On Numerical Error Propagation with Sensitivity

Eva Darulova  Viktor Kuncak
EPFL
first.last@epfl.ch

Abstract

An emerging area of research is to automatically compute reasonably precise upper bounds on numerical errors including roundoffs. Previous approaches for this task are limited in their precision and scalability, especially in the presence of branches and loops. We argue that one reason for these limitations is the focus of past approaches on approximating errors of individual reachable states. We propose instead a more relational and modular approach to analysis that characterizes analytically the input/output behavior of code fragments and reuses this characterization to reason about larger code fragments. We use the derivatives of the functions corresponding to program paths to capture a program’s sensitivity to input changes. To apply this approach for finite-precision code, we decouple the computation of newly introduced roundoff errors from the amplification of existing errors. This enables us to precisely and efficiently account for propagation of errors through long-running computation. Using this approach we implemented an analysis for programs containing nonlinear computation, conditionals, and loops. In the presence of loops our approach can find closed-form symbolic invariants capturing upper bounds on numerical errors, even when the error grows with the number of iterations. We evaluate our system on a number of benchmarks from embedded systems and scientific computation, showing substantial improvements in precision and scalability over the state of the art.

1. Introduction

Numerical software, common in scientific computing or embedded systems, inevitably uses an approximation of the real arithmetic in which its algorithms have been designed. Many problem domains come with additional sources of precision, such as measurement and truncation errors, increasing the uncertainty on the computed results. Adequate tools are needed to help users select suitable approximations (data types and algorithms) which satisfy their accuracy requirements, especially for safety-critical applications. Precise and sound error estimation is hard particularly because of nonlinearity of computations, long-running loops that cannot be unfolded statically, and discontinuities due to branches. Roundoff errors and error propagation depend on the ranges of variables in complex and non-obvious ways; even determining these ranges precisely for nonlinear codes poses a significant challenge. In numerical loops, roundoff errors grow, in general, unboundedly. As an illustration, consider a simulation of the planet Jupiter orbiting the Sun for which we plot the absolute errors of one of the coordinates, $x$, in Figure 1. While the values of $x$ stay bounded, the errors grow, making it impossible to find inductive specifications of constant absolute bounds. Finally, due to numerical errors, the control flow in the finite-precision implementation may diverge from the ideal real-valued one by taking a different branch and produce a result that is far off the expected one. Quantifying discontinuity errors is hard due to many correlations and nonlinearity but also often lack of smoothness or continuity of the underlying functions.

Limitations of state of the art tools. Existing state-of-the-art sound and automated error estimation techniques rely on stepwise application of affine arithmetic (AA) [13], which, due to its linear nature, inevitably introduce over-approximations when applied to nonlinear functions. Fluctuat [19] is an abstract interpretation based tool which uses affine arithmetic both for the range and the error computation, but adds constraints on the noise terms to improve the ranges computed by AA [18]. Rosa [13] improves ranges with an SMT- backed procedure, but uses for the error computation essentially the same technique as Fluctuat. Affine arithmetic tracks a computation step by step, and thus fails to capture the overall effect of a function on uncertainties. When it comes to loops, roundoff errors may accumulate without a constant upper bound. Fluctuat is forced to unroll loops or apply widening, often returning a trivial upper bound of $\infty$ in such cases. Rosa can only handle recursive functions with given inductive specifications with constant absolute error bounds. Such specifications do not exist in general and furthermore, Rosa does not compute these specifications automatically. Finally, both Fluctuat and Rosa include a procedure to soundly estimate discontinuity errors, but the approaches work well only for linear or unary functions, severely limiting the analysis of numerical code containing branches.

Separation of Errors. We propose a new error computation based on separating the propagation of existing errors from the roundoff or truncation errors committed during the computation. This separation allows us to distinguish the implementation aspects from the mathematical properties of the underlying function and handle them individually with appropriate techniques. In particular, this separation allows us to directly use the properties of the real-valued functions underlying the finite-precision implementation. Such a separation of errors applies fairly generally: it enabled us to 1) improve computed error bounds on straight-line nonlinear code, 2) characterize errors in loops as functions of the number of iterations, and 3) scale discontinuity error computations to multivariate functions.

Propagation and Sensitivity. Using the separation of errors, our new propagation procedure improves the computed error bounds compared to affine arithmetic. Our approach considers an entire arithmetic expression at a time, computing an approximation of the global effect of the function on the input errors, in contrast to the local linear approximation of affine arithmetic. Additionally, our procedure is backed by a nonlinear SMT-solver, which allows us to capture nonlinear correlations precisely. This can then be used for computing more precise bounds on the propagation error. Moreover, it provides information about the sensitivity of the function with respect to the errors, which is useful for understanding the behavior of the function and for modular inter-procedural analysis. We primarily use an approximation that is a linear expression in the input errors, based on the Jacobian of the (entire) function. We also develop a higher-order extension which can, for the first time,
We therefore propose to compute the numerical errors as a function which preserve enough precision. We show in our experimental results that this tradeoff between precision and scalability outperforms current techniques, sometimes by three orders of magnitude. We apply the idea of separation of errors into three challenging dimensions of numerical error estimation: nonlinearity, loops and discontinuities.

1.1 Summary of Contributions

Our contribution is a new approach for numerical error estimation for nonlinear codes with loops and discontinuities. Specifically:

- We propose an approach for automatic error estimation based on the idea of separation of errors into propagation errors and roundoff errors. We show that this idea is general and can be applied to three challenging dimensions of numerical error estimation: nonlinearity, loops and discontinuities.
- We develop an approach for computing propagation errors using Lipschitz continuity to characterize the sensitivity of a function to input changes and apply this approach to non-linear computation. Our approach computes an approximation of the global effect of a function on input errors. We show that this enables us to compute significantly better error estimates than current affine arithmetic based techniques.
- We derive a technique for computing error bounds as a function of the number of iterations of a loop. This allows us, for the first time, to compute error bounds for loops where errors grow unboundedly in an inductive way. We show that our technique scales to loops of a complexity that cannot be handled by loop unrolling with current tools.
- We apply the idea of separation of errors to programs with branches to develop a novel way of soundly estimating the discontinuity errors arising from the real-valued ideal computation taking a different branch than the finite precision one. We show that our new approach is more scalable than the state of the art, and provides significantly better results for multivariate, nonlinear functions.
- We have implemented our techniques and report substantially improved results compared to existing tools on a number of benchmarks from the scientific computing and embedded systems domain. We summarize many of our benchmarks; their full source code is also available as supplementary material.

