	1.3E-13	4.0E-14	3.2E-14	2.1E-12	7.7E-14	7.7E-14
	0.79	3.98	5.31	0.90	5.58	1.03	7.95	1.09	8.00	1.18	6.93	10.59	3.95	0.08	0.15	18.91	8.30	11.96	5.47

(Continued)

Table 9. Continued

Benchmark	Fluctuat	Fluctuat (subdiv.)	FPT-a	FPT-b	FPT-c	FPT-d	FPT-e	FPT-f	FPT-g	FPT-h	FPT-i	FPT-j	Gappa (hints)	Gappa	Gappa (simple hints)	PRECiSA	R2Float	Rosa	Rosa (opt)
turbine3	7.0E-14	1.3E-14	1.7E-14	1.2E-14	1.0E-14	**9.6E-15**	1.7E-14	1.2E-14	1.0E-14	**9.6E-15**	1.7E-14	7.4E-14	*3.5E-01*	*4.0E+01*	*1.0E+01*	1.7E-14	*1.7E-11*	4.6E-14	4.6E-14
	0.62	5.08	6.06	0.87	6.76	1.00	9.59	1.15	9.86	1.21	8.26	10.61	6.30	0.11	0.24	20.69	137.37	26.22	7.61
verhulst	5.5E-16	4.8E-16	3.5E-16	3.2E-16	2.6E-16	**2.5E-16**	3.5E-16	3.2E-16	2.6E-16	**2.5E-16**	3.5E-16	3.2E-16	2.8E-16	4.2E-16	3.2E-16	2.8E-16	4.7E-16	4.7E-16	4.7E-16
	0.26	0.09	2.13	0.76	2.12	0.77	3.48	1.01	3.42	0.99	2.49	0.56	0.42	0.03	0.05	4.29	2.53	12.43	7.16
azimuth	–	–	1.5E-14	1.2E-14	1.1E-14	**8.8E-15**	1.5E-14	1.2E-14	1.1E-14	8.9E-15	–	–	–	–	–	crash	2.8E-13	–	–
	–	–	77.94	3.89	74.17	3.69	62.29	5.23	60.54	4.62	–	–	–	–	–	crash	2.30	–	–
logexp	–	–	2.1E-15	2.1E-15	2.1E-15	**2.0E-15**	2.1E-15	2.1E-15	2.1E-15	**2.0E-15**	–	–	–	–	–	–	2.5E-15	–	–
	–	–	1.77	0.72	1.75	0.71	2.88	0.98	2.85	0.96	–	–	–	–	–	–	0.81	–	–
sphere	–	–	1.3E-14	1.0E-14	1.0E-14	**8.2E-15**	1.3E-14	1.0E-14	1.0E-14	8.3E-15	–	–	–	–	–	1.0E-14	1.5E-14	–	–
	–	–	3.47	2.91	4.96	4.01	5.66	5.11	6.48	5.37	–	–	–	–	–	25.91	0.08	–	–
himmilbeau	1.0E-12	1.0E-12	1.3E-12	1.3E-12	1.0E-12	1.0E-12	1.3E-12	1.3E-12	1.0E-12	1.0E-12	1.3E-12	1.3E-12	**8.5E-13**	1.0E-12	1.0E-12	1.0E-12	1.4E-12	1.0E-12	1.0E-12
	0.09	0.34	5.34	0.78	4.50	0.78	7.50	1.03	6.88	1.03	6.73	2.17	1.87	0.08	0.13	26.28	0.69	10.15	4.73
kepler0	1.1E-13	1.0E-13	1.0E-13	9.5E-14	9.2E-14	**7.5E-14**	1.0E-13	9.5E-14	9.2E-14	**7.5E-14**	1.0E-13	1.3E-13	1.1E-13	1.2E-13	1.1E-13	1.2E-13	1.2E-13	8.3E-14	8.3E-14
	0.12	8.59	8.22	0.91	12.54	3.02	12.52	1.13	16.86	1.31	9.96	10.59	7.34	0.13	0.42	37.58	0.87	8.01	5.71
kepler1	5.1E-13	3.5E-13	4.5E-13	3.6E-13	3.7E-13	**2.9E-13**	4.5E-13	3.6E-13	3.7E-13	**2.9E-13**	5.5E-13	6.4E-13	4.7E-13	5.4E-13	4.9E-13	crash	4.7E-13	4.1E-13	3.8E-13
	0.11	157.75	13.81	0.93	17.88	2.11	18.41	1.24	22.34	2.08	95.04	10.66	10.69	0.24	0.61	crash	22.90	138.99	18.82
kepler2	2.6E-12	2.2E-12	2.1E-12	2.0E-12	1.9E-12	**1.6E-12**	2.1E-12	2.0E-12	1.9E-12	**1.6E-12**	2.6E-12	3.1E-12	2.4E-12	2.9E-12	2.5E-12	crash	2.1E-12	2.2E-12	2.0E-12
	0.12	22.41	24.76	1.38	33.57	1.02	27.83	1.87	37.07	1.30	155.10	10.69	24.18	0.45	1.15	crash	16.63	69.97	19.67
hartman3	–	–	1.3E-14	5.7E-15	9.3E-15	**3.6E-15**	1.3E-14	7.0E-15	9.3E-15	4.6E-15	–	–	–	–	–	–	3.0E-13	–	–
	–	–	48.21	36.10	37.59	29.83	68.90	12.99	57.51	12.67	–	–	–	–	–	–	3.61	–	–

