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

Alexey Solovyev, Charles Jacobsen,
Zvonimir Rakamarić, Ganesh Gopalakrishnan

UUCS-15-001

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

April 6, 2015

Abstract

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 overestimates. Also, there are no available rigorous approaches that handle transcendental functions. We have developed a new approach called *Symbolic Taylor Expansions* that avoids this difficulty, 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. In addition to providing far tighter upper bounds of round-off error in a vast majority of cases, FPTaylor also emits analysis certificates in the form of HOL Light proofs. We release FPTaylor along with our benchmarks for evaluation.

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

Alexey Solovyev, Charles Jacobsen,
Zvonimir Rakamarić, and Ganesh Gopalakrishnan

School of Computing, University of Utah,
Salt Lake City, UT 84112, USA
{monad,charlesj,zvonimir,ganesh}@cs.utah.edu

Abstract. 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 overestimates. Also, there are no available rigorous approaches that handle transcendental functions. We have developed a new approach called *Symbolic Taylor Expansions* that avoids this difficulty, 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. In addition to providing far tighter upper bounds of round-off error in a vast majority of cases, FPTaylor also emits analysis certificates in the form of HOL Light proofs. We release FPTaylor along with our benchmarks for evaluation.

Keywords: floating-point, round-off error analysis, global optimization

1 Introduction

Many algorithms are conceived (and even formally verified) in real numbers, but ultimately deployed using floating-point numbers. Unfortunately, the finitary nature of floating-point, along with its uneven distribution of representable numbers introduces round-off errors, as well as does not preserve many familiar laws (e.g., associativity of +) [22]. This mismatch often necessitates re-verification using tools that precisely compute round-off error bounds (e.g., as illustrated in [21]). While SMT solvers can be used for small problems [52, 24], the need to scale necessitates the use of various abstract interpretation methods [11], the most popular choices being interval [41] or affine arithmetic [54]. However, these tools very often generate pessimistic error bounds, especially when nonlinear functions are involved. No tool that is currently maintained rigorously handles transcendental functions that arise in problems such as the safe separation of aircraft [20].

The final publication was accepted to FM 2015 and is available at link.springer.com

Key to Our Approach. In a nutshell, the aforesaid difficulties arise because of a tool's attempt to abstract the "difficult" (nonlinear or transcendental) functions. Our new approach called *Symbolic Taylor Expansions* (realized in a tool FPTaylor) side-steps these issues entirely as follows. (1) We view round-off errors as "noise," and compute Taylor expansions in a symbolic form. (2) In these symbolic Taylor forms, all difficult functional expressions appear as symbolic coefficients; they do not need to be abstracted. (3) We then apply a *rigorous* global maximization method that has no trouble handling the difficult functions and can be executed sufficiently fast thanks to the ability to trade off accuracy for performance.

Let us illustrate these ideas using a simple example. First, we define *absolute round-off error* 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. Now, consider the estimation of worst case absolute round-off error in $t/(t+1)$ computed with floating-point arithmetic where $t \in [0, 999]$ is a floating-point number. (Our goal here is to demonstrate basic ideas of our method; pertinent background is in Sect. 3.) 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 $1000 = 999 + 1$. Interval abstraction replaces the expression $d = t \oplus 1$ with the abstract pair $([1, 1000], 512\epsilon)$ 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 $\frac{t}{d^2}$.[1] At this point, interval abstraction does not yield a satisfactory 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 $999 \times 512\epsilon \approx 512000\epsilon$.

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

[1] 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 $(t/d_{\text{exact}}) - (t/(d^2_{\text{exact}}))\delta$.

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 (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 Sect. 4 we demonstrate how rigorous upper bounds are derived for all error terms):

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

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.

Our combination of Taylor expansion with symbolic coefficients and global optimization yields an error bound which is $512000/2 = 256000$ times better than a naïve error estimation technique implemented in many other tools for floating-point analysis. Our approach never had to examine the inner details of / and + in our example (these could well be replaced by "difficult" functions; our technique would work the same way). The same cannot be said of SMT or interval/affine arithmetic. The key enabler is that most rigorous global optimizers deal with a very large class of functions smoothly.

Our Key Contributions:

• We describe all the details of our global optimization approach, as there seems to be a lack of awareness (even misinformation) among some researchers.

• 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 is the only tool we know providing guaranteed bounds for transcendental expressions. It handles uncertainties in input variables, supports estimation of relative and absolute round-off errors, provides a rigorous treatment of subnormal numbers, and handles mixed precision.

• 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

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

1: double $t \leftarrow [0, 999]$
2: double $r \leftarrow t/(t+1)$
3: **compute error:** r

(a) Microbenchmark 1

1: double $x \leftarrow [1.001, 2.0]$
2: double $y \leftarrow [1.001, 2.0]$
3: double $t \leftarrow x \times y$
4: double $r \leftarrow (t-1)/(t \times t - 1)$
5: **compute error:** r

(b) Microbenchmark 2

Fig. 1: Microbenchmarks

Table 1: Comparison of round-off error estimation tools

Feature	Gappa	Fluctuat	Rosa	FPTaylor
Basic FP operations/formats	✓	✓	✓	✓
Transcendental functions				✓
Relative error	✓	✓		✓
Uncertainties in inputs	✓	✓	✓	✓
Mixed precision	✓	✓		✓

cases, while incurring comparable runtimes. We also empirically verify that our overapproximations are within a factor of 3.5 of the corresponding underapproximations computed using a recent tool [7].
• 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.* It therefore promises to offer a similar safeguard for its future users.

2 Preliminary Comparison

We compare several existing popular tools for estimating round-off error with FPTaylor on microbenchmarks from Fig. 1. (These early overviews are provided to help better understand this problem domain.) Gappa [15] is a verification assistant based on interval arithmetic. Fluctuat [16] (commercial tool with a free academic version) statically analyzes C programs involving floating-point computations using abstract domains based on affine arithmetic [23]. Part of the Leon verification framework, Rosa [14] can compile programs with real numerical types into executable code where real types are replaced with floating-point types of sufficient precision to guarantee given error bounds. It combines affine arithmetic with SMT solvers to estimate floating-point errors. Z3 [42] and MathSAT 5 [8] are SMT solvers which support floating-point theory. In Table 1, we compare relevant features of FPTaylor with these tools (SMT solvers are not included in this table).

