On Sound Relative Error Bounds for Floating-Point Arithmetic

Anastasiia Izycheva and Eva Darulova
Max Planck Institute for Software Systems, Saarland Informatics Campus, Germany
Email: izycheva@mpi-sws.org, eva@mpi-sws.org

***Abstract*—State-of-the-art static analysis tools for verifying finite-precision code compute worst-case absolute error bounds on numerical errors. These are, however, often not a good estimate of accuracy as they do not take into account the magnitude of the computed values. Relative errors, which compute errors relative to the value's magnitude, are thus preferable. While today's tools do report relative error bounds, these are merely computed via absolute errors and thus not necessarily tight or more informative. Furthermore, whenever the computed value is close to zero on part of the domain, the tools do not report any relative error estimate at all. Surprisingly, the quality of relative error bounds computed by today's tools has not been systematically studied or reported to date.**

In this paper, we investigate how state-of-the-art static techniques for computing sound absolute error bounds can be used, extended and combined for the computation of relative errors. Our experiments on a standard benchmark set show that computing relative errors *directly*, as opposed to via absolute errors, is often beneficial and can provide error estimates up to six orders of magnitude tighter, i.e. more accurate. We also show that interval subdivision, another commonly used technique to reduce over-approximations, has less benefit when computing relative errors directly, but it can help to alleviate the effects of the inherent issue of relative error estimates close to zero.

I. Introduction

Numerical software, common in embedded systems or scientific computing, is usually designed in real-valued arithmetic, but has to be implemented in finite precision on digital computers. Finite precision, however, introduces unavoidable roundoff errors which are individually usually small, but which can accumulate and affect the validity of computed results. It is thus important to compute *sound* worst-case roundoff error bounds to ensure that results are accurate enough - especially for safety-critical applications. Due to the unintuitive nature of finite-precision arithmetic and its discrepancy with continuous real arithmetic, *automated* tool support is essential.

One way to measure worst-case roundoff is *absolute error*:

$$err_{\text{abs}} = \max_{x \in I} \left| f(x) - \hat{f}(\hat{x}) \right| \tag{1}$$

where f and x denote a possibly multivariate real-valued function and variable respectively, and $\hat{f}$ and $\hat{x}$ their finite-precision counter-parts. Note that absolute roundoff errors are only meaningful on a restricted domain, as for unrestricted x the error would be unbounded in general. In this paper, we consider interval constraints on input variables, that is for each input variable $x \in I = [a, b]$, $a, b \in \mathbb{R}$.

Furthermore, we focus on floating-point arithmetic, which is a common choice for many finite-precision programs, and for which several tools now exist that compute absolute errors fully automatically for nonlinear straight-line code [1]–[4].

Absolute errors are, however, not always an adequate measure of result quality. Consider for instance an absolute error of 0.1. Whether this error is small and thus acceptable for a computation depends on the application as well as the magnitude of the result's value: if $|f(x)| \gg 0.1$, then the error may be acceptable, while if $|f(x)| \approx 0.1$ we should probably revise the computation or increase its precision. *Relative error* captures this relationship:

$$err_{\text{rel}} = \max_{x \in I} \left| \frac{f(x) - \hat{f}(\hat{x})}{f(x)} \right| \tag{2}$$

Note that the input domain needs to be restricted also for relative errors.

Today's static analysis tools usually report absolute as well as relative errors. The latter is, however, computed via absolute errors. That is, the tools first compute the absolute error and then divide it by the largest function value:

$$err_{\text{rel\_approx}} = \frac{\max_{x \in I} \left| f(x) - \hat{f}(\hat{x}) \right|}{\min_{x \in I} |f(x)|} \tag{3}$$

Clearly, Equation 2 and Equation 3 both compute sound relative error bounds, but $err_{\text{rel\_approx}}$ is an over-approximation due to the loss of correlation between the nominator and denominator. Whether this loss of correlation leads to coarse error bounds in practice has, perhaps surprisingly, not been studied yet in the context of automated sound error estimation.

Beyond curiosity, we are interested in the automated computation of relative errors for several reasons. First, relative errors are more informative and often also more natural for user specifications. Secondly, when computing sound error bounds, we necessarily compute over-approximations. For absolute errors, the over-approximations become bigger for larger input ranges, i.e. the error bounds are less tight. Since relative errors consider the range of the expression, we expect these over-approximations to be smaller, thus making relative errors more suitable for modular verification.