2. Examples

We next present several examples that illustrate the capabilities of our system. Our framework accepts programs written in a functional real-valued specification language. The current implementation follows Rosa [13] and accepts a subset of Scala extended with appropriate specification constructs. Although the choice of concrete syntax is largely orthogonal to our technique, we do rely on the model where programs specify ideal real computation, with the additional indication of the bounds on permissible errors in the implementation due to the finite representation of numbers [13].

Pre- and postconditions given in the require and ensuring clauses respectively provide the ranges of all inputs, as well as, optionally, initial errors, written as $x +/- 1e-11$. If no errors are specified, roundoff errors are assumed.

Rosa is essentially a “verifying compiler” that checks that the given contract is satisfied and generates code over either floating-point or fixed-point arithmetic, depending on the precision determined in the verification step. In this paper, we focus mainly on the verification portion, and thus will assume that we do the verification against one chosen precision (for example, double precision).
Better Propagation of Errors in Nonlinear Codes

Figure 2 shows the code of a jet engine controller benchmark. The initial errors of 1e-11 model possible noise on the sensors. This example is challenging to analyse because of the complexity of the function, but also because of the large number of correlations between the variables [13]. Fluctuat and Rosa compute an error bound of \(1e^{-11}\) model possible noise on the sensors. This example is challenging to analyse because of the complexity of the function,

\[\text{val } x1 = ((2*x1*(t/(x1*x1 + 1))*}\ \] \[\text{val } x2 = (t/(x1*x1 + 1) - 3) \times x1\times x1(4*(t/(x1*x1 + 1)) - 6))\times \] \[\text{val } x1 + 3\times x1\times x1(\text{t/(x1*x1 + 1)}) + x1\times x1 \times x1 + x1 + \] \[3\times (3\times x1\times x1 + 2\times x2 - x1)/(x1\times x1 + 1))\] \]

Figure 2: Jet engine benchmark

Unbounded Loops: Pendulum

Figure 3 shows a Runge Kutta order 2 simulation of a pendulum, where \(t\) and \(w\) are the angle the pendulum forms with the vertical and the angular velocity respectively. We approximate the sine function with its Taylor series polynomial, and consider two versions: the order 3 and order 7 Taylor approximation. In both cases we focus on roundoff errors between the system following the dynamics given by the polynomial approximation, and the system following the same dynamics but implemented in finite precision.

The precondition now specifies two sets of ranges. \(-2 \leq \ t \leq 2\) constrains the real-valued variable \(t\), whereas \(-2.01 \leq -t\) specifies that its actually implemented finite-precision counterpart remains in a slightly larger range. This latter interval includes all errors including roundoffs and is therefore larger.

We represent loops as recursive functions. After 100 iterations, our tool determines that the error on the result is at most 9.07e-14 when including roundoffs and is therefore larger.

Tackling Discontinuous Programs

Embedded systems often use piece-wise approximations of more complex functions. In Figure 3 we show a possible piece-wise polynomial approximation of the jet engine controller from Figure 2 (we obtained this approximation by fitting a polynomial to a sample of values of the original function). The result is similarly well behaved. At the same time, this constraint also makes the analysis of the loop errors challenging, as does the presence of square root and a large number of variables. The plot from the introduction Figure 1 reports the errors of this benchmark for one specific initial configuration (corresponding to the position of Jupiter). The analysis done by our tool in this example covers all configurations where the planet’s position and velocity coordinates satisfy the constraints in the precondition.

After 100 iterations, our tool computes an error for the position variables \(x, y, z\) of \(1.894e-8, 2.087e-9, 2.635e-8\) respectively. Fluctuat again reports an error of \(\infty\) and Rosa times out due to unrolling.

Tackling Discontinuous Programs

Embedded systems often use piece-wise approximations of more complex functions. In Figure 3 we show a possible piece-wise polynomial approximation of the jet engine controller from Figure 2 (we obtained this approximation by fitting a polynomial to a sample of values of the original function). For many applications it is crucial that the error introduced by the discontinuity remains small, since we may otherwise risk unstable behaviour. The real-valued difference between the two branches is at most 0.21021. However this is not a sound estimate for the discontinuity error in the presence of roundoff and initial errors. With our tool, we can confirm that the discontinuity error is bounded by 0.21202, with all errors taken into account. In contrast, Rosa
3. Propagation of Errors in Nonlinear Arithmetic

The first challenge we adress is error propagation in straight-line nonlinear functions, without loops and branches. We follow the methodology and specification language in Rosa, and concentrate in this exposition on the error computation. The input is a real-valued arithmetic expression representing a function \( f : \mathbb{R}^n \rightarrow \mathbb{R} \) over some inputs \( x_i \in \mathbb{R} \), absolute error bounds on the inputs \( \lambda_i \), and a target precision. The arithmetic operators our tool accepts are \((+,-,\times,\div,\sqrt{\cdot})\).