In our experiments involving SMT solvers, instead of comparing real mathematical and floating-point results, we compared low-precision and high-precision

Table 2: Experimental results for microbenchmarks (timeout is set to 30 minutes; entries in **bold font** represent the best results)

	Microbenchmark 1			Microbenchmark 2		
Tool	Range	Error	Time	Range	Error	Time
FPTaylor	**[0, 1.0]**	**1.663e-16**	1.2 s	**[0.2, 0.5]**	**1.532e-14**	1.4 s
Gappa	[0, 999]	5.695e-14	0.01 s	[0.0, 748.9]	1.044e-10	0.01 s
Gappa (hints)	**[0, 1.0]**	**1.663e-16**	1.4 s	[0.003, 66.02]	6.935e-14	3.2 s
Fluctuat	[0, 996]	5.667e-11	0.01 s	[0.0, 748.9]	1.162e-09	0.01 s
Fluctuat (div.)	**[0, 1.0]**	**1.664e-16**	28.5 s	[0.009, 26.08]	1.861e-12	20.0 s
Rosa	[0, 1.9]	2.217e-10	6.4 s	**[0.2, 0.5]**	2.636e-09	2.7 s
Z3	**[0, 1.0]**	Timeout	–	**[0.2, 0.5]**	Timeout	–
MathSAT 5	**[0, 1.0]**	Timeout	–	**[0.2, 0.5]**	Timeout	–

floating-point computations, mainly because current SMT solvers do not (as far as we know) support mixed real and floating-point reasoning. Moreover, SMT solvers were used to verify given error bounds since they cannot find the best error bounds directly. (In principle, a binary search technique can be used with an SMT solver for finding optimal error bounds [14].)

Table 2 reports the ranges of expression values obtained under the tool-specific abstraction (e.g., Gappa's abstraction estimates $t/(t+1)$ to be within [0,999]), estimated absolute errors, and time for each experiment. We also ran Gappa and Fluctuat with user-provided hints for subdividing the range of the input variables and for simplifying mathematical expressions. We used the following versions of tools: Gappa 1.1.2, Fluctuat 3.1071, Rosa from May 2014, Z3 4.3.2, and MathSAT5 5.2.11.

FPTaylor outperformed the other tools on Microbenchmark 2. Only Gappa and Fluctuat with manually provided subdivision hints were able to get the same results as FPTaylor for Microbenchmark 1. The range computation demonstrates a fundamental problem of interval arithmetic: it does not preserve dependencies between variables and thus significantly overapproximates results. Support of floating-point arithmetic in SMT solvers is still preliminary: they timed out on error estimation benchmarks (at 30 minutes timeout).

Rosa returned good range values since it uses an SMT solver internally for deriving tight ranges of all intermediate computations. Nevertheless, Rosa did not yield very good error estimation results for our nonlinear microbenchmarks since it represents rounding errors with affine forms—known for not handling nonlinear operators well.

3 Background

Floating-point Arithmetic. The IEEE 754 standard [28] concisely formalized in (e.g.) [22] 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 three general binary formats with sizes of 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 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 [22]

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

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 3 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 3 by multiplying all entries by 2, and (1) does not distinguish between rounding operators toward zero and infinities.

Table 3: Rounding to nearest operator parameters

Precision (bits)	ϵ	δ
single (32)	2^{-24}	2^{-150}
double (64)	2^{-53}	2^{-1075}
quad. (128)	2^{-113}	2^{-16495}

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 exactly rounded if the following equation holds for all floating-point values $x_1, \ldots, x_k$:

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

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

Combining (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 (3) can be improved. For instance, if op is $-$ or $+$ then $d = 0$ [22]. Also, if op is $\times$ and one of the arguments

is a nonnegative power of two 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 exactly rounded. For example, most implementations of floating-point transcendental functions are not exactly rounded but they yield results which are very close to exactly rounded results [25]. The technique introduced by Bingham et al. [3] can verify relative error bounds of hardware implementations of transcendental functions. So we can still use (3) to model transcendental functions but we need to increase values of ϵ and δ appropriately. There exist software libraries that exactly compute rounded values of transcendental functions [12, 17]. For such libraries, (3) can be applied without any changes.

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 [44]. Suppose $f(x_1, \ldots, x_k)$ is a twice continuously differentiable multivariate function on an open convex domain $D \subset \mathbb{R}^k$. For any fixed point $\mathbf{a} \in D$ (we use bold symbols to represent vectors) the following formula holds (for example, see Theorem 3.3.1 in [39])

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

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 $e_1, \ldots, e_k$:

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

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.

4 Symbolic Taylor Expansions

Given a function $f \colon \mathbb{R}^n \to \mathbb{R}$, 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., $\mathbf{x} \in I$. The domain I can be quite arbitrary. The only requirement is that it is bounded and the function f is twice differentiable in some open neighborhood of I. In FPTaylor, the domain I is defined with inequalities over input variables. In the benchmarks presented later, we have $a_i \le x_i \le b_i$ for all $i = 1, \ldots, n$. In this case, $I = [a_1, b_1] \times \ldots \times [a_n, b_n]$ is a product of intervals.

Let $\mathrm{fp}(f)\colon \mathbb{R}^n \to \mathbb{F}$ 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 (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 3 presented the following model of floating-point arithmetic (see (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 (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 (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 with the second order error term (5) 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 $|p_i| \leq \epsilon$ for $i = 1, \ldots, k$ and $|p_i| \leq \delta$ 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 3; 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 (7) can be estimated as follows:

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

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 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 (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:

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

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

4.1 Solving Optimization Problems

We compute error bounds using rigorous global optimization techniques [45]. 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 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. For instance, the optimal maximal floating-point value of the function $f(x) = 0.3$ is the smallest floating-point number r which is greater than 0.3. 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 inaccurate) 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 an inaccurate but fast global optimization technique if necessary.

The optimization problem (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 optimization solvers may not accept absolute values since they are not smooth functions. In addition, even if a solver accepts absolute values, they make the optimization problem considerably harder.

There is a naïve approach to simplify and solve this optimization problem. Find minimum (y_i) and maximum (z_i) values for each term $s_i(\mathbf{x}) = \frac{\partial \tilde{f}}{\partial e_i}(\mathbf{x}, \mathbf{0}, \mathbf{0})$ separately and then compute

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

This result can be inaccurate, but in many cases it is close to the optimal result as our experimental results demonstrate (see Sect. 5.2).

We also apply global optimization to compute a 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 (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.

4.2 Improved Rounding Model

The rounding model described by (1) and (3) is imprecise. For example, if we round a real number $x \in [8, 16]$ then (1) yields $\mathrm{rnd}(x) = x + xe$ with $|e| \leq \epsilon$. A more precise 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 $\left[2^n, 2^{n+1}\right]$ for integer n.