One may think that computing relative errors is no more challenging than computing absolute errors; this is not the case for two reasons. First, the complexity of computing relative errors is higher (compare Equation 1 and Equation 2) and due to the division, the expression is now nonlinear even for linear f. Both complexity and nonlinearity have already been challenging for absolute errors computed by automated tools, usually leading to coarser error bounds. Furthermore, whenever the range of f includes zero, we face an inherent division by zero. Indeed, today's static analysis tools report no relative error for most standard benchmarks for this reason.

Today's static analysis tools employ a variety of different strategies (some orthogonal) for dealing with the over-approximation of worst-case absolute roundoff errors due to nonlinear arithmetic: the tool Rosa uses a forward dataflow analysis with a linear abstract domain combined with a nonlinear decision procedure [3], Fluctuat augments a similar linear analysis with interval subdivision [1], and FPTaylor chooses an optimization-based approach [2] backed by a branch-and-bound algorithm.

In this paper, we investigate how today's methods can be used, extended and combined for the computation of relative errors. To the best of our knowledge, this is the first systematic study of fully automated techniques for the computation of relative errors. We mainly focus on the issue of computing tight relative error bounds and for this extend the optimization based approach for computing absolute errors to computing relative errors *directly* and show experimentally that it often results in tighter error bounds, sometimes by up to six orders of magnitude. We furthermore combine it with interval subdivision (we are not aware of interval subdivision being applied to this approach before), however, we note that, perhaps surprisingly, the benefits are rather modest.

We compare this direct error computation to forward analysis which computes relative errors via absolute errors on a standard benchmark set, and note that the latter outperforms direct relative error computation only on a single univariate benchmark. On the other hand, this approach clearly benefits from interval subdivision.

We also observe that interval subdivision is beneficial for dealing with the inherent division by zero issue in relative error computations. We propose a practical (and preliminary) solution, which reduces the impact of potential division-by-zero's to small subdomains, allowing our tool to compute relative errors for the remainder of the domain. We demonstrate on our benchmarks that this approach allows our tool to provide more useful information than state-of-the-art tools.

Contributions:

- We extend an optimization-based approach [2] for bounding absolute errors to relative errors and thus provide the first feasible and fully automated approach for computing relative errors *directly*.
- We perform the first experimental comparison of different techniques for automated computation of sound relative error bounds.
- We show that interval subdivision is beneficial for reducing the over-approximation in absolute error computations, but less so for relative errors computed directly.
- We demonstrate that interval subdivision provides a practical solution to the division by zero challenge of relative error computations for certain benchmarks.

We have implemented all techniques within the tool Daisy [5], which is available at https://github.com/malyzajko/daisy.

II. Background

We first give a brief overview over floating-point arithmetic as well as state-of-the-art techniques for automated sound worst-case absolute roundoff error estimation.

A. Floating-Point Arithmetic

The error definitions in section I include a finite-precision function $\hat{f}(\hat{x})$ which is highly irregular and discontinuous and thus unsuitable for automated analysis. We abstract it following the floating-point IEEE 754 standard [6], by replacing every floating-point variable, constant and operation by:

$$\begin{gathered} x \odot y = (x \circ y)(1+e) + d, \\ \tilde{x} = x(1+e) + d \qquad \tilde{\sqrt{x}} = \sqrt{x}(1+e) + d \end{gathered} \tag{4}$$

where $\odot \in \{\oplus, \ominus, \otimes, \oslash\}$ and $\circ \in \{+, -, \times, /\}$ are floating-point and real arithmetic operations, respectively. e is the relative error introduced by rounding at each operation and is bounded by the so-called machine epsilon $|e| \leq \epsilon_M$. Denormals (or subnormals) are values with a special representation which provide gradual underflow. For these, the roundoff error is expressed as an absolute error d and is bounded by δ_M, (for addition and subtraction $d = 0$). This abstraction is valid in the absence of overflow and invalid operations resulting in Not a Number (NaN) values. These values are detected separately and reported as errors. In this paper, we consider double precision floating-point arithmetic with $\epsilon_M = 2^{-53}$ and $\delta_M = 2^{-1075}$. Our approach is parametric in the precision, and thus applicable to other floating-point types as well.

Using this abstraction we replace $\hat{f}(\hat{x})$ with a function $\tilde{f}(x, e, d)$, where x are the input variables and e and d the roundoff errors introduced for each floating-point operation. In general, x, e and d are vector-valued, but for ease of reading we will write them without vector notation. Note that our floating-point abstraction is real-valued. With this abstraction, we and all state-of-the-art analysis tools approximate absolute errors as:

$$err_{abs} \leq \max_{x \in I, |e_i| \leq \epsilon_M, |d_i| \leq \delta} \left| f(x) - \tilde{f}(x, e, d) \right| \tag{5}$$

B. State-of-the-art in Absolute Error Estimation

When reviewing existing automated tools for absolute roundoff error estimation, we focus on their techniques for reducing over-approximations due to nonlinear arithmetic in order to compute tight error bounds.