We denote by \( f \) and \( x \) the exact ideal real-valued function and variables and by \( \hat{f} : \mathbb{R}^n \rightarrow \mathbb{R}, \hat{x} \in \mathbb{R}^n \) their actual finite-precision counter-parts. Note that for our analysis all variables are real-valued; the finite-precision variable \( \hat{x} \) is considered as a noisy versions of \( x \). We want to bound the absolute error on the result of evaluating \( f(x) \) in finite precision arithmetic:

\[
|f(x) - \hat{f}(\hat{x})| \quad \text{where } |x - \hat{x}| \leq \lambda
\]

### 3.1 Separation of Errors

Approaches based on interval or affine arithmetic treat all errors equally in the sense that initial errors are propagated in the same way as roundoff errors which are committed during the computation. We propose to separate these errors as follows:

\[
|f(x) - \hat{f}(\hat{x})| = |f(x) - f(\hat{x}) + f(\hat{x}) - \hat{f}(\hat{x})| \\
\leq |f(x) - f(\hat{x})| + |f(\hat{x}) - \hat{f}(\hat{x})|
\]

The first term captures the error in the result of \( f \) caused by the initial error between \( x \) and \( \hat{x} \). The second term covers the roundoff error committed when evaluating \( \hat{f} \) in finite-precision, but note that we compute this roundoff error from a precise input without initial error. Thus, we separate the overall error into the propagation of existing errors, and the newly commited roundoff errors. Figure 6 illustrates this separation.

We denote by \( \sigma_i : \mathbb{R}^n \rightarrow \mathbb{R} \) the function which returns the roundoff error committed when evaluating an expression in finite-precision arithmetic: \( \sigma_i(\hat{x}) = |f(\hat{x}) - \hat{f}(\hat{x})| \). We omit the subscript \( i \), when it is clear from the context. Further, \( g : \mathbb{R}^n \rightarrow \mathbb{R} \) denotes a function which bounds the difference in \( f \), given a difference in its inputs: \( |f(x) - f(y)| \leq g(|x - y|) \). When \( f \) is multivariate, the absolute value is component-wise, i.e. \( g(|x_1 - y_1|, \ldots, |x_n - y_n|) \), but when it is clear from the context, we will write \( g(|x - y|) \) for clarity. Thus, the overall numerical error is given by:

\[
|f(x) - \hat{f}(\hat{x})| \leq g(|x - \hat{x}|) + \sigma(\hat{x})
\]

### 3.2 Computing Roundoff Errors

We briefly review the roundoff error computation in Rosa [13], which we use as the function \( \sigma \) without modification. Rosa represents roundoff errors as an affine form

\[
\hat{x} = x_0 + \sum_{i=1}^{k} x_i \epsilon_i, \quad \epsilon_i \in [-1, 1]
\]

where each \( x_i \) represents the magnitude of a deviation from the central value \( x_0 \). For each arithmetic operation, Rosa adds a new term \( x_{k+1} \epsilon_{k+1} \) with \( x_{k+1} \) the magnitude of the roundoff error committed at that operation. Existing errors are propagated with the standard rules of affine arithmetic, for details see [13][14]. The total error represented by an affine form is the maximum absolute value of the interval

\[
[x_o - \text{rad}(\hat{x}), x_o + \text{rad}(\hat{x})], \quad \text{rad}(\hat{x}) = \sum_i |x_i|
\]

### 3.3 Computing Propagation Coefficients

We instantiate Equation 2 with \( g(x) = K \cdot x \) and bound the deviation on the result by a linear function in the input errors:

\[
|f(x) - f(y)| \leq K|\epsilon|
\]

We will use this definition for most of the rest of this paper. The constant \( K \) is to be determined for each function individually, and is usually called the Lipschitz constant. We will also use the, in this context, more descriptive name propagation coefficient. Note that we need to compute the propagation coefficient \( K \) for the mathematical function \( f \) and not its finite-precision counterpart \( \hat{f} \).

At a high level, error amplification or diminution depends on the derivative of the function at the value of the inputs. The steeper the function, the more the errors are magnified. We formally derive the computation of the propagation coefficients \( K_i \) for a multivariate function \( f \) in the following.
Let \( h : [0, 1] \to \mathbb{R} \) such that \( h(\theta) := f(y + \theta(z - y)) \). Without loss of generality, assume \( y < z \). Then \( h(0) = f(y) \) and \( h(1) = f(z) \) and

\[
\frac{d}{d\theta} h(\theta) = \nabla f(y + \theta(z - y)) \cdot (z - y)
\]

By the mean value theorem:

\[
f(z) - f(y) = h(1) - h(0) = h'(\zeta) \quad \text{where} \quad \zeta \in [0, 1]
\]

\[
|f(z) - f(y)| = |h'(\zeta)| = |\nabla f(y + \zeta(z - y)) \cdot (z - y)|
\]

\[
= \left| \left( \frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n} \right) \cdot (z - y) \right|, \quad w = y + \zeta(z - y)
\]

\[
= \left| \frac{\partial f}{\partial x_1} \cdot (z_1 - y_1) + \cdots + \frac{\partial f}{\partial x_n} \cdot (z_n - y_n) \right|
\]

\[
\leq \left| \frac{\partial f}{\partial x_1} \right| \cdot |z_1 - y_1| + \cdots + \left| \frac{\partial f}{\partial x_n} \right| \cdot |z_n - y_n| \quad (**)
\]

where the partial derivatives are evaluated at \( w = y + \zeta(z - y) \), which we omit for readability. We compute the partial derivatives symbolically, but before we can evaluate the propagation error, we need to determine over which inputs we need to evaluate them.

**Bounding Ranges of Partial Derivatives** In the above, the value of \( w \) is constraint to be in \([y, z]\), so for a sound analysis we have to determine the maximum absolute value of the partial derivative over \([y, z]\), \( y \) and \( z \) in our application range over the values of \( x \) and \( \tilde{x} \) respectively, so we compute the maximum absolute value of \( \frac{\partial f}{\partial x_i} \) over the interval that contains the intervals of \( x \) and \( \tilde{x} \). Both interval and affine arithmetic suffer from possibly large over-approximations, which is why we use the range computation from Rosa [13] to bound the ranges of the derivatives. This procedure relies on Z3 to narrow down initial ranges precomputed by interval arithmetic. Finally, we have \( |y_i - z_i| \leq \lambda_i \), so that we obtain

\[
|f(x) - f(\tilde{x})| \leq \sum_{i=1}^{n} K_i \lambda_i \quad (3)
\]

where \( K_i = \sup_{x, \tilde{x}} \left| \frac{\partial f}{\partial x_i} \right| \).