Grey rows show computed round-off errors; white rows show runtimes in seconds; bold font marks best results for each benchmark; italic font marks pessimistic results at least three orders of magnitude worse than the best ones; *OoM* marks the tool running out of memory.

Table 10. Experimental Results for Different Backend Optimizers on Larger Benchmarks

Benchmark	FPTaylor-a	FPTaylor-b	FPTaylor-c	FPTaylor-d	FPTaylor-e	FPTaylor-f	FPTaylor-g	FPTaylor-h	FPTaylor-i	FPTaylor-j
cheb_02	**5.6e-16**	**5.6e-16**	**5.6e-16**	**5.6e-16**	**5.6e-16**	**5.6e-16**	**5.6e-16**	**5.6e-16**	**5.6e-16**	**5.6e-16**
	1.9	0.7	8.61	5.82	3.63	0.96	4.78	0.89	13.7	0.58
cheb_07	**9.8e-15**	**9.8e-15**	1.8e-14	1.8e-14	9.9e-15	**9.8e-15**	1.8e-14	1.8e-14	**9.8e-15**	**9.8e-15**
	6.02	1.12	357.04	286.47	8.29	2.49	31.29	1.5	7.72	2.89
cheb_10	**2.1e-14**	**2.1e-14**	1.6e-13	1.6e-13	**2.1e-14**	**2.1e-14**	1.6e-13	1.6e-13	crash	crash
	16.66	5.29	2448.01	1846.75	20.44	43.19	245.83	8.37	crash	crash
legendre_05	5.1e-15	5.1e-15	**4.3e-15**	**4.3e-15**	5.1e-15	5.1e-15	**4.3e-15**	**4.3e-15**	5.1e-15	5.1e-15
	6.14	1.03	136.66	198.07	9.74	1.64	9.76	1.39	7.63	2.73
legendre_08	**1.4e-14**	**1.4e-14**	1.8e-14	1.5e-14	**1.4e-14**	**1.4e-14**	1.8e-14	1.5e-14	**1.4e-14**	5.4e-13
	15.64	4.06	1706.3	1656.39	22.03	14.57	31.93	7.74	20.91	381.1
legendre_10	**2.2e-14**	**2.2e-14**	6.3e-14	4.2e-14	**2.2e-14**	**2.2e-14**	6.3e-14	4.2e-14	crash	crash
	49.32	21.06	7415.83	5936.07	60.31	157.84	64.09	83.23	crash	crash
trid_02	1.6e-11	1.5e-11	1.7e-11	**1.2e-11**	1.6e-11	1.5e-11	1.3e-11	**1.2e-11**	1.6e-11	1.5e-11
	3.43	0.74	13.68	0.76	5.65	1.1	10.93	1.02	4.28	0.89
trid_10	2.5e-10	2.6e-10	2.0e-10	2.0e-10	2.5e-10	2.5e-10	2.0e-10	**1.9e-10**	2.7e-10	2.7e-10
	230.81	165.69	189.75	229.51	133.64	65.82	185.93	101.54	1777.16	101.01

Grey rows show computed round-off errors; white rows show runtimes in seconds; bold font marks best results for each benchmark.

ACKNOWLEDGMENTS

We would like to thank Nelson Beebe, Wei-Fan Chiang, and John Harrison for their feedback and encouragement.