Rosa [3] computes absolute error bounds using a forward data-flow analysis and a combination of abstract domains. Note that the magnitude of the absolute roundoff error at an arithmetic operation depends on the magnitude of the operation's value (this can easily be seen from Equation 4), and these are in turn determined by the input parameter ranges. Thus, Rosa tracks two values for each intermediate abstract syntax tree node: a sound approximation of the range and the worst-case absolute error. The transfer function for errors uses the ranges to propagate errors from subexpressions and to compute the new roundoff error committed by the arithmetic operation in question.

One may think that evaluating an expression in interval arithmetic [7] and interpreting the width of the resulting interval as the error bound would be sufficient. While this is sound, it computes too pessimistic error bounds, especially if we consider relatively large ranges on inputs: we cannot distinguish which part of the interval width is due to the input interval or due to accumulated roundoff errors. Hence, we need to compute ranges and errors separately.

Rosa implements different range arithmetics with different accuracy-efficiency tradeoffs for bounding ranges and errors. To compute ranges, Rosa offers a choice between interval arithmetic, affine arithmetic [8] (which tracks linear correlations between variables) and a combination of interval arithmetic with a nonlinear arithmetic decision procedure. The latter procedure first computes the range of an expression in standard interval arithmetic and then refines, i.e. tightens, it using calls to the nlsat [9] decision procedure within the Z3 SMT solver [10]. For tracking errors, Rosa uses affine arithmetic; since roundoff errors are in general small, tracking linear correlations is in general sufficient.

Fluctuat [1] is an abstract interpreter which separates errors similarly to Rosa and which uses affine arithmetic for computing both the ranges of variables and for the error bounds. In order to reduce the over-approximations introduced by affine arithmetic for nonlinear operations, Fluctuat uses interval subdivision. The user can designate up to two variables in the program whose input ranges will be subdivided into intervals of equal width. The analysis is performed separately and the overall error is then the maximum error over all subintervals. Interval subdivision increases the runtime of the analysis significantly, especially for multivariate functions, and the choice of which variables to subdivide and by how much is usually not straight-forward.

FPTaylor, unlike Daisy and Fluctuat, formulates the roundoff error bounds computation as an optimization problem, where the absolute error expression from Equation 1 is to be maximized, subject to interval constraints on its parameters. Due to the discrete nature of floating-point arithmetic, FPTaylor optimizes the continuous, real-valued abstraction (Equation 5). However, this expression is still too complex and features too many variables for optimization procedures in practice, resulting in bad performance as well as bounds which are too coarse to be useful (see subsection V-A for our own experiments). FPTaylor introduces the Symbolic Taylor approach, where the objective function of Equation 5 is simplified using a first order Taylor approximation with respect to e and d:

$$\tilde{f}(x, e, d) = \tilde{f}(x, 0, 0) + \sum_{i=1}^{k} \frac{\partial \tilde{f}}{\partial e_i}(x, 0, 0) e_i + R(x, e, d), \quad (6)$$

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

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

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

$$err_{abs} \leq \epsilon_M \max_{x \in I} \sum_{i=1}^{k} \left| \frac{\partial \tilde{f}}{\partial e_i}(x, 0, 0) \right| + M_R \quad (7)$$

where M_R is an upper bound for the error term $R(x, e, d)$ (more details can be found in [2]). FPTaylor uses interval arithmetic to estimate the value of M_R as the term is second order and thus small in general.

To solve the optimization problem in Equation 7, FPTaylor uses rigorous branch-and-bound optimization. Branch-and-bound is also used to compute the resulting real function $f(x)$ range, which is needed for instance to compute relative errors. Real2Float [4], another tool, takes the same optimization-based approach, but uses semi-definite programming for the optimization itself.

III. Bounding Relative Errors

The main goal of this paper is to investigate how today's sound approaches for computing absolute errors fare for bounding relative errors and whether it is possible and advantageous to compute relative errors directly (and not via absolute errors). In this section, we first concentrate on obtaining tight bounds in the presence of nonlinear arithmetic, and postpone a discussion of the orthogonal issue of division by zero to the next section. Thus, we assume for now that the range of the function for which we want to bound relative errors does not include zero, i.e. $0 \notin f(x)$ and $0 \notin \tilde{f}(\tilde{x})$, for $x, \tilde{x}$ within some given input domain. We furthermore consider f to be a straight-line arithmetic expression. Conditionals and loops have been shown to be challenging [11] even for absolute errors and we thus leave their proper treatment for future work. We consider function calls to be an orthogonal issue; they can be handled by inlining, thus reducing to straight-line code, or require suitable summaries in postconditions, which is also one of the motivations for this work.