**Additional Constraints** The propagation coefficients are computed using the input ranges. As we have seen in Figure 4, we can often restrict the inputs further by additional constraints such as \( x \times x + y \times y + z \times z \gg \theta \). Since we are using an SMT solver to bound the ranges of the partial derivatives, these additional constraints can naturally be taken into account to compute more precise propagation coefficients.

**Sensitivity to Input Errors** Beyond providing a way to compute the propagated initial errors, Equation 3 also makes the sensitivity of the function to input errors explicit, at least to a linear approximation. That is, we can read off from Equation 3 how much initial errors get magnified or diminished by the computation. The user can use this knowledge, for example, to determine which inputs need to be determined more precisely, e.g. by more precise measurements. We report the values of \( K \) back to the user.

### 3.4 Higher Order Taylor Approximation

In Subsection 3.3 we presented one possible instantiation of the error propagation function \( g \). The resulting propagation function is a function in the input errors. The errors do, however, also depend on the ranges of the inputs. This fact is only implicitly reflected in the computed coefficients via the ranges used for bounding the partial derivatives.

We can in fact make this relationship more explicit. Recall Taylor’s Theorem in several variables:

**Taylor’s Theorem** Suppose \( f : \mathbb{R}^n \to \mathbb{R} \) is of class \( C^{k+1} \) on an open convex set \( S \). If \( \alpha \in S \) and \( \alpha + h \in S \), then

\[
f(\alpha + h) = \sum_{|\alpha| \leq k} \frac{\partial^m f(\alpha)}{\alpha!} h^\alpha + R_{\alpha,k}(h)
\]

where the remainder in Lagrange’s form is given by:

\[
R_{\alpha,k}(h) = \sum_{|\alpha| = k+1} \frac{\partial^m f(\alpha + ch)}{\alpha!} h^\alpha
\]

for some \( c \in (0, 1) \).

Computing the Taylor expansion of \( f(\tilde{x}) \) to first order in our setting:

\[
f(\tilde{x}) = f(x) + \sum_{j=1}^{n} \partial_j f(x) h_j + \frac{1}{2} \sum_{j,k=1}^{n} \partial_j \partial_k f(w) h_j h_k
\]

\[
|f(\tilde{x}) - f(x)| \leq \sum_{j=1}^{n} \partial_j f(x) h_j + \frac{1}{2} \sum_{j,k=1}^{n} H_{j,k}(w) h_j h_k
\]

where \( w \) is in the interval containing \( x \) and \( \tilde{x} \), and \( H \) is the Hessian matrix of \( f \). If we consider the expansion for \( k = 1 \), we obtain an expression for computing the upper bound on the propagated error which is also a function of the input values.

We observe that the second order taylor remainder is, in general, small, due to the fact that we take the square of the initial errors, which we assume to be small in our applications. We can bound the remainder with the same technique we use to compute the propagation coefficients. Then, together with the partial derivatives of \( f \), we obtain error specifications which can be used for a more precise modular verification process.

**Application to Interprocedural Analysis** Having more precise specifications enables us to re-use methods across different call-sites, with possibly different constraints on the arguments. We present an example in [subsection 6.1] which demonstrates the effectiveness of this summarization technique. We are not aware of other work that is capable of computing such summaries for numerical errors. [21] presents an approach to compute method summaries based on affine arithmetic evaluation and instantiation. These summaries, however, capture the real-valued ranges only and not the numerical errors.

### 4. Loops

We now apply the propagation of errors idea to loops. We want to compute the overall error after \( m \)-fold iteration \( f^m \) of \( f \). We define for any function \( H : H^0(x) = x, H^{m+1}(x) = H(H^m(x)) \). We are thus interested in bounding:

\[
|f^m(x) - \hat{f}^m(\tilde{x})|
\]

\( f, g \) and \( \sigma \) are now vector-valued: \( f, g, \sigma : \mathbb{R}^n \to \mathbb{R}^n \), because we are nesting the potentially multivariate function \( f \). In essence, we want to compute the effect of iterating Equation 2. We define

**Theorem:** Let \( g \) be such that \( |f(x) - f(y)| \leq g(|x - y|) \), it satisfies \( g(x + y) \leq g(x) + g(y) \) and is monotonically. Further, \( \sigma \) and \( \lambda \) satisfy \( \sigma(\tilde{x}) = |f(\tilde{x}) - \hat{f}(\tilde{x})| \) and \( |x - \tilde{x}| \leq \lambda \). The absolute value is taken component-wise. Then the numerical error after \( m \) iterations is given by

\[
|f^m(x) - \hat{f}^m(\tilde{x})| \leq g^m(|x - \tilde{x}|) + \sum_{i=0}^{m-1} g^i(\sigma(\hat{f}^{m-i-1}(\tilde{x})))
\]

(4)
Proof: We show this by induction. The base case $m = 1$ has already been covered in [subsection 3.1]. By adding and subtracting $f(\hat{f}^m(\tilde{x}))_1$ we get

\[
\left| f^m(x)_1 - \hat{f}^m(\tilde{x})_1 \right| \leq \sum_{i=1}^{m-1} K_i^\lambda + \sum_{i=1}^{m-1} K_i^\sigma + \sigma \left( f(\hat{f}^{m-1}(\tilde{x}))_1 \right)
\]

We instantiate the propagation function \( \lambda \) sound results. When \( m = 1 \), we obtain a closed-form for the expression \( \sigma \). Then we obtain a closed-form for the expression \( \sigma \) and similarly obtain \( K \).