REFERENCES

Jean-Marc Alliot, Nicolas Durand, David Gianazza, and Jean-Baptiste Gotteland. 2012a. Finding and proving the optimum: Cooperative stochastic and deterministic search. In *Proceedings of the 20th European Conference on Artificial Intelligence (ECAI'12)*. ACM, 55–60. DOI: https://doi.org/10.3233/978-1-61499-098-7-55

Jean-Marc Alliot, Nicolas Durand, David Gianazza, and Jean-Baptiste Gotteland. 2012b. Implementing an interval computation library for OCaml on x86/AMD64 architectures (short paper). In *Proceedings of the International Conference on Functional Programming (ICFP'12)*. ACM.

Marc Andrysco, Ranjit Jhala, and Sorin Lerner. 2016. Printing floating-point numbers: A faster, always correct method. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'16)*. ACM, 555–567. DOI: https://doi.org/10.1145/2837614.2837654

Earl T. Barr, Thanh Vo, Vu Le, and Zhendong Su. 2013. Automatic detection of floating-point exceptions. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'13)*. ACM, 549–560. DOI: https://doi.org/10.1145/2429069.2429133

Jesse Bingham and Joe Leslie-Hurd. 2014. Verifying relative error bounds using symbolic simulation. In *Proceedings of the 26th International Conference on Computer Aided Verification (CAV'14)*. Springer, 277–292. DOI: https://doi.org/10.1007/978-3-319-08867-9_18

Sylvie Boldo, François Clément, Jean-Christophe Filliâtre, Micaela Mayero, Guillaume Melquiond, and Pierre Weis. 2013. Wave equation numerical resolution: A comprehensive mechanized proof of a C program. *Journal of Automated Reasoning (JAR)* 50, 4 (2013), 423–456. DOI: https://doi.org/10.1007/s10817-012-9255-4

Sylvie Boldo, Jean-Christophe Filliâtre, and Guillaume Melquiond. 2009. Combining Coq and Gappa for certifying floating-point programs. In *Proceedings of the 16th Symposium on the Integration of Symbolic Computation and Mechanised Reasoning (CALCULEMUS'09)*. Springer, 59–74. DOI: https://doi.org/10.1007/978-3-642-02614-0_10

Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. 2015. Verified compilation of floating-point computations. *Journal of Automated Reasoning (JAR)* 54, 2 (2015), 135–163. DOI: https://doi.org/10.1007/s10817-014-9317-x

Sylvie Boldo and Guillaume Melquiond. 2011. Flocq: A unified library for proving floating-point algorithms in Coq. In *Proceedings of the IEEE Symposium on Computer Arithmetic*. 243–252.

Angelo Brillout, Daniel Kroening, and Thomas Wahl. 2009. Mixed abstractions for floating-point arithmetic. In *Formal Methods in Computer-Aided Design (FMCAD'09)*. 69–76.

Liqian Chen, Antoine Miné, and Patrick Cousot. 2008. A sound floating-point polyhedra abstract domain. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems (APLAS'08)*. Springer, 3–18. DOI: https://doi.org/10.1007/978-3-540-89330-1_2

Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. 2017. Rigorous floating-point mixed-precision tuning. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'17)*. ACM, 300–315. DOI: https://doi.org/10.1145/3009837.3009846

Wei-Fan Chiang, Ganesh Gopalakrishnan, Zvonimir Rakamarić, and Alexey Solovyev. 2014. Efficient search for inputs causing high floating-point errors. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'14)*. ACM, 43–52. DOI: https://doi.org/10.1145/2555243.2555265

Alessandro Cimatti, Alberto Griggio, BastiaanJoost Schaafsma, and Roberto Sebastiani. 2013. The MathSAT5 SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'13)*, Lecture Notes in Computer Science, Vol. 7795. Springer, 93–107. DOI: https://doi.org/10.1007/978-3-642-36742-7_7

Coq 2016. The Coq Proof Assistant. http://coq.inria.fr.

Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'77)*. ACM, 238–252.

Patrick Cousot, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2005. The ASTRÉE analyser. In *Proceedings of the 14th European Symposium on Programming Languages and Systems (ESOP'05)*, Lecture Notes in Computer Science, Vol. 3444. Springer, 21–30.