The forward dataflow analysis approach of Rosa and Fluctuat does not easily generalize to relative errors, as it requires intertwining the range and error computation. Instead, we extend FPTaylor's approach to computing relative errors directly (subsection III-A). We furthermore implement interval subdivision (subsection III-B), which is an orthogonal measure to reduce over-approximation and experimentally evaluate different combinations of techniques on a set of standard benchmarks (subsection V-A).

A. Bounding Relative Errors Directly

The first strategy we explore is to compute relative errors directly, without taking the intermediate step through absolute errors. That is, we extend FPTaylor's optimization based approach and maximize the relative error expression using the floating-point abstraction from Equation 4:

$$\max|\tilde{g}(x,e,d)| = \max_{x\in I, |e_i|\leq \epsilon_M, |d_i|\leq\delta} \left| \frac{f(x) - \tilde{f}(x,e,d)}{f(x)} \right| \quad (8)$$

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

We call the optimization of Equation 8 without simplifications the *naive approach*. While it has been mentioned previously that this approach does not scale well [2], we include it in our experiments (in subsection V-A) nonetheless, as we are not aware of any concrete results actually being reported. As expected, the naive approach returns error bounds which are so large that they are essentially useless.

We thus simplify $\tilde{g}(x,e,d)$ by applying the Symbolic Taylor approach introduced by FPTaylor [2]. As before, we take the Taylor approximation around the point $(x,0,0)$, so that the first term of the approximation is zero as before: $\tilde{g}(x,0,0) = 0$. We obtain the following optimization problem:

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

where M_R is an upper bound for the remainder term $R(x,e,d)$. Unlike in Equation 7 we do not pull the factor e_i for each term out of the absolute value, as we plan to compute tight bounds for mixed-precision in the future, where the upper bounds on all e_i are not all the same (this change does not affect the accuracy for uniform precision computations).

Computing Upper Bounds: The second order remainder R is still expected to be small, so that we use interval arithmetic to compute a sound bound on M_R (in our experiments R is indeed small for all benchmarks except 'doppler'). To bound the first order terms $\frac{\partial \tilde{g}}{\partial e_i}$ we need a sound optimization procedure to maintain overall soundness, which limits the available choices significantly.

FPTaylor uses the global optimization tool Gelpia [12], which internally uses a branch-and-bound based algorithm. Unfortunately, we found it difficult to integrate because of its custom interface. Furthermore, we observed unpredictable behavior in our experiments (e.g. nondeterministic crashes and substantially varying running times for repeated runs on the same expression).

Instead, we use Rosa's approach which combines interval arithmetic with a solver-based refinement. Our approach is parametric in the solver and we experiment with Z3 [10] and dReal [13]. Both support the SMT-lib interface, provide rigorous results, but are based on fundamentally different techniques. Z3 implements a complete decision procedure for nonlinear arithmetic [9], whereas dReal implements the framework of δ-complete decision procedures. Internally, it is based on a branch-and-bound algorithm and is thus in principle similar to Gelpia's optimization-based approach.

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

B. Interval Subdivision

The over-approximation committed by static analysis techniques grows in general with the width of the input intervals, and thus with the width of all intermediate ranges. Intuitively, the worst-case error which we consider is usually achieved only for a small part of the domain, over-approximating the error for the remaining inputs. Additionally the domain where worst-case errors are obtained may be different at each arithmetic operation. Thus, by subdividing the input domain we can usually obtain tighter error bounds. Note that interval subdivision can be applied to any error estimation approach. Fluctuat has applied interval subdivision to absolute error estimation, but we are not aware of a combination with the optimization-based approach, nor about a study of its effectiveness for relative errors.

We apply subdivision to input variables and thus compute:

$$\max_{k\in[1\ldots m]} \left(\max_{x_j\in I_{jk}} |\tilde{g}(x,e,d)| \right) \quad (9)$$

where m is an number of subdivisions for each input interval. That is, for multivariate functions, we subdivide the input interval for each variable and maximize the error over the Cartesian product. Clearly, the analysis running time is exponential in the number of variables. While Fluctuat limits subdivisions to two user-designated variables and a user-defined number of subdivisions each, we choose to limit the total number of analysis runs by a user-specified parameter p. That is, given p, m (the desired number of subdivisions for each variable), and n (number of input variables), the first $\lfloor \log_m(p-n) \rfloor$ variables' domains are subdivided m times, with larger input domains subdivided first. The remaining variable ranges remain undivided.