Suppose we can compute \( K \) from the \( m \)th iteration propagated through the remaining iterations. Then we obtain a closed-form for the expression

\[
\left| f^m(x)_n - \hat{f}^m(\tilde{x})_n \right| = \sum_{i=1}^{m-1} K_i^\lambda + \sum_{i=1}^{m-1} K_i^\sigma + \sigma \left( f(\hat{f}^{m-1}(\tilde{x}))_n \right)
\]

Applying the definitions of \( g \) and \( \sigma \)

\[
\leq g \left( \left| f^{m-1}(x)_1 - \hat{f}^{m-1}(\tilde{x})_1 \right| \right) + \sigma \left( f^{m-1}(\tilde{x}) \right)
\]

then using the induction hypothesis and monotonicity of \( g \).

\[
\leq g \left( g^{m-1}(\tilde{\lambda}) + \sum_{i=1}^{m-2} g^i(\sigma(\hat{f}^{m-1}(\tilde{x}))) \right) + \sigma(\hat{f}^{m-1}(\tilde{x}))
\]

then using \( g(x + y) \leq g(x) + g(y) \), we finally have

\[
\leq g^{m}(\tilde{\lambda}) + \sum_{i=1}^{m-1} g^i(\sigma(\hat{f}^{m-1}(\tilde{x}))) + \sigma(\hat{f}^{m-1}(\tilde{x}))
\]

\[
= g^{m}(\tilde{\lambda}) + \sum_{i=0}^{m-1} g^i(\sigma(\hat{f}^{m-1}(\tilde{x})))
\]

In words, the overall error after \( m \) iterations can be decomposed into the initial error propagated through \( m \) iterations, and roundoff error from the \( m \)th iteration propagated through the remaining iterations.

### 4.1 Closed Form Expression

We instantiate the propagation function \( g \) as before and would like to derive a closed-form expression for the error, as [Equation 4] is not very evaluation friendly. In fact, evaluating [Equation 4] as given, with a fresh set of propagation coefficients for each iteration \( i \) amounts to loop unrolling, but with a loss of correlation between each loop iteration. Suppose we can compute \( K \) as a matrix of propagation coefficients, and similarly obtain \( \sigma(f) = \sigma \) as a vector of constants, both valid over all iterations. Then we obtain a closed-form for the expression of the error:

\[
|f^m(x) - \hat{f}^m(\tilde{x})| \leq K^m \lambda + \sum_{i=1}^{m-1} K^i \sigma + \sigma
\]

where \( \lambda \) is the vector of initial errors. If \( (I - K)^n \) exists,

\[
|f^m(x) - \hat{f}^m(\tilde{x})| \leq K^m \lambda + ((I - K)^{-1}(I - K^m)) \sigma
\]

We obtain \( K^m \) with power-by-squaring and compute the inverse with the Gauss-Jordan method with rational coefficients to obtain sound results. When \( K = I \), \( g \) becomes the identity function and so

\[
|f^m(x) - \hat{f}^m(\tilde{x})| \leq \lambda + \sum_{i=1}^{m-1} \sigma + \sigma = \lambda + m \cdot \sigma
\]

Now the question remains, how to determine \( K \) and \( \sigma \). As before, this boils down to determining the ranges of the variables \( x, \tilde{x} \), over which to compute the coefficients of \( K_{ij} = \sup_{x, \tilde{x}} \frac{\partial f_i}{\partial x_j} \) and which to use for the roundoff error computation. We consider two cases.

### Inductive Constant Ranges

When the ranges of the variables of the loop are inductive, that is, both the real-valued and the finite-precision values remain within the initial ranges, then these are clearly the ranges for the computation of \( K \) and roundoffs \( \sigma \). We require the user to specify both the real-valued ranges of variables (e.g., \( a \ll x \& \& x \ll b \)) as well as the actual finite-precision ones (\( c \ll -x \& \& -x \ll d \)). We also require that the actual ranges always include the real ones \((a, b) \subseteq [c, d]\), hence it is the actual ranges \((c, d)\) that are used for the computation of \( K \) and \( \sigma \).

### Iteration-dependent Ranges

For many loops however, inductive constant ranges either do not exist or are very hard to prove. Or it may be that only the real-valued ranges are inductive, but, because of roundoff errors, the actual ones are not. We still want to analyse these loops, but it is clear that the validity of the computed \( K, \sigma \) and final errors is limited to the validity of the ranges which we use for their computation.

Our tool attempts to verify that the range bounds are valid, i.e. inductive. Should it not succeed, it will nonetheless perform the error computation with the user-given actual ranges. The generated code will, however, include the precondition \( \text{require}(c \ll x \& \& x \ll d) \), which, in Scala, is checked at runtime. We believe that this is a reasonable compromise, as these assertions are fast to check and in many applications ranges do stay bounded.

### 5. Discontinuity

We now turn to the third challenge, errors due to discontinuities. Recall the piece-wise jet engine approximation from [Figure 5]. Due to the initial errors on \( x \) and \( y \), the real-valued computation may take a different branch than the finite-precision one, and thus produce a different result. We call this difference the discontinuity error; it is the difference between the real valued and the finite-precision computation when they take different branches. Note that we do not assume smoothness or continuity or any other special characteristic of the functions.

Previous approaches construct a constraint encoding the difference between the real value computed by one branch and the finite-precision value computed by the other. The other direction is handled symmetrically. They differ in how they handle the constraints introduced by the branch condition. Fluctuat constrains the noise terms of the real and floating-point computation in its abstract domain based on a logical product with the interval domain [20]. Rosa essentially constructs one constraint that encodes the computation along both paths and the correlation between the variables of these two paths. The resulting difference is refined with the Z3 SMT solver. Fluctuat’s approach becomes quickly imprecise when the functions are not linear due to the underlying domain. Rosa’s approach produces very precise but complex constraints which work nicely for unary functions, but are hard to handle beyond those. In this section, we show how to apply the separation of errors idea and overcome the limitations of the previous techniques.