Nasrine Damouche, Matthieu Martel, and Alexandre Chapoutot. 2017. Improving the numerical accuracy of programs by automatic transformation. *International Journal on Software Tools for Technology Transfer (STTT)* 19, 4 (2017), 427–448. DOI: https://doi.org/10.1007/s10009-016-0435-0

Catherine Daramy, David Defour, Florent de Dinechin, and Jean-Michel Muller. 2003. CR-LIBM: A correctly rounded elementary function library. *Advanced Signal Processing Algorithms, Architectures, and Implementations XIII, SPIE* 5205 (2003), 458–464.

Eva Darulova and Viktor Kuncak. 2011. Trustworthy numerical computation in Scala. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'11)*. ACM, 325–344. DOI: https://doi.org/10.1145/2048066.2048094

Eva Darulova and Viktor Kuncak. 2014. Sound compilation of reals. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'14)*. ACM, 235–248.

Eva Darulova and Viktor Kuncak. 2017. Towards a compiler for reals. *ACM Trans. Program. Lang. Syst.* 39, 2 (2017), 8:1–8:28. DOI: https://doi.org/10.1145/3014426

Marc Daumas and Guillaume Melquiond. 2010. Certification of bounds on expressions involving rounded operators. *ACM Trans. Math. Software* 37, 1 (2010), Article 2, 20 pages.

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, Lecture Notes in Computer Science, Vol. 4963. Springer, Berlin, 337–340. DOI: https://doi.org/10.1007/978-3-540-78800-3_24

David Delmas, Eric Goubault, Sylvie Putot, Jean Souyris, Karim Tekkal, and Franck Védrine. 2009. Towards an industrial use of FLUCTUAT on safety-critical avionics software. In *Formal Methods for Industrial Critical Systems (FMICS'09)*, Lecture Notes in Computer Science, Vol. 5825. Springer, Berlin, 53–69. DOI: https://doi.org/10.1007/978-3-642-04570-7_6

Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. 2007. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Software* 33, 2 (2007).

Frama-C 2017. Frama-C Software Analyzers. Retrieved October 13, 2017 from http://frama-c.com/index.html.

Sicun Gao, Soonho Kong, and Edmund M. Clarke. 2013. dReal: An SMT solver for nonlinear theories over the reals. In *Proceedings of the 24th International Conference on Automated Deduction (CADE'13)*. 208–214.

Attila Gáti. 2012. Miller analyzer for Matlab: A Matlab package for automatic roundoff analysis. *Comput. Inf.* 31, 4 (2012), 713–726.

Gelpia 2017. Gelpia: A Global Optimizer for Real Functions. Retrieved October 13, 2017 from https://github.com/soarlab/gelpia.

Dimitra Giannakopoulou, Falk Howar, Malte Isberner, Todd Lauderdale, Zvonimir Rakamarić, and Vishwanath Raman. 2014. Taming test inputs for separation assurance. In *Proceedings of the 29th IEEE/ACM International Conference on Automated Software Engineering (ASE'14)*. ACM, 373–384.

David Goldberg. 1991. What every computer scientist should know about floating-point arithmetic. *Comput. Surveys* 23, 1 (March 1991), 5–48. DOI: https://doi.org/10.1145/103162.103163

Alwyn Goodloe, César Muñoz, Florent Kirchner, and Loïc Correnson. 2013. Verification of numerical programs: From real numbers to floating point numbers. In *Proceedings of the 5th NASA Formal Methods Symposium (NFM'13)*, Lecture Notes in Computer Science, Vol. 7871. Springer, 441–446.

Frédéric Goualard. 2014. How do you compute the midpoint of an interval? *ACM Trans. Math. Software* 40, 2, Article 11 (2014), 25 pages.

Frédéric Goualard. 2017. GAOL (Not Just Another Interval Library). Retrieved October 13, 2017 from http://frederic.goualard.net/#research-software.

Eric Goubault and Sylvie Putot. 2011. Static analysis of finite precision computations. In *Proceedings of the International Workshop on Verification, Model Checking, and Abstract Interpretation (VMCAI'11)*, Lecture Notes in Computer Science, Vol. 6538. Springer, Berlin, 232–247. DOI: https://doi.org/10.1007/978-3-642-18275-4_17

Leopold Haller, Alberto Griggio, Martin Brain, and Daniel Kroening. 2012. Deciding floating-point logic with systematic abstraction. In *Formal Methods in Computer-Aided Design (FMCAD'12)*. 131–140.

John Harrison. 2000. Formal verification of floating point trigonometric functions. In *Proceedings of the 3rd International Conference on Formal Methods in Computer-Aided Design (FMCAD'00)*, Lecture Notes in Computer Science, Vol. 1954. Springer, 254–270.