C. Implementation

We implement all the described techniques in the tool Daisy [5]. Daisy, a successor of Rosa [3], is a source-to-source compiler which generates floating-point implementations from real-valued specifications given in the following format:

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

Daisy is parametric in the *approach* (naive, forward dataflow analysis or optimization-based), the *solver* used (Z3 or dReal) and the number of *subdivisions* (including none). Any combination of these three orthogonal choices can be run by simply changing Daisy's input parameters.

Furthermore, Daisy simplifies the derivative expressions in the optimization-based approach ($x + 0 = x, x \times 1 = x$, etc.). Unsimplified expressions may affect the running time of the solvers (and thus also the accuracy of the error bounds), as we observed that the solvers do not necessarily perform these otherwise straight-forward simplifications themselves.

Finally, to maintain soundness, we need to make sure that we do not introduce internal roundoff errors during the computation of error bounds. For this purpose we implement all internal computations in Daisy using infinite-precision rationals.

IV. Handling Division by Zero

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

Note that these divisions by zero are *inherent* to the expression which we consider and are usually not due to over-approximations in the analysis. Thus, we can only *reduce* the effect of division by zero, but we cannot eliminate it entirely. Here, we aim to reduce the domain for which we cannot compute relative errors. Similar to how relative and absolute errors are combined in the IEEE 754 floating-point model (Equation 4), we want to identify parts of the input domain on which relative error computation is not possible due to division by zero and compute absolute errors. For the remainder of the input domain, we compute relative errors as before.

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

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

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

We realize that while this approach provides a practical solution, it is still preliminary and can be improved in several ways. First, a smarter subdivision strategy would be beneficial. Currently, we divide the domain into equal-width intervals, and vary only their number. The more fine-grained the subdivision, the bigger part of the domain can be covered by relative error computations, however the running time increases correspondingly. Ideally, we could exclude from the relative error computation only a small domain around the zeros of the function f. While for univariate functions, this approach is straight-forward (zeros can be, for instance, obtained with a nonlinear decision procedure), for multivariate functions this is challenging, as the zeros are not simple points but curves. Secondly, subdivision could only be used for determining which sub-domains exhibit potential division by zero. The actual relative error bound computation can then be performed on the remainder of the input domain without subdividing it. This would lead to performance improvements, even though the 'guaranteed-no-zero' domain can still consist of several disconnected parts. Again, for univariate functions this is a straight-forward extension, but non-trivial for multivariate ones. Finally, we could compute $\max_{x_j \in I_{jk}} \left| \frac{f(x) - \tilde{f}(x,e,d)}{f(x)+\epsilon} \right|$. for some small ϵ, which is a standard approach in scientific computing. It is not sound, however, so that we do not consider it here.

V. Experimental Evaluation

We compare the different strategies for relative error computation on a set of standard benchmarks with FPTaylor and the forward dataflow analysis approach from Rosa (now implemented in Daisy) as representatives of state-of-the-art tools. We do not compare to Fluctuat directly as the underlying error estimation technique based on forward analysis with affine arithmetic is very similar to Daisy's. Indeed experiments performed previously [2], [11] show only small differences in computed error bounds. We rather choose to implement interval subdivision within Daisy.

All experiments are performed in double floating-point precision (the precision FPTaylor supports), although all techniques in Daisy are parametric in the precision. The experiments were performed on a desktop computer running Debian GNU/Linux 8 64-bit with a 3.40GHz i5 CPU and 7.8GB RAM. The benchmarks bsplines, doppler, jetEngine, rigidBody, sine, sqrt and turbine are nonlinear functions from [3]; invertedPendulum and the traincar benchmarks are linear embedded examples from [14]; and himmilbeau and kepler are nonlinear examples from the Real2Float project [4].

A. Comparing Techniques for Relative Error Bounds

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

Table I shows the relative error bounds computed with the different techniques and tools, and Table II the corresponding analysis times. Bold marks the best result, i.e. tightest computed error bound, for each benchmark. The column 'Under-approx' gives an (unsound) relative error bound obtained with dynamic evaluation on 100000 inputs; these values provide an idea of the true relative errors. The 'Naive approach' column confirms that simplifications of the relative error expression are indeed necessary (note the exponents of the computed bounds).