#### 5.1 Applying Separation of Errors

Using our previous notation, let us consider a function with a single branch statement like in the example above and let \( f_1 \) and \( f_2 \) be the real-valued functions corresponding to the \( \text{if} \) and the \( \text{else} \) branch respectively. Then, the discontinuity error is given by \( |f_1(x) - f_2(\tilde{x})| \), i.e. the real computation takes branch \( f_1 \), and the finite-precision one \( f_2 \). The opposite case is analogous. We again
We expect the individual parts to be easier to handle for the underlying SMT-solver, since we reduce the number of variables and correlations. Fluctuat and Rosa compute the discontinuity error as one difference between the computations on the two paths of a branch. In contrast in the presented work, we split the error and compute its parts separately, obtaining a more scalable approach. We use the same procedure as in section 3.

We perform our test mostly with double precision, unless otherwise specified, as this is a common choice for numerical programs. Note however, that our tool supports both floating-point arithmetic with different precisions, as well as fixed-point arithmetic with different bitlengths. In our experience, while the absolute errors naturally exhibit a divergence between the real-valued and the finite-precision arithmetic. We use the same procedure as in section 3. Our tool accepts any branch condition that is of the form \(\text{if } (x) \text{ then } \cdots \text{ else } \cdots\). In practice, we will merge the two error variables and only add one to the constraint: \(\delta = \delta_1 + \delta_2\). The procedure for other branch conditions is analogous.

We create such a constraint both for the variables representing finite-precision values (\(\tilde{x}\)), as well as the real-valued ones \(x\) and use them as additional constraints when computing the individual parts of equation (5):

The constraints just described essentially express the correlations between \(x\) and \(\tilde{x}\), but they do not necessarily narrow down their overall ranges. These are, however, important for the computation of the roundoff errors (third part of equation (5)), since these depend directly on the ranges. If the branch condition is a "range constraint", that is, if it is of the form \(x < c\), where \(c\) is a constant, then we use these constraints to narrow the variable ranges. If, however, the branch condition is a relative condition such as \(x < y\), then the ranges of \(x\) and \(y\) remain unconstrained, i.e. \(x\) and \(y\) individually can still take all the values in their input ranges. In this case, we have to accept the resulting over-approximation, but we found that, in general, the contribution of the roundoff error term to be relatively small anyway.

6. Experimental Evaluation

We have chosen a number of benchmarks from the domains of scientific computing and embedded systems to evaluate our techniques.

We perform our test mostly with double precision, unless otherwise specified, as this is a common choice for numerical programs. Note however, that our tool supports both floating-point arithmetic with different precisions, as well as fixed-point arithmetic with different bitlengths. In our experience, while the absolute errors naturally change with varying precisions and data types, the relative differences in comparisons remain very similar.

We compare our results against those obtained by Rosa and Fluctuat. These are the only available tools that we are aware of that can compute sound numerical error bounds. Unless otherwise stated, all numerical error values are rounded. Experiments were performed on a desktop computer running Ubuntu 12.04.4 with a 3.5GHz i7 processor and 16GB of RAM.

6.1 Straight-line Nonlinear Computation

We first evaluate our new error propagation technique for straight-line code on a number of benchmarks from [13]. The results are summarized in Figure 8. The error computations in Rosa and in Fluctuat are very similar, and essentially differ only in how the ranges of variables are constrained (logical product with an abstract domain or SMT solver). The initial errors in the top benchmarks of Figure 8 are roundoff errors only, in the bottom section we add an initial absolute error of \(1e-11\) to all inputs. We can see that our new technique can improve the computed error estimates in most cases. Even for benchmarks where the initial errors are only roundoffs, we can still sometimes

---

The benchmarks are available as supplementary material.
which is often the formulation produced by code generation tools, 

\[ u, v, T \]

are non-refactored doppler benchmark, the computed coefficients for sensitivity degrade the computed results. Fine-grained steps would increase the running time unnecessarily or valid only for expressions defined as computation. Our tool currently performs the error propagation our technique and thus improve the overall bounds even further. 

\[
\text{val } \text{tmp} = 331.4 + 0.6 * T
\]

\[
- (\text{tmp} * v) / ((\text{tmp} + u) * (\text{tmp} + u))
\]

In the second case, we apply the error propagation twice, once for computing the error on \( \text{tmp} \) and once for the error on the result. The hope is to compute intermediate values more precisely with our technique and thus improve the overall bounds even further. The experimental results confirm the benefit of this step-wise error computation. Our tool currently performs the error propagation only for expressions defined as \text{val} or final expressions, as too fine-grained steps would increase the running time unnecessarily or degrade the computed results.

**Sensitivity** The propagation coefficients provide information about the sensitivity of a function to input errors. For example, for the non-refactored doppler benchmark, the computed coefficients for \( u, v, T \) are

3.216238, 0.006881929, 0.7557531

![Figure 8: Comparison of computed absolute errors with our new Lipschitz constant-based approach against state of the art on straight-line, nonlinear benchmarks. We mark the best result per benchmark in bold.](image)

![Figure 9: Running times of our approach compared with Rosa (in seconds).](image)

respectively. Thus, absolute input errors on \( u \) get magnified by approximately 3.22 in the worst case, whereas input errors on \( v \) have a much more favourable factor of below one. This information can, for instance, be used to direct optimization efforts, and we report the computed propagation coefficients in comments in the generated code.

**Running times** Figure 9 compares the running times of our new technique against the runtimes of Rosa for selected benchmarks. Fluctuat in general computed the result within one second, since it is not using an SMT solver internally. We have not specifically optimized our implementation, and improvements are certainly possible. Nevertheless, while our two-step computation clearly increases the runtime, we believe that the times do remain acceptable for a static verification approach.

**First-order Method Summaries** Section 3.4 introduced a possible extension of the propagation coefficients to postconditions where the errors are functions of both the initial errors and the ranges of the corresponding variable. Here we give a possible scenario how these `Taylor summaries` can be used. The verification framework in [13] is modular in that each method is verified separately, and method postconditions are used, where possible, at call sites. The specifications have to be general however, to allow a method to be used in many instances, yet precise enough to facilitate a successful verification.