We define $p_2(x) = \max_{n \in \mathbb{Z}}\{2^n \mid 2^n < x\}$ and rewrite (1) and (3) as

$$\begin{aligned} \mathrm{rnd}(x) &= x + p_2(x)e + d, \\ op_{\mathrm{fp}}(x_1, \ldots, x_k) &= op(x_1, \ldots, x_k) + p_2\big(op(x_1, \ldots, x_k)\big)e + d \ . \end{aligned} \tag{12}$$

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 Sect. 5.2).

4.3 Formal Verification of FPTaylor Results in HOL Light

We formalized error estimation with the simplified optimization problem (11) in HOL Light [27]. 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 (1) and (12). We chose HOL Light as the tool for our formalization because it is the only proof assistant for which there exists a tool for formal verification of nonlinear inequalities (including inequalities with transcendental functions) [53]. Verification of nonlinear inequalities is necessary since the validity of results of global optimization procedures can be proved with 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 (1) and (12). We also implemented a comprehensive formalization of floating-point arithmetic in HOL Light (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 (1) and (12).

The formalization of FPTaylor helped us to find a subtle bug in our implementation. We use an external tool for algebraic simplifications of internal expressions in FPTaylor (see Sect. 5.1 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.

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 absolute values and the discontinuous function $p_2(x)$ defined in Sect. 4.2. We are working on improvements of the inequality verification tool which will include these functions. 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 [9].

5 Implementation and Evaluation

5.1 Implementation

We implemented a prototype tool called FPTaylor for estimating round-off errors in floating-point computations based on our method described in Sect. 4. The tool implements several additional features we did not describe, such as estimation of relative errors and support for transcendental functions and mixed precision floating-point computations.

FPTaylor is implemented in OCaml and uses several third-party tools and libraries. An interval arithmetic library [1] is used for rigorous estimations of floating-point constants and second order error terms in Taylor expansions. Internally, FPTaylor implements a very simple 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 Sect. 4.2. 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 [29] that implements various global optimization algorithms. 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. Z3 SMT solver [42] can also be used as an optimization backend by employing a simple binary search algorithm similar to the one described in related work [14]. Z3-based optimization supports any inequality constraints but 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 [31], GlobSol [30], and OpenOpt [46]. We rely on Maxima computer algebra system [37] for performing symbolic simplifications. Using Maxima is optional but it can significantly improve performance of optimization tools by simplifying symbolic expressions beforehand.

```
1: Variables
2:  float64 x in [1.001, 2.0],
3:  float64 y in [1.001, 2.0];
4: Definitions
5:  t rnd64= x * y;
6: // Constraints
7: //  x + y <= 2;
8: Expressions
9:  r rnd64= (t-1)/(t*t-1);
```

Fig. 2: FPTaylor input file example

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 Fig. 2 constraints are not used and commented out), and expressions. FPTaylor analyses 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 [15]). See the FPTaylor user manual distributed with the tool for all usage details.

5.2 Experimental Results

We compared FPTaylor with Gappa (version 1.1.2) [15], the Rosa real compiler (version from May 2014) [14], and Fluctuat (version 3.1071) [16] (see Sect. 6 for more information on these tools). We tested our tool on all benchmarks from the Rosa paper [14] and on three simple benchmarks with transcendental functions.[3] We also tried SMT tools which support floating-point reasoning [8, 42] but they were not able to produce any results even on simple examples in a reasonable time (we ran them with a 30-minute timeout).

Table 4 presents our experimental results. In the table, column *FPTaylor(a)* shows results computed using the simplified optimization problem (11), column *FPTaylor(b)* using the full optimization problem (10) and the improved rounding model (12). Columns *Gappa (hints)* and *Fluctuat (subdivisions)* present results of Gappa and Fluctuat with manually provided subdivision hints. More precisely, in these experiments Gappa and Fluctuat were 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).

Benchmarks *sine*, *sqroot*, and *sineOrder3* are different polynomial approximations of sine and square root. Benchmarks *carbonGas*, *rigidBody1*, *rigidBody2*, *doppler1*, *doppler2*, and *doppler3* are nonlinear expressions used in physics. Benchmarks *verhulst* and *predatorPrey* are from biological modeling. Benchmarks *turbine1*, *turbine2*, *turbine3*, and *jetEngine* are from control theory. Benchmark *logExp* is from Gappa++ paper [33] 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 [56], 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 Next Generation Air Transportation System (NextGen) called AutoResolver, whose task is to provide separation assurance for airplanes [20].

Table 5 contains additional information about benchmarks. Columns *Vars*, *Ops*, and *Trans* show the number of variables, the total number of floating-point operations, and the total number of transcendental operations in each benchmark. The column *FPTaylor(b)* repeats results of FPTaylor from Table 4. The column *s3fp* shows lower bounds of errors estimated with the underapproximation tool s3fp [7]. The column *Ratio* gives ratios of overapproximations computed with FPTaylor(b) and underapproximations computed with s3fp.

[3] Our benchmarks are available at `https://github.com/soarlab/FPTaylor`

Table 4: Experimental results for absolute round-off error bounds (**bold font** marks the best results for each benchmark; *italic font* marks pessimistic results)

Benchmark	Gappa	Gappa (hints)	Fluctuat	Fluctuat (subdiv.)	Rosa	FPT.(a)	FPT.(b)
Univariate polynomial approximations							
sine	*1.46*	*5.17e-09*	7.97e-16	6.86e-16	9.56e-16	6.71e-16	**4.43e-16**
sqroot	5.71e-16	**5.37e-16**	6.84e-16	6.84e-16	8.41e-16	7.87e-16	5.78e-16
sineOrder3	8.89e-16	**6.50e-16**	1.16e-15	1.03e-15	1.11e-15	9.96e-16	7.95e-16
Rational functions with 1, 2, and 3 variables							
carbonGas	2.62e-08	**6.00e-09**	4.52e-08	8.88e-09	4.64e-08	1.25e-08	9.99e-09
verhulst	5.41e-16	2.84e-16	5.52e-16	4.78e-16	6.82e-16	3.50e-16	**2.50e-16**
predPrey	2.44e-16	1.66e-16	2.50e-16	2.35e-16	2.94e-16	1.87e-16	**1.59e-16**
rigidBody1	3.22e-13	**2.95e-13**	3.22e-13	3.22e-13	5.08e-13	3.87e-13	**2.95e-13**
rigidBody2	3.65e-11	**3.61e-11**	3.65e-11	3.65e-11	6.48e-11	5.24e-11	**3.61e-11**
doppler1	2.03e-13	1.61e-13	3.91e-13	1.40e-13	4.92e-13	1.57e-13	**1.35e-13**
doppler2	3.92e-13	2.86e-13	9.76e-13	2.59e-13	1.29e-12	2.87e-13	**2.44e-13**
doppler3	1.08e-13	8.70e-14	1.57e-13	7.63e-14	2.03e-13	8.16e-14	**6.97e-14**
turbine1	9.51e-14	2.63e-14	9.21e-14	8.31e-14	1.25e-13	2.50e-14	**1.86e-14**
turbine2	1.38e-13	3.54e-14	1.30e-13	1.10e-13	1.76e-13	3.34e-14	**2.15e-14**
turbine3	*39.91*	*0.35*	6.99e-14	5.94e-14	8.50e-14	1.80e-14	**1.07e-14**
jetEngine	*8.24e+06*	*4426.37*	*4.08e-08*	1.82e-11	*1.62e-08*	1.49e-11	**1.03e-11**
Transcendental functions with 1 and 4 variables							
logExp	–	–	–	–	–	1.71e-15	**1.53e-15**
sphere	–	–	–	–	–	1.29e-14	**8.08e-15**
azimuth	–	–	–	–	–	1.41e-14	**8.78e-15**