TABLE I
RELATIVE ERROR BOUNDS COMPUTED BY DIFFERENT TECHNIQUES

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

TABLE II
ANALYSIS TIME OF DIFFERENT TECHNIQUES FOR RELATIVE ERRORS ON BENCHMARKS WITHOUT DIVISION BY ZERO

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

The last four columns show the error bounds when relative errors are computed directly using the optimization based approach (denoted 'DaisyOPT') from subsection III-A, with the two solvers and with and without subdivisions. For subdivisions, we use $m = 2$ for univariate benchmarks, $m = 8$ for multivariate and $p = 50$ for both as in our experiments these parameters showed a good trade-off between performance and accuracy. For most of the benchmarks we find that direct evaluation of relative errors computes tightest error bounds with acceptable analysis times. Furthermore, for most benchmarks Z3, resp. its nonlinear decision procedure, is able to compute slightly tighter error bounds, but for three of our benchmarks dReal performs significantly better, while the running times are comparable.

Somewhat surprisingly, we note that interval subdivision has limited effect on accuracy when combined with direct relative error computation, while also increasing the running time significantly.

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

The column 'Daisy+subdiv' shows relative errors computed via absolute errors, using the forward analysis with subdivision (with the same parameters as above). Here we observe that unlike for the directly computed relative errors, interval subdivision is mostly beneficial.

Finally, for the experiments in Table I, we use as large input domains as possible, without introducing result ranges which include zero. When comparing relative error bounds computed for smaller and larger input domains, where a small input domain means that the input intervals have smaller width, we observe that relative errors computed directly usually scale better than relative errors computed via absolute errors, i.e. the over-approximation committed is smaller. For example, (for space reasons only) for the doppler benchmark we obtain the following relative errors:

	Daisy (via absolute)	relative err. directly
small input domain	1.48e-11	1.26e-15
large input domain	2.08e-04	1.93e-13

B. Handling Division by Zero

To evaluate whether interval subdivision is helpful when dealing with the inherent division by zero challenge, we now consider the standard benchmark set, with standard input domains. Table III summarizes our results. We first note that division by zero indeed occurs quite often, as the missing results in the Daisy and FPtaylor columns show.

The last three columns show our results when using interval subdivision. Note that to obtain results on as many benchmarks as possible we had to change the parameters for subdivision to $m = 8$ and $p = 50$ for univariate and $m = 4, p = 100$ for multivariate benchmarks. The result consists of three values: the first value is the maximum relative error computed over the sub-domains where relative error was possible to compute; in the brackets we report the maximum absolute error for the sub-domains where relative error computation is not possible, and the integer is the amount of these sub-domains where absolute errors were computed. We only report a result if the number of sub-domains with division by zero is less than 80% of the total amount of subdomains, as larger numbers would probably be impractical to be used within, e.g. modular verification techniques. Whenever we report '-' in the table, this means that division by zero occurred on too many or all subdomains.

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

VI. Related work

The goal of this work is an automated and sound static analysis technique for computing tight relative error bounds for floating-point arithmetic. Most related are current static analysis tools for computing *absolute* roundoff error bounds [1]–[4].

Another closely related tool is Gappa [15], which computes both absolute and relative error bounds in Coq. It appears relative errors can be computed both directly and via absolute errors. The *automated* error computation in Gappa uses intervals, thus, a computation via absolute errors will be less accurate than Daisy performs. The direct computation amounts to the naive approach, which we have shown to work poorly.

The direct relative error computation was also used in the context of verifying computations which mix floating-point arithmetic and bit-level operations [16]. Roundoff errors are computed using an optimization based approach similar to FPTaylor's. Their approach is targeted to specific low-level operations including only polynomials, and the authors do not use Taylor's theorem. However, tight error estimates are not the focus of the paper, and the authors only report that they use whichever bound (absolute or relative) is better. we are not aware of any systematic evaluation of different approaches for sound relative error bounds.

More broadly related are abstract interpretation-based static analyses which are sound wrt. floating-point arithmetic [17], [18], some of which have been formalized in Coq [19] These domains, however, do not quantify the difference between the real-valued and the finite-precision semantics and can only show the absence of runtime errors such division-by-zero or overflow.

Floating-point arithmetic has also been formalized in an SMT-lib [20] theory and solvers exist which include floating-point decision procedures [20], [21]. These are, however, not suitable for roundoff error quantification, as a combination with the theory of reals would be at the propositional level only and thus not lead to useful results.

Floating-point arithmetic has also been formalized in theorem provers such as Coq [22] and HOL Light [23], and some automation support exists in the form of verification condition generation and reasoning about ranges [24], [25]. Entire numerical programs have been proven correct and accurate within these [26], [27]. While very tight error bounds can be proven for specific computations [28], these verification efforts are to a large part manual and require substantial user expertise in both floating-point arithmetic as well as theorem proving. Our work focuses on a different trade-off between accuracy, automation and generality.