For example, consider the following seventh order approximation to the sine function, as it may be used in an embedded system, where trigonometric functions are often approximated.

\[
\text{def sine}(\text{Real}): \text{Real} = \{ \\
\text{require}(-3.5 < x \&\& x < 3.5 \&\& x +/- 1e-8) \\
\text{x} - (xx*x*x*x)/6.0 + (xx*x*x*x*x*x)/120.0 - (xx*x*x*x*x*x*x*x)/5040.0 \\
\text{ensuring}(\text{res} \Rightarrow -1.0 < \text{res} \&\& \text{res} < 1.0 \&\& \text{res} +/- 2e-7)
\}
\]

The postcondition is successfully verified for the given range and input error. But what if, at a call site, the range or the initial error is smaller?
Consider two calls to `sine`:

```plaintext
require(-0.5 <= y && y <= 0.5 && y +/- 1e-8) ...
sineTaylor(y)

require(-3.0 <= z && z <= 1)
...
sineTaylor(z)
```

With Rosa, one can either use the postcondition with given error on the result of 2e-7, or inline the function and essentially re-do the error computation. In contrast, our approach described in subsection 3.4 will instead use the computed summaries and determine the error (correctly) fails and our tool reports a counter-example. Thus, our nextItem on these kinds of loops, due to the unboundedly growing errors. In abstract interpretation with the domains in Fluctuat does not stabilize using complete unrolling of loops, because we have found that substantial number of iterations.

### 6.2 Loops

We evaluate our proposed technique for the case of loops on a number of examples which demonstrate several features of our system.

**Newton-Raphson Method** We begin with an example of a loop in which we wish to show that the value of a variable always remains in a certain range. Such a property is important, for instance, in the case of iterative algorithms and controllers, where unboundedness suggests that the system diverges. The following function taken from [10] implements a Newton-Raphson approximation. (We abbreviate e.g. \( xxxx \) as \( x^4 \)).

```plaintext
def newton(x: Real, k: LoopCounter): Real = {
  require(-1.0 < x && x < 1.0 && -1.0 < -x && -x < 1.0)
  if (k < 10) {
    newton(x - (x - (x^4)/6.0 + (x^3)/120.0 + (x^2)/5040.0) /
        (1 - (x*x))/2.0 + (x^4)/24.0 + (x^5)/720.0), k + 1)
  } else {
    x
  }
} ensuring (res => -1.0 < res && res < 1.0 &&
  -1.0 < -res && -res < 1.0)
```

Note that the precondition is also the loop invariant we wish to check. Our tool can automatically verify that this specification is inductive, also in the presence of roundoff errors. Fluctuat, relying on affine arithmetic cannot prove that even one iteration remains in the given bound, and applying it to an unbounded loop produces \( \infty \) as the error bound. Our result holds for any number of iterations and does not rely on unrolling. If we were to change the range specification to \( -1.2 < x && x < 1.2 && -1.2 < -x && -x < 1.2 \), the verification (correctly) fails and our tool reports a counter-example. Thus, our system can be used to establish boundedness of values in loops even if the number of iterations is unknown at compile time.

**Running Average** Now we turn to numerical error estimation. Figure 10 shows the implementation of an online computation of the average of numbers coming from the range \([-1200, 1200]\). The method `nextItem` models a fresh value from an undetermined source. Round-off errors allow the computed value to go outside of \([-1200, 1200]\), we thus use the actual range \([-1200.5, 1200.5]\) for the error computation. We chose this range optimistically, to cover a substantial number of iterations.

Figure 10 compares the errors and runtimes computed by our tool against the errors determined by Fluctuat. We compare with Fluctuat using complete unrolling of loops, because we have found that abstract interpretation with the domains in Fluctuat does not stabilize on these kinds of loops, due to the unboundedly growing errors. In contrast, our system can discover parametric bounds that grow as a function of the loop iteration. In particular, the overall error is given as

\[
K^m \lambda + ((I - K)^{-1}(I - K^m)) \sigma
\]

where our tool determines \( K = 0.999001996, \sigma = 1.5082e-10 \) and \( \lambda = 1.1369e-13 \) for example for the case of no added initial errors. Rosa does not finish loop unrolling even for small numbers of iterations in reasonable time, hence we do not list it here. In this example, the computation depends on the loop counter \( n \), making the constraints different for different loop bounds and thus different bounds on \( n \). This may make the constraint more or less hard to solve for \( Z_3 \), which results in the, maybe surprising, variation in running times.

The comparison in Figure 11 shows the tradeoff our technique makes between precision and scalability. Recall that our proposed technique makes two approximations: firstly, it approximates the effect of each loop iteration in isolation, potentially loosing correlation information and secondly, the propagation coefficients are computed across the whole input range. While Fluctuat, performing loop unrolling and keeping correlations computes smaller error bounds, our
we have a choice of two different approximations for the sine errors, the final loop bound computation is fast.

iterations in this case, and using our closed-form expression for the constant, since the propagation coefficient matrix is the same for all computation time grows with the number of iterations, our time is compute meaningful error bounds. Also note, that while Fluctuat's larger numbers of iterations, our tool is the only one that can still times, although our bounds are more precise nearly throughout. For Fluctuat is able to compute error bounds for smaller running times, against the results obtained by our tool. Rosa's Figure 12 compares computed error bounds by Fluctuat, again with preferable with respect to numerical errors.

Notice that the latter numbers are smaller, in two cases by an order of magnitude, which strongly suggests that this approximation is


\[ x^2 + y^2 + z^2 \geq 15 \]

\[ x \geq 8.47e-07 \]

\[ y \geq 1.17e-07 \]

\[ z \geq 9.19e-07 \]

As the table shows, the over-approximation is modest and our tool still computes meaningful results.