For all these 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. All results are given for double precision floating-point numbers and we ran Gappa, Fluctuat, and Rosa with standard settings. We used a simple branch and bound optimization method in FPTaylor since it works better than a Z3-based optimization on most benchmarks. For transcendental functions, we used increased values of ϵ and δ: $\epsilon = 1.5 \cdot 2^{-53}$ and $\delta = 1.5 \cdot 2^{-1075}$.

Gappa with user provided hints computed best results in 5 out of 15 benchmarks (we do not count last 3 benchmarks with transcendental functions). FPTaylor computed best results in 12 benchmarks.[4] Gappa without hints was able to find a better result than FPTaylor only in the *sqroot* benchmark. On the

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

Table 5: Additional benchmark information

Benchmark	Vars	Ops	Trans	FPTaylor(b)	s3fp	Ratio
Univariate polynomial approximations						
sine	1	18	0	4.43e-16	2.85e-16	1.6
sqroot	1	14	0	5.78e-16	4.57e-16	1.3
sineOrder3	1	5	0	7.95e-16	3.84e-16	2.1
Rational functions with 1, 2, and 3 variables						
carbonGas	1	11	0	9.99e-09	4.11e-09	2.4
verhulst	1	4	0	2.50e-16	2.40e-16	1.1
predPrey	1	7	0	1.59e-16	1.47e-16	1.1
rigidBody1	3	7	0	2.95e-13	2.47e-13	1.2
rigidBody2	3	14	0	3.61e-11	2.88e-11	1.3
doppler1	3	8	0	1.35e-13	8.01e-14	1.7
doppler2	3	8	0	2.44e-13	1.54e-13	1.6
doppler3	3	8	0	6.97e-14	4.54e-14	1.5
turbine1	3	14	0	1.86e-14	1.01e-14	1.8
turbine2	3	10	0	2.15e-14	1.20e-14	1.8
turbine3	3	14	0	1.07e-14	5.04e-15	2.1
jetEngine	2	48	0	1.03e-11	6.37e-12	1.6
Transcendental functions with 1 and 4 variables						
logExp	1	3	2	1.53e-15	1.19e-15	1.3
sphere	4	5	2	8.08e-15	5.05e-15	1.6
azimuth	4	14	7	8.78e-15	2.53e-15	3.5

other hand, in several benchmarks (*sine*, *jetEngine*, and *turbine3*), Gappa (even with hints) computed very pessimistic results. Rosa consistently computed decent error bounds, with one exception being *jetEngine*. FPTaylor outperformed Rosa on all benchmarks even with the simplified rounding model and optimization problem. Fluctuat results without subdivisions are similar to Rosa's results. Fluctuat results with subdivisions are good but they were obtained with carefully chosen subdivisions. FPTaylor with the improved rounding model outperformed Fluctuat with subdivisions on all but one benchmark (*carbonGas*). Only FPTaylor and Fluctuat with subdivisions found good error bounds for the *jetEngine* benchmark.

FPTaylor yields best results with the full optimization problem (10) and with the improved rounding model (12). But these results are at most 2 times better (and even less in most cases) than results computed with the simple rounding model (3) and the simplified optimization problem (11). The main advantage of the simplified optimization problem is that it can be applied to more complex problems. Finally, we compared results of FPTaylor with lower bounds of errors

estimated with a state-of-the-art underapproximation tool s3fp [7]. All FPTaylor results are only 1.1–2.4 times worse than the estimated lower bounds for polynomial and rational benchmarks and 1.3–3.5 times worse for transcendental tests.

Table 6 compares performance results of different tools on first 15 benchmarks (the results for the *jetEngine* benchmark and the total time for all 15 benchmarks are shown; FPTaylor takes about 33 seconds on three transcendental benchmarks). Gappa and Fluctuat (without hints and subdivisions) are considerably faster than both Rosa and FPTaylor. But Gappa often fails on nonlinear examples as Table 4 demonstrated. Fluctuat without subdivisions is also not as good as FPTaylor. All other tools (including FPTaylor) have roughly the same performance. Rosa is slower than FPTaylor because it relies on an inefficient optimization algorithm implemented with Z3.

Table 6: Performance results on an Intel Core i7 2.8GHz machine (in seconds)

Tool	jetEng.	Total
Gappa	0.02	0.38
Gappa(hints)	21.47	80.27
Fluctuat	0.01	0.75
Fluct.(div.)	23.00	228.36
Rosa	129.63	205.14
FPTaylor(a)	14.73	86.92
FPTaylor(b)	16.63	102.23

We also formally verified all results in the column *FPTaylor(a)* of Table 4. For all these results, corresponding HOL Light theorems were automatically produced using our formalization of FPTaylor described in Sect. 4.3. The total verification time of all results without the *azimuth* benchmark was 48 minutes on an Intel Core i7 2.8GHz machine. Verification of the *azimuth* benchmark took 261 minutes. Such performance figures match up with the state of the art, considering that even results pertaining to basic arithmetic operations must be formally derived from primitive definitions.

6 Related Work

Taylor Series. Method based on Taylor series have a rich history in floating-point reasoning, including algorithms for constructing symbolic Taylor series expansions for round-off errors [40, 55, 19, 43], 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 [51]. Note that in this case round-off errors are not computed directly and cannot be extracted from computed enclosures without large overestimations.