Another common theme is to run a higher-precision program alongside the original one to obtain error bounds by testing [29]–[32]. Testing has also been used as a verification method for optimizing mixed-precision computations [33], [34]. These approaches based on testing, however, only consider a limited number of program executions and thus cannot prove sound error bounds.

VII. Conclusion

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

We furthermore note that today's rigorous optimization tools could be improved in terms of reliability as well as scalability. Finally, while interval subdivision can help to alleviate the effect of the inherent division by zero issue in relative error computation, it still remains an open challenge.

TABLE III
RELATIVE ERROR BOUNDS COMPUTED BY DIFFERENT TECHNIQUES ON STANDARD BENCHMARKS (WITH POTENTIAL DIVISION BY ZERO)

Benchmark	Daisy	FPTaylor	Daisy + subdiv	DaisyOPT Z3 + subdiv	DaisyOPT dReal + subdiv
bspline0	-	-	1.58e-01 (1.08e-18, 1)	3.00e-15 (1.08e-18, 1)	3.00e-15 (1.08e-18, 1)
bspline1	-	3.32e-15	2.80e-13	3.22e-15	3.22e-15
bspline2	-	3.50e-15	9.20e-16	8.92e-16	8.92e-16
bspline3	-	-	1.31e-14 (9.67e-19, 1)	6.66e-16 (9.67e-19, 1)	6.66e-16 (9.67e-19, 1)
sine	-	-	1.07e-15 (2.00e-16, 2)	7.02e-16 (2.02e-16, 2)	7.02e-16 (2.02e-16, 2)
sineOrder3	-	-	2.29e-15 (3.10e-16, 2)	8.94e-16 (3.17e-16, 2)	8.94e-16 (3.17e-16, 2)
sqroot	-	-	7.09e-15 (2.83e-14, 3)	1.92e-15 (3.11e-14, 3)	1.92e-15 (3.11e-14, 3)
doppler	1.48e-11	4.99e-12	8.95e-13	1.26e-15	1.35e-15
himmilbeau	-	-	3.75e-14 (1.00e-12, 12)	2.57e-14 (1.00e-12, 12)	2.84e-15 (1.00e-12, 12)
invPendulum	-	-	4.94e-15 (2.60e-14, 32)	2.82e-15 (2.60e-14, 32)	3.08e-15 (2.60e-14, 32)
jet	-	-	-	-	-
kepler0	4.35e-15	4.57e-15	2.38e-13 (8.08e-14, 49)	2.16e-15 (7.92e-14, 49)	3.88e-15 (8.32e-14, 49)
kepler1	1.33e-14	1.17e-14	-	-	-
kepler2	-	4.21e-14	-	-	-
rigidBody1	-	-	2.29e-14 (2.16e-13, 46)	1.07e-15 (2.16e-13, 46)	1.78e-15 (2.16e-13, 46)
rigidBody2	-	-	2.65e-12 (3.51e-11, 50)	1.67e-15 (3.65e-11, 50)	3.80e-15 (3.65e-11, 50)
traincar_state8	-	-	-	-	-
traincar_state9	-	-	-	-	-
turbine1	6.12e-14	1.18e-14	6.03e-15	1.75e-15	5.21e-15
turbine2	-	-	5.64e-14 (3.65e-14, 25)	2.74e-15 (1.20e-13, 25)	6.97e-14 (3.90e-14, 25)
turbine3	1.52e-13	2.21e-14	2.77e-14	6.50e-15	6.71e-15

REFERENCES

[1] E. Goubault and S. Putot, "Static Analysis of Finite Precision Computations," in *VMCAI*, 2011.

[2] A. Solovyev, C. Jacobsen, Z. Rakamarić, and G. Gopalakrishnan, "Rigorous Estimation of Floating-Point Round-off Errors with Symbolic Taylor Expansions," in *FM*, 2015.

[3] E. Darulova and V. Kuncak, "Sound Compilation of Reals," in *POPL*, 2014.

[4] V. Magron, G. A. Constantinides, and A. F. Donaldson, "Certified Roundoff Error Bounds Using Semidefinite Programming," *CoRR*, vol. abs/1507.03331, 2015.

[5] MPI-SWS, "Daisy - a framework for accuracy analysis and synthesis of numerical programs," 2017, https://github.com/malyzajko/daisy.

[6] "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2008*, Aug 2008.

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

[8] L. H. de Figueiredo and J. Stolfi, "Affine Arithmetic: Concepts and Applications," *Numerical Algorithms*, vol. 37, no. 1-4, 2004.

[9] D. Jovanović and L. de Moura, "Solving non-linear arithmetic," in *IJCAR*, 2012.