John Harrison. 2006. Floating-point verification using theorem proving. In *SFM 2006*. Lecture Notes in Computer Science, Vol. 3965. Springer Berlin Heidelberg, 211–242. DOI: https://doi.org/10.1007/11757283_8

John Harrison. 2009. HOL light: An overview. In *TPHOLs 2009*, Lecture Notes in Computer Science, Vol. 5674. Springer Berlin Heidelberg, 60–66. DOI: https://doi.org/10.1007/978-3-642-03359-9_4

2008. IEEE Standard for floating-point arithmetic. *IEEE Std 754-2008* (2008), 1–70.

Charles Jacobsen, Alexey Solovyev, and Ganesh Gopalakrishnan. 2015. A parameterized floating-point formalization in HOL light. *Electro. Notes Theor. Comput. Sci.* 317 (2015), 101–107.

Steven G. Johnson. 2017. The NLopt Nonlinear-Optimization Package. Retrieved October 13, 2017 from https://nlopt.readthedocs.io/en/latest/.

William Kahan. 2006. How Futile Are Mindless Assessments of Roundoff in Floating-Point Computation? Retrieved October 13, 2017 from https://people.eecs.berkeley.edu/~wkahan/Mindless.pdf.

R. Baker Kearfott. 2009. GlobSol user guide. *Opt. Methods Softw.* 24, 4–5 (2009), 687–708.

Yahia Lebbah. 2009. ICOS: A branch and bound based solver for rigorous global optimization. *Optim. Methods Softw.* 24, 4–5 (2009), 709–726.

Wonyeol Lee, Rahul Sharma, and Alex Aiken. 2016. Verifying bit-manipulations of floating-point. In *Proceedings of the 37th Annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'16)*. 70–84.

Miriam Leeser, Saoni Mukherjee, Jaideep Ramachandran, and Thomas Wahl. 2014. Make it real: Effective floating-point reasoning via exact arithmetic. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE'14)*. 1–4.

Michael D. Linderman, Matthew Ho, David L. Dill, Teresa H. Meng, and Garry P. Nolan. 2010. Towards program optimization through automated analysis of numerical precision. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'10)*. ACM, 230–237. DOI: https://doi.org/10.1145/1772954.1772987

Victor Magron, George Constantinides, and Alastair Donaldson. 2017. Certified roundoff error bounds using semidefinite programming. *ACM Trans. Math. Software* 43, 4 (Jan. 2017), Article 34, 31 pages. DOI: https://doi.org/10.1145/3015465

Matthieu Martel. 2006. Semantics of roundoff error propagation in finite precision calculations. *High.-Order Symb. Comput.* 19, 1 (2006), 7–30. DOI: https://doi.org/10.1007/s10990-006-8608-2

Matthieu Martel. 2009. Program transformation for numerical precision. In *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM'09)*. ACM, 101–110.

Matthieu Martel. 2011. RangeLab: A static-analyzer to bound the accuracy of finite-precision computations. In *Proceedings of the 13th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'11)*. IEEE Computer Society, 118–122.

Érik Martin-Dorel, Laurence Rideau, Laurent Théry, Micaela Mayero, and Ioana Pasca. 2013. Certified, efficient and sharp univariate Taylor models in Coq. In *Proceedings of the 2013 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'13)*. IEEE, 193–200.

Maxima. 2013. Maxima, a Computer Algebra System. Version 5.30.0. Retrieved April 3, 2013 from http://maxima.sourceforge.net/.

Guillaume Melquiond. 2012. Floating-point arithmetic in the Coq system. *Inf. Comput.* 216 (2012), 14–23. DOI: https://doi.org/10.1016/j.ic.2011.09.005

Piotr Mikusinski and Michael Taylor. 2002. *An Introduction to Multivariable Analysis from Vector to Manifold*. Birkhäuser Boston.

Webb Miller. 1975. Software for roundoff analysis. *ACM . Math. Softw.* 1, 2 (1975), 108–128.

R. E. Moore. 1966. *Interval Analysis*. Prentice-Hall.

Mark P. W. Mutrie, Richard H. Bartels, and Bruce W. Char. 1992. An approach for floating-point error analysis using computer algebra. In *Papers from the International Symposium on Symbolic and Algebraic Computation (ISSAC'92)*. ACM, 284–293. DOI: https://doi.org/10.1145/143242.143332

Anthony Narkawicz and César Munoz. 2013. A formally verified generic branching algorithm for global optimization. In *Working Conference on Verified Software: Theories, Tools, and Experiments*. Springer, 326–343.