Abstract Interpretation. Abstract interpretation [11] is widely used for analysis of floating-point computations. Abstract domains for floating-point values include intervals [41], affine forms [54], and general polyhedra [6]. There exist different tools based on these abstract domains. Gappa [15] is a tool for checking different aspects of floating-point programs, and is used in the Frama-C verifier [18]. Gappa works with interval abstractions of floating-point numbers and

applies rewriting rules for improving computed results. Gappa++ [33] is an improvement of Gappa that extends it with affine arithmetic [54]. It also provides definitions and rules for some transcendental functions. Gappa++ is currently not supported and does not run on modern operating systems. SmartFloat [13] 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 [16] 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 [23]. Astrée [10] 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 [34]. Based on this work, a tool called RangeLab is implemented [36] and a technique for improving accuracy of floating-point computations is presented [35]. Ponsini et al. [49] 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.

The work closest to ours is Rosa [14] 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.

SMT. 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 [5]. Recently, a preliminary standard for floating-point arithmetic in SMT solvers was developed [52]. Z3 [42] and MathSAT 5 [8] SMT solvers partially support this standard. There exist several other tools which use SMT solvers for reasoning about floating-point numbers. FPhile [47] 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. [32] translate a given floating-point formula into a corresponding formula for real numbers with appropriately defined rounding operators. Ariadne [2] relies on SMT solving for detecting floating-point exceptions. Haller et al. [24] 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 [14]. Since existing SMT solvers do not directly support

mixed real/floating-point reasoning, 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).

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 [38, 26]. Boldo et al. [4] formalized a non-trivial 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 [9].

7 Conclusions and Future Work

We presented a new method to estimate round-off errors of floating-point computations called Symbolic Taylor Expansions. We support our work through rigorous formal proofs, and also present a tool FPTaylor that implements our method. FPTaylor is the only tool we know 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 [18, 50]. Conditional expressions can be verified in FPTaylor in the same way as it is done in Rosa [14] (see Appendix B 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 [48] which improves the accuracy of numerical programs. Herbie relies on testing for round-off error estimations. FPTaylor can provide strong guarantees for results produced by Herbie.

We also plan to improve the performance of FPTaylor by parallelizing its global optimization algorithms, thus paving the way to analyze larger problems.

Ideas presented in this paper can be directly incorporated into existing tools. For instance, an implementation similar to Gappa++ [33] can be achieved by incorporating our error estimation method inside Gappa [15]; the Rosa compiler [14] can be easily extended with our technique.

Acknowledgments. We would like to thank Nelson Beebe, Wei-Fan Chiang, John Harrison, and Madan Musuvathi for their feedback and encouragement. This work is supported in part by NSF CCF 1421726.

References

1. Alliot, J.M., Durand, N., Gianazza, D., Gotteland, J.B.: Implementing an interval computation library for OCaml on x86/amd64 architectures (short paper). In: ICFP 2012. ACM (2012)

2. Barr, E.T., Vo, T., Le, V., Su, Z.: Automatic Detection of Floating-point Exceptions. In: POPL 2013. pp. 549–560. POPL '13, ACM, New York, NY, USA (2013)
3. Bingham, J., Leslie-Hurd, J.: Verifying Relative Error Bounds Using Symbolic Simulation. In: Biere, A., Bloem, R. (eds.) CAV 2014, LNCS, vol. 8559, pp. 277–292. Springer International Publishing (2014)
4. Boldo, S., Clément, F., Filliâtre, J.C., Mayero, M., Melquiond, G., Weis, P.: Wave Equation Numerical Resolution: A Comprehensive Mechanized Proof of a C Program. Journal of Automated Reasoning 50(4), 423–456 (2013)
5. Brillout, A., Kroening, D., Wahl, T.: Mixed abstractions for floating-point arithmetic. In: FMCAD 2009. pp. 69–76 (2009)
6. Chen, L., Miné, A., Cousot, P.: A Sound Floating-Point Polyhedra Abstract Domain. In: Ramalingam, G. (ed.) APLAS 2008, LNCS, vol. 5356, pp. 3–18. Springer Berlin Heidelberg (2008)
7. Chiang, W.F., Gopalakrishnan, G., Rakamarić, Z., Solovyev, A.: Efficient Search for Inputs Causing High Floating-point Errors. In: PPoPP 2014. pp. 43–52. PPoPP '14, ACM, New York, NY, USA (2014)
8. Cimatti, A., Griggio, A., Schaafsma, B., Sebastiani, R.: The MathSAT5 SMT Solver. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 93–107 (2013)
9. The Coq Proof Assistant. `http://coq.inria.fr/`
10. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTRÉE Analyser. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 21–30. Springer Berlin Heidelberg (2005)
11. Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: POPL 1977. pp. 238–252. POPL '77, ACM, New York, NY, USA (1977)
12. Daramy, C., Defour, D., de Dinechin, F., Muller, J.M.: CR-LIBM: a correctly rounded elementary function library. Proc. SPIE 5205, 458–464 (2003)
13. Darulova, E., Kuncak, V.: Trustworthy Numerical Computation in Scala. In: OOPSLA 2011. pp. 325–344. OOPSLA '11, ACM, New York, NY, USA (2011)
14. Darulova, E., Kuncak, V.: Sound Compilation of Reals. In: POPL 2014. pp. 235–248. POPL '14, ACM, New York, NY, USA (2014)
15. Daumas, M., Melquiond, G.: Certification of Bounds on Expressions Involving Rounded Operators. ACM Trans. Math. Softw. 37(1), 2:1–2:20 (2010)
16. Delmas, D., Goubault, E., Putot, S., Souyris, J., Tekkal, K., Védrine, F.: Towards an Industrial Use of FLUCTUAT on Safety-Critical Avionics Software. In: Alpuente, M., Cook, B., Joubert, C. (eds.) FMICS 2009, LNCS, vol. 5825, pp. 53–69. Springer Berlin Heidelberg (2009)
17. Fousse, L., Hanrot, G., Lefèvre, V., Pélissier, P., Zimmermann, P.: MPFR: A Multiple-precision Binary Floating-point Library with Correct Rounding. ACM Trans. Math. Softw. 33(2) (2007)
18. Frama-C Software Analyzers. `http://frama-c.com/`
19. Gáti, A.: Miller Analyzer for Matlab: A Matlab Package for Automatic Roundoff Analysis. Computing and Informatics 31(4), 713– (2012)
20. Giannakopoulou, D., Howar, F., Isberner, M., Lauderdale, T., Rakamarić, Z., Raman, V.: Taming Test Inputs for Separation Assurance. In: ASE 2014. pp. 373–384. ASE '14, ACM, New York, NY, USA (2014)
21. Goodloe, A., Muñoz, C., Kirchner, F., Correnson, L.: Verification of Numerical Programs: From Real Numbers to Floating Point Numbers. In: Brat, G., Rungta, N., Venet, A. (eds.) NFM 2013. LNCS, vol. 7871, pp. 441–446. Springer, Moffett Field, CA (2013)