[10] L. De Moura and N. Bjørner, "Z3: an efficient SMT solver," in *TACAS*, 2008.

[11] E. Darulova and V. Kuncak, "Towards a Compiler for Reals," *ACM TOPLAS*, vol. 39, no. 2, 2017.

[12] M. S. Baranowski and I. Briggs, "Gelpia - global extrema locator parallelization for interval arithmetic," https://github.com/soarlab/gelpia.

[13] S. Gao, S. Kong, and E. M. Clarke, "dReal: An SMT Solver for Nonlinear Theories over the Reals," in *CADE*, 2013.

[14] E. Darulova, V. Kuncak, R. Majumdar, and I. Saha, "Synthesis of Fixed-point Programs," in *EMSOFT*, 2013.

[15] M. Daumas and G. Melquiond, "Certification of bounds on expressions involving rounded operators," *ACM Trans. Math. Softw.*, vol. 37, no. 1, pp. 2:1–2:20, Jan. 2010.

[16] W. Lee, R. S. 0001, and A. Aiken, "Verifying Bit-Manipulations of Floating-Point." *PLDI*, 2016.

[17] L. Chen, A. Miné, and P. Cousot, "A Sound Floating-Point Polyhedra Abstract Domain," in *APLAS*, 2008.

[18] B. Jeannet and A. Miné, "Apron: A Library of Numerical Abstract Domains for Static Analysis," in *CAV*, 2009.

[19] J.-H. Jourdan, V. Laporte, S. Blazy, X. Leroy, and D. Pichardie, "A Formally-Verified C Static Analyzer," in *POPL*, 2015.

[20] P. R. Martin Brain, Cesare Tinelli and T. Wahl, "An Automatable Formal Semantics for IEEE-754 Floating-Point Arithmetic," Technical Report, http://smt-lib.org/papers/BTRW15.pdf, 2015.

[21] M. Brain, V. D'Silva, A. Griggio, L. Haller, and D. Kroening, "Deciding floating-point logic with abstract conflict driven clause learning," *Formal Methods in System Design*, vol. 45, no. 2, pp. 213–245, Dec. 2013.

[22] S. Boldo and G. Melquiond, "Flocq: A Unified Library for Proving Floating-Point Algorithms in Coq," in *ARITH*, 2011.

[23] C. Jacobsen, A. Solovyev, and G. Gopalakrishnan, "A Parameterized Floating-Point Formalizaton in HOL Light," *Electronic Notes in Theoretical Computer Science*, vol. 317, pp. 101–107, 2015.

[24] M. D. Linderman, M. Ho, D. L. Dill, T. H. Meng, and G. P. Nolan, "Towards program optimization through automated analysis of numerical precision," in *CGO*, 2010.

[25] A. Ayad and C. Marché, "Multi-prover verification of floating-point programs," in *IJCAR*, 2010.

[26] S. Boldo, F. Clément, J.-C. Filliâtre, M. Mayero, G. Melquiond, and P. Weis, "Wave Equation Numerical Resolution: A Comprehensive Mechanized Proof of a C Program," *Journal of Automated Reasoning*, vol. 50, no. 4, pp. 423–456, 2013.

[27] T. Ramananandro, P. Mountcastle, B. Meister, and R. Lethin, "A Unified Coq Framework for Verifying C Programs with Floating-Point Computations," in *CPP*, 2016.

[28] S. Graillat, V. Lefèvre, and J.-M. Muller, "On the maximum relative error when computing x^n in floating-point arithmetic," Université Pierre et Marie Curie Paris 6 ; CNRS, Research Report, ensl-00945033, 2014.

[29] F. Benz, A. Hildebrandt, and S. Hack, "A Dynamic Program Analysis to Find Floating-point Accuracy Problems," in *PLDI*, 2012.

[30] W.-F. Chiang, G. Gopalakrishnan, Z. Rakamaric, and A. Solovyev, "Efficient Search for Inputs Causing High Floating-point Errors," in *PPoPP*, 2014.

[31] M. O. Lam, J. K. Hollingsworth, and G. Stewart, "Dynamic Floating-point Cancellation Detection," *Parallel Computing*, vol. 39, no. 3, 2013.

[32] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock, "Automatically Improving Accuracy for Floating Point Expressions," in *PLDI*, 2015.

[33] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough, "Precimonious: Tuning Assistant for Floating-point Precision," in *SC*, 2013.

[34] M. O. Lam, J. K. Hollingsworth, B. R. de Supinski, and M. P. Legendre, "Automatically Adapting Programs for Mixed-precision Floating-point Computation," in *ICS*, 2013.