NASA. 2017. NASA World Wind Java SDK. Retrieved October 13, 2017 from http://worldwind.arc.nasa.gov/java/.

Arnold Neumaier. 2003. Taylor forms—Use and limits. *Reliab. Comput.* 9, 1 (Feb. 2003), 43–79. DOI: https://doi.org/10.1023/A:1023061927787

Arnold Neumaier. 2004. Complete search in continuous global optimization and constraint satisfaction. *Acta Num.* 13 (2004), 271–369.

OpenOpt. 2017. OpenOpt: Universal Numerical Optimization Package. Retrieved October 13, 2017 from http://openopt.org.

Gabriele Paganelli and Wolfgang Ahrendt. 2013. Verifying (in-)stability in floating-point programs by increasing precision, using SMT solving. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'13)*. 209–216. DOI: https://doi.org/10.1109/SYNASC.2013.35

Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. 2015. Automatically improving accuracy for floating point expressions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*. ACM, 1–11. DOI: https://doi.org/10.1145/2737924.2737959

Olivier Ponsini, Claude Michel, and Michel Rueher. 2014. Verifying floating-point programs with constraint programming and abstract interpretation techniques. *Autom. Softw. Eng.* (2014), 1–27. DOI: https://doi.org/10.1007/s10515-014-0154-2

Zvonimir Rakamarić and Michael Emmi. 2014. SMACK: Decoupling source language details from verifier implementations. In *Computer Aided Verification (CAV'14)*, Lecture Notes in Computer Science, Vol. 8559. Springer International Publishing, 106–113. DOI: https://doi.org/10.1007/978-3-319-08867-9_7

N. Revol, K. Makino, and M. Berz. 2005. Taylor models and floating-point arithmetic: Proof that arithmetic operations are validated in COSY. *J. Logic Algebr.c Program.* 64, 1 (2005), 135–154.

Philipp Rümmer and Thomas Wahl. 2010. An SMT-LIB theory of binary floating-point arithmetic. In *Informal Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (SMT Workshop'10)*.

Alexey Solovyev. 2017. TOPLAS FPTaylor Results Table. Retrieved October 10, 2017 from http://tinyurl.com/TOPLAS-FPTaylor-Results-Table.

Alexey Solovyev and Thomas C. Hales. 2013. Formal verification of nonlinear inequalities with Taylor interval approximations. In *NASA Formal Methods (NFM'13)*, Lecture Notes in Computer Science, Vol. 7871. Springer, Berlin, 383–397. DOI: https://doi.org/10.1007/978-3-642-38088-4_26

Alexey Solovyev, Charles Jacobsen, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. 2015. Rigorous estimation of floating-point round-off errors with symbolic Taylor expansions. In *Proceedings of the 20th International Symposium on Formal Methods Formal (FM'15)*. 532–550.

Jorge Stolfi and Luiz H. de Figueiredo. 2003. An introduction to affine arithmetic. *TEMA Trends Appl. Comput. Math.* 4, 3 (2003), 297–312.

David R. Stoutemyer. 1977. Automatic error analysis using computer algebraic manipulation. *ACM Trans. Math. Software* 3, 1 (1977), 26–43.

Microsoft Support. 2018. Floating-point arithmetic may give inaccurate results in Excel. https://support.microsoft.com/en-us/help/78113/floating-point-arithmetic-may-give-inaccurate-results-in-excel Last updated April 17, 2018.

Sonja Surjanovic and Derek Bingham. 2017. Trid Function. Retrieved October 10, 2017 from http://www.sfu.ca/%7Essurjano/trid.html Tridiagonal Examples.

Laura Titolo. 2017. Schloss Dagstuhl: Seminar Homepage. Retrieved October 10, 2017 from http://www.dagstuhl.de/en/program/calendar/semhp/?semnr=17352.

Eric Weisstein. 2017a. Chebyschev Polynomial of the First Kind—From Wolfram MathWorld. Retrieved October 10, 2017 from http://mathworld.wolfram.com/ChebyshevPolynomialoftheFirstKind.html.

Eric Weisstein. 2017b. Legendre Polynomial—From Wolfram MathWorld. Retrieved October 13, 2017 from http://mathworld.wolfram.com/LegendrePolynomial.html.

Received January 2017; revised March 2018; accepted June 2018