22. Goualard, F.: How Do You Compute the Midpoint of an Interval? ACM Trans. Math. Softw. 40(2), 11:1–11:25 (2014)
23. Goubault, E., Putot, S.: Static Analysis of Finite Precision Computations. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011, LNCS, vol. 6538, pp. 232–247. Springer Berlin Heidelberg (2011)
24. Haller, L., Griggio, A., Brain, M., Kroening, D.: Deciding floating-point logic with systematic abstraction. In: FMCAD 2012. pp. 131–140 (2012)
25. Harrison, J.: Formal Verification of Floating Point Trigonometric Functions. In: Hunt, W.A., Johnson, S.D. (eds.) FMCAD 2000, LNCS, vol. 1954, pp. 254–270. Springer Berlin Heidelberg (2000)
26. Harrison, J.: Floating-Point Verification Using Theorem Proving. In: Bernardo, M., Cimatti, A. (eds.) SFM 2006, LNCS, vol. 3965, pp. 211–242. Springer Berlin Heidelberg (2006)
27. Harrison, J.: HOL Light: An Overview. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009, LNCS, vol. 5674, pp. 60–66. Springer Berlin Heidelberg (2009)
28. IEEE Standard for Floating-point Arithmetic. IEEE Std 754-2008 pp. 1–70 (2008)
29. Johnson, S.G.: The NLopt nonlinear-optimization package. `http://ab-initio.mit.edu/nlopt`
30. Kearfott, R.B.: GlobSol User Guide. Optimization Methods Software 24(4-5), 687–708 (2009)
31. Lebbah, Y.: ICOS: A Branch and Bound Based Solver for Rigorous Global Optimization. Optimization Methods Software 24(4-5), 709–726 (2009)
32. Leeser, M., Mukherjee, S., Ramachandran, J., Wahl, T.: Make it real: Effective floating-point reasoning via exact arithmetic. In: DATE 2014. pp. 1–4 (2014)
33. Linderman, M.D., Ho, M., Dill, D.L., Meng, T.H., Nolan, G.P.: Towards Program Optimization Through Automated Analysis of Numerical Precision. In: CGO 2010. pp. 230–237. CGO '10, ACM, New York, NY, USA (2010)
34. Martel, M.: Semantics of roundoff error propagation in finite precision calculations. Higher-Order and Symbolic Computation 19(1), 7–30 (2006)
35. Martel, M.: Program Transformation for Numerical Precision. In: PEPM 2009. pp. 101–110. PEPM '09, ACM, New York, NY, USA (2009)
36. Martel, M.: RangeLab: A Static-Analyzer to Bound the Accuracy of Finite-Precision Computations. In: SYNASC 2011. pp. 118–122. SYNASC '11, IEEE Computer Society, Washington, DC, USA (2011)
37. Maxima: Maxima, a Computer Algebra System. Version 5.30.0 (2013), `http://maxima.sourceforge.net/`
38. Melquiond, G.: Floating-point arithmetic in the Coq system. Information and Computation 216(0), 14–23 (2012)
39. Mikusinski, P., Taylor, M.: An Introduction to Multivariable Analysis from Vector to Manifold. Birkhäuser Boston (2002)
40. Miller, W.: Software for Roundoff Analysis. ACM Trans. Math. Softw. 1(2), 108–128 (1975)
41. Moore, R.: Interval analysis. Prentice-Hall series in automatic computation, Prentice-Hall (1966)
42. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C., Rehof, J. (eds.) TACAS 2008, LNCS, vol. 4963, pp. 337–340. Springer Berlin Heidelberg (2008)
43. Mutrie, M.P.W., Bartels, R.H., Char, B.W.: An Approach for Floating-point Error Analysis Using Computer Algebra. In: ISSAC 1992. pp. 284–293. ISSAC '92, ACM, New York, NY, USA (1992)

44. Neumaier, A.: Taylor Forms - Use and Limits. Reliable Computing 2003, 9–43 (2002)
45. Neumaier, A.: Complete search in continuous global optimization and constraint satisfaction. Acta Numerica 13, 271–369 (2004)
46. OpenOpt: universal numerical optimization package. http://openopt.org
47. Paganelli, G., Ahrendt, W.: Verifying (In-)Stability in Floating-Point Programs by Increasing Precision, Using SMT Solving. In: SYNASC 2013. pp. 209–216 (2013)
48. Panchekha, P., Sanchez-Stern, A., Wilcox, J.R., Tatlock, Z.: Automatically Improving Accuracy for Floating Point Expressions. In: PLDI 2015. PLDI '15, ACM (2015)
49. Ponsini, O., Michel, C., Rueher, M.: Verifying floating-point programs with constraint programming and abstract interpretation techniques. Automated Software Engineering pp. 1–27 (2014)
50. Rakamarić, Z., Emmi, M.: SMACK: Decoupling Source Language Details from Verifier Implementations. In: Biere, A., Bloem, R. (eds.) CAV 2014, LNCS, vol. 8559, pp. 106–113. Springer International Publishing (2014)
51. Revol, N., Makino, K., Berz, M.: Taylor models and floating-point arithmetic: proof that arithmetic operations are validated in COSY. The Journal of Logic and Algebraic Programming 64(1), 135–154 (2005)
52. Rümmer, P., Wahl, T.: An SMT-LIB Theory of Binary Floating-Point Arithmetic. In: SMT Workshop 2010 (2010)
53. Solovyev, A., Hales, T.: Formal verification of nonlinear inequalities with taylor interval approximations. In: Brat, G., Rungta, N., Venet, A. (eds.) NFM 2013, LNCS, vol. 7871, pp. 383–397. Springer Berlin Heidelberg (2013)
54. Stolfi, J., de Figueiredo, L.: An Introduction to Affine Arithmetic. TEMA Tend. Mat. Apl. Comput. 4(3), 297–312 (2003)
55. Stoutemyer, D.R.: Automatic Error Analysis Using Computer Algebraic Manipulation. ACM Trans. Math. Softw. 3(1), 26–43 (1977)
56. NASA World Wind Java SDK. http://worldwind.arc.nasa.gov/java/

A Formal Derivation of Taylor Forms

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 (resp., constants, variable) are replaced with floating-point operations (resp., constants, variables). Inductive rules which define $\mathrm{fp}(f)$ are the following:

$$\begin{aligned} \mathrm{fp}(x) &= x, \ x \text{ is a floating-point variable or constant} \\ \mathrm{fp}(x) &= \mathrm{rnd}(x), \ x \text{ is a real variable or constant} \\ \mathrm{fp}\big(op(f_1, \ldots, f_r)\big) &= \mathrm{rnd}\big(op(\mathrm{fp}(f_1), \ldots, \mathrm{fp}(f_r))\big), \\ &\qquad \text{where } op \text{ is } +, -, \times, /, \surd, \mathrm{fma} \end{aligned} \tag{13}$$

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 (3) with slightly larger bounds for e and d.

Use (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:

$$\forall \mathbf{x} \in I, \, \exists \mathbf{e} \in D_\epsilon, \mathbf{d} \in D_\delta, \ \mathrm{fp}(f)(\mathbf{x}) = \tilde{f}(\mathbf{x}, \mathbf{e}, \mathbf{d}) \ , \tag{14}$$

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

We start by describing the main data structure on which our derivation rules operate. We have the following Taylor expansion of $\tilde{f}(\mathbf{x}, \mathbf{e}, \mathbf{d})$:

$$\tilde{f}(\mathbf{x}, \mathbf{e}, \mathbf{d}) = f(\mathbf{x}) + \sum_{i=1}^{k} s_i(\mathbf{x}) e_i + R_2(\mathbf{x}, \mathbf{e}, \mathbf{d}) \ . \tag{15}$$

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 later.

Formula (15) 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 $|R_2(\mathbf{x}, \mathbf{e}, \mathbf{d})| \le M_2$ for all $\mathbf{x} \in I$, $\mathbf{e} \in D_\epsilon$, and $\mathbf{d} \in D_\delta$. Define $s_{k+1}(\mathbf{x}) = \frac{M_2}{\epsilon}$. Then the following formula holds:

$$\forall \mathbf{x} \in I, \mathbf{e} \in D_\epsilon, \mathbf{d} \in D_\delta, \, \exists e_{k+1}, |e_{k+1}| \le \epsilon \wedge R_2(\mathbf{x}, \mathbf{e}, \mathbf{d}) = s_{k+1}(\mathbf{x}) e_{k+1} \ . \tag{16}$$

This formula follows from the simple fact that $\left| \frac{R_2(\mathbf{x}, \mathbf{e}, \mathbf{d})}{s_{k+1}(\mathbf{x})} \right| \le \frac{M_2}{s_{k+1}(\mathbf{x})} = \epsilon$. Next, we substitute (15) into (14), find e_{k+1} from (16), and replace $R_2(\mathbf{x}, \mathbf{e}, \mathbf{d})$ with

$s_{k+1}(\mathbf{x})e_{k+1}$. We get the following identity:

$$\forall \mathbf{x} \in I,\ \exists e_1, \ldots, e_{k+1},\ |e_i| \le \epsilon \wedge \mathrm{fp}(f)(\mathbf{x}) = f(\mathbf{x}) + \sum_{i=1}^{k+1} s_i(\mathbf{x})e_i \ . \tag{17}$$

The identity (17) 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 $\langle f, \mathsf{s}\rangle$ be a pair of a symbolic expression f (we do not distinguish between a function f and its symbolic expression) and a list $\mathsf{s} = [s_1; \ldots; s_r]$ of symbolic expressions s_i. We call the pair $\langle f, \mathsf{s}\rangle$ a *Taylor form*. We also use capital letters to denote Taylor forms, e.g., $F = \langle f, \mathsf{s}\rangle$. For any function $h(\mathbf{x})$, we write $h \sim \langle f, \mathsf{s}\rangle$ if and only if

$$\forall \mathbf{x} \in I,\ \exists \mathbf{e} \in D_\epsilon,\ h(\mathbf{x}) = f(\mathbf{x}) + \sum_{i=1}^{r} s_i(\mathbf{x})e_i \ . \tag{18}$$

If $h \sim \langle f, \mathsf{s}\rangle$ we say that $\langle f, \mathsf{s}\rangle$ corresponds to h. We are interested in Taylor forms $\langle f, \mathsf{s}\rangle$ corresponding to $\mathrm{fp}(f)$. Note that the expression on the right hand side of (18) 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 3 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 $f(x) = 0.1 \times x$ and $x \in [1, 3]$. From (13) we get $\mathrm{fp}(f)(x) = \mathrm{rnd}\big(\mathrm{rnd}(0.1) \times \mathrm{rnd}(x)\big)$. (Note that x is a real variable so it must be rounded.) Take the rules $\mathsf{CONST}_{\mathsf{RND}}$ and $\mathsf{VAR}_{\mathsf{RND}}$ and apply them to corresponding subexpressions of $\mathrm{fp}(f)$:

$$\mathsf{CONST}_{\mathsf{RND}}\big(\mathrm{rnd}(0.1)\big) = \langle 0.1, [f_{\mathrm{err}}(0.1)]\rangle = \langle 0.1, [0.1]\rangle \ ,$$
$$\mathsf{VAR}_{\mathsf{RND}}\big(\mathrm{rnd}(x)\big) = \langle x, [f_{\mathrm{err}}(x)]\rangle = \langle x, [x]\rangle \ .$$

Here, the function $f_{\mathrm{err}} : \mathbb{R} \to \mathbb{R}$ estimates the relative 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 $\mathsf{CONST}_{\mathsf{RND}}$ (resp., $\mathsf{VAR}_{\mathsf{RND}}$) may yield better results than application of rules CONST (resp., VAR) and RND in sequence. Now, we apply the rule MUL to the Taylor forms of $\mathrm{rnd}(0.1)$ and $\mathrm{rnd}(x)$:

$$\mathsf{MUL}\big(\langle 0.1, [0.1]\rangle, \langle x, [x]\rangle\big) = \langle 0.1x, [0.1 \times x] \ @\ [x \times 0.1] \ @\ [\epsilon M_2]\rangle \ ,$$

where $M_2 \ge \max_{x \in [1,3]} |x \times 0.1|$. This upper bound can be computed with a simple method like interval arithmetic (it is multiplied by a small number ϵ

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

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

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

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

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

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

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

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

Fig. 3: Derivation rules of Taylor forms

in the resulting form and hence the value of M_2 may be conservative). In our example, we take $M_2 = 1$ and get

$$F = \mathsf{MUL}\big(\langle 0.1, [0.1]\rangle, \langle x, [x]\rangle\big) = \langle 0.1x, [0.1x; 0.1x; \epsilon]\rangle \ .$$

Finally, we apply the rule RND to F:

$$\begin{aligned} G &= \mathsf{RND}\big(\langle 0.1x, [0.1x; 0.1x; \epsilon]\rangle\big) \\ &= \big\langle 0.1x, [0.1x] \mathbin{@} [0.1x; 0.1x; \epsilon] \mathbin{@} [\epsilon M_2 + \tfrac{\delta}{\epsilon}]\big\rangle \ , \end{aligned}$$

where $M_2 \geq \max_{x\in[1,3]}(|0.1x| + |0.1x| + |\epsilon|)$. Again, the upper bound M_2 can be computed with a simple method. We take $M_2 = 1$ and get the final Taylor form corresponding to our example:

$$G = \big\langle 0.1x, [0.1x; 0.1x; 0.1x; \epsilon; \epsilon + \tfrac{\delta}{\epsilon}]\big\rangle \ .$$

The main property of rules in Fig. 3 is given by the following theorem.

Theorem 1. *Suppose* RULE *is one of the derivation rules in Fig. 3 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 give a proof for the multiplication rule MUL. Proofs for other rules can be found at the end of this appendix.

Suppose that $h_1 \sim \langle f, \mathsf{s}\rangle$ and $h_2 \sim \langle g, \mathsf{t}\rangle$. Fix $\mathbf{x} \in I$, then by (18) we have

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

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

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

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 $M_2 \geq \max_{\mathbf{x}\in I}\left(\sum_{i=1,j=1}^{t,r}|s_i(\mathbf{x})t_j(\mathbf{x})|\right)$. We have $M_2\epsilon^2 \geq |R_2(\mathbf{x})|$ for all $\mathbf{x} \in I$. Hence, for any $\mathbf{x}$ we can find $w = w(\mathbf{x})$, $|w| \leq \epsilon$, 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 $h_1 h_2 \sim \mathsf{MUL}\big(\langle f, \mathsf{s}\rangle, \langle g, \mathsf{t}\rangle\big)$.

The next theorem summarizes the main result of this section.

Theorem 2. *For any input function* $\mathrm{fp}(f)$*, the Taylor form constructed with the rules described in Fig. 3 corresponds to the function* $\mathrm{fp}(f)$*. That is, if the constructed Taylor form is* $\langle f, \mathsf{s}\rangle$ *then* $\mathrm{fp}(f) \sim \langle f, \mathsf{s}\rangle$ *and the property* (18) *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. The proof can be found at the end of this appendix. 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 $\mathrm{fp}(f) \sim \langle f, \mathsf{s}\rangle = F$ and $\mathrm{fp}(g) \sim \langle g, \mathsf{t}\rangle = G$. Theorem 1 implies $h = \mathrm{fp}(f) \times \mathrm{fp}(g) \sim \mathsf{MUL}\big(F, G\big) = H$ and $\mathrm{rnd}(h) \sim \mathsf{RND}\big(H\big)$. Therefore $\mathrm{fp}(f \times g) \sim \mathsf{RND}\big(\mathsf{MUL}\big(F, G\big)\big)$ and the result follows by induction.

Implementation Details. In our presentation above, 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 $\mathrm{fp}(f)(x, y) = \mathrm{rnd}\big(\mathrm{rnd}(xy) - \mathrm{rnd}(xy)\big)$ and if we compute its Taylor form with rules from Figure 3, 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: $[f] \mathbin{@} \mathsf{s} \mathbin{@} [\epsilon M_2 + \frac{\delta}{\epsilon}]$. This list needs to be replaced with the list $[f e_{a_f}] \mathbin{@} \mathsf{s} \mathbin{@} [(\epsilon M_2 + \frac{\delta}{\epsilon}) e_a]$. 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.

We implemented several other improvements of the derivation rules for obtaining better error bounds: (1) Whenever we multiply an expression by a power of 2, we do not need to round the result; (2) If we divide by a power of 2, we only need to consider potential subnormal errors (given by the term $\frac{\delta}{\epsilon}$); (3) There are no subnormal errors for rounding after addition or subtraction (i.e., we do not need to add the term $\frac{\delta}{\epsilon}$ in the RND rule).

Proofs for Rules. We prove the property

$$f_1 \sim F_1, \ldots, f_k \sim F_k \implies op(f_1, \ldots, f_k) \sim \mathsf{RULE}\big(F_1, \ldots, F_k\big) \ .$$

for rules defined in Figure 3.

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 $\mathrm{rnd}(c)$ (that is, 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 (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 (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 $\langle f, \mathsf{s} \rangle$, the rounding rule RND returns another Taylor form which corresponds to a rounding operator applied to the expression defined by $\langle f, \mathsf{s} \rangle$. We need to prove that $h \sim \langle f, \mathsf{s} \rangle$ implies $\mathrm{rnd}(h) \sim \mathsf{RND}\big(\langle f, \mathsf{s} \rangle\big)$ (here, $\mathrm{rnd}(h)$ is a function defined by $\mathrm{rnd}(h)(\mathbf{x}) = \mathrm{rnd}(h(\mathbf{x}))$). Fix $\mathbf{x}$. The assumption $h \sim \langle f, \mathsf{s} \rangle$ 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 (18)). Equation (1) allows us to find e_{k+1} and d with $|e_{k+1}| \leq \epsilon$, $|d| \leq \delta$ 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 $M_2 \geq \max_{\mathbf{x} \in I} \left(\sum_{i=1}^{k} |s_i(\mathbf{x})| \right)$. 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 $|e_{k+2}| \leq \epsilon$. We can write

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

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

SUB (ADD). Consider the subtraction rule (the addition rule is analogous). Suppose $h_1 \sim \langle f, \mathsf{s} \rangle$ and $h_2 \sim \langle g, \mathsf{t} \rangle$. Show that $h_1 - h_2 \sim \mathsf{SUB}\big(\langle f, \mathsf{s} \rangle, \langle g, \mathsf{t} \rangle\big)$. We

can find $e_1, \ldots, e_k$ and $v_1, \ldots, v_r$, $|e_i| \leq \epsilon$, $|v_j| \leq \epsilon$, such that

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

Hence the result follows.

MUL. This result is proved in Theorem 1.

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 $|\theta_k| \leq |e_k| \leq \epsilon$. 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 $|\theta_k| \leq |e_k| \leq \epsilon$. 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:

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

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

Table 7: Round-off error estimation results for the example in Fig. 4

Fluctuat	Fluctuat (subdiv.)	Rosa	FPTaylor
∞	∞	1.78e-11	5.72e-12

B Additional Tool Capabilities and Illustrations

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., [18, 50]).

Conditional expressions can be verified in FPTaylor in the same way as it is done in Rosa [14]. 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 4 cases: 2 cases when both $f(x)$ and $\tilde{f}(x)$ take the same path, and 2 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 ($c(x) < 0$, etc.) 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

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

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 [16] 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 7 shows results

```
int main(void)
{
  double a, b;
  double 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

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 to find any error bound in this example. Results of FPTaylor are about 3 times better then Rosa's results.