To compare to the simulation from the introduction, the errors on \( x, y, z \) after 100 iterations were on the order of 1e-14. These errors are however, obtained for one specific run, whereas our tool performs a worst-case analysis for a large number of runs at the same time. To our knowledge, our tool is the only one that can handle programs of this complexity. Fluctuat returns with an error bound of \([ -\infty, \infty] \), and Rosa again does not finish unrolling in a reasonable time.

6.3 Discontinuity Errors

We compare the results of our technique to compute discontinuity errors against those implemented in Fluctuat and Rosa in Figure 13. The first part of the table contains unary functions from which we added a benchmark based on a piece-wise defined cubic spline. We observe that Rosa indeed can leverage its more precise constraint and compute tighter bounds.

The second half of the table presents results for benchmarks in two variables. We have derived these benchmarks by piece-wise approximating a complex function, a common pattern seen in

<table>
<thead>
<tr>
<th># iter</th>
<th>Fluctuat time</th>
<th>Our tool time</th>
</tr>
</thead>
<tbody>
<tr>
<td>order 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1.46e-15 0.04</td>
<td>1.47e-15 4</td>
</tr>
<tr>
<td>10</td>
<td>2.88e-15 0.13</td>
<td>2.87e-15 4</td>
</tr>
<tr>
<td>15</td>
<td>4.53e-15 0.39</td>
<td>4.44e-15 4</td>
</tr>
<tr>
<td>20</td>
<td>6.51e-15 0.89</td>
<td>6.19e-15 4</td>
</tr>
<tr>
<td>25</td>
<td>8.98e-15 2</td>
<td>8.15e-15 4</td>
</tr>
<tr>
<td>50</td>
<td>5.07e-14 16</td>
<td>2.21e-14 4</td>
</tr>
<tr>
<td>100</td>
<td>\infty -</td>
<td>9.07e-14 4</td>
</tr>
<tr>
<td>250</td>
<td>\infty -</td>
<td>3.11e-12 4</td>
</tr>
<tr>
<td>500</td>
<td>\infty -</td>
<td>9.58e-10 4</td>
</tr>
<tr>
<td>1000</td>
<td>\infty -</td>
<td>9.02e-05 4</td>
</tr>
<tr>
<td>order 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1.47e-15 0.06</td>
<td>1.47e-15 8</td>
</tr>
<tr>
<td>10</td>
<td>2.92e-15 0.35</td>
<td>2.88e-15 8</td>
</tr>
<tr>
<td>15</td>
<td>4.68e-15 1</td>
<td>4.45e-15 8</td>
</tr>
<tr>
<td>20</td>
<td>6.95e-15 3</td>
<td>6.21e-15 8</td>
</tr>
<tr>
<td>25</td>
<td>1.01e-14 5</td>
<td>8.18e-15 8</td>
</tr>
<tr>
<td>50</td>
<td>2.43e-13 49</td>
<td>2.21e-14 8</td>
</tr>
<tr>
<td>100</td>
<td>\infty -</td>
<td>8.82e-14 8</td>
</tr>
<tr>
<td>250</td>
<td>\infty -</td>
<td>2.67e-12 8</td>
</tr>
<tr>
<td>500</td>
<td>\infty -</td>
<td>6.54e-10 8</td>
</tr>
<tr>
<td>1000</td>
<td>\infty -</td>
<td>3.89e-05 8</td>
</tr>
</tbody>
</table>

Figure 12: Comparison of absolute errors on \( t \) and running times computed by Fluctuat and our tool on the pendulum benchmark. Rosa times out. Time is measured in seconds.

Pendulum Recall the pendulum simulation from Figure 3 where we have a choice of two different approximations for the sine function. During the analysis, our tool determines the following propagation coefficient matrix \( K \) for the order 3 approximation:

\[
\begin{bmatrix}
1.0002500818333124 & 0.01 \\
0.05250059956010460 & 1.0002625029978005
\end{bmatrix}
\]

and for the order 5 approximation:

\[
\begin{bmatrix}
1.00008334411850905 & 0.01 \\
0.04903325006220655 & 1.0000872967293692
\end{bmatrix}
\]

Notice that the latter numbers are smaller, in two cases by an order of magnitude, which strongly suggests that this approximation is preferable with respect to numerical errors.

Figure 13 compares computed error bounds by Fluctuat, again with loop unrolling, against the results obtained by our tool. Rosa's loop unrolling again fails to provide results in acceptable times. Fluctuat is able to compute error bounds for smaller running times, although our bounds are more precise nearly throughout. For larger numbers of iterations, our tool is the only one that can still compute meaningful error bounds. Also note, that while Fluctuat’s computation time grows with the number of iterations, our time is constant, since the propagation coefficient matrix is the same for all iterations in this case, and using our closed-form expression for the errors, the final loop bound computation is fast.

Figure 13: Comparison of computed absolute errors with our new Lipschitz constant-based approach against state of the art on benchmarks with discontinuities.

2-body Simulation Finally, recall the simulation of Jupiter orbiting the Sun in Figure 4. The precondition contains the constraint \( x^2 + y^2 + z^2 \geq 20.0 \) which limits the possible orbits we want to do the analysis for. The following table compares the final errors computed for the given constraint and for the case when we relax the condition to \( x^2 + y^2 + z^2 \geq 15.0 \). As the number of possible cases our tool has to consider is larger, we also expect the over-approximation committed during the analysis to grow.

As the table shows, the over-approximation is modest and our tool still computes meaningful results.

To compare to the simulation from the introduction, the errors on \( x, y, z \) after 100 iterations were on the order of 1e-14. These errors are however, obtained for one specific run, whereas our tool performs a worst-case analysis for a large number of runs at the same time. To our knowledge, our tool is the only one that can handle programs of this complexity. Fluctuat returns with an error bound of \([ -\infty, \infty] \), and Rosa again does not finish unrolling in a reasonable time.

<table>
<thead>
<tr>
<th>benchmark</th>
<th>Fluctuat</th>
<th>Rosa</th>
<th>Our tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>simpleInterpolater</td>
<td>3.45e-05</td>
<td>2.346e-05</td>
<td>3.401e-05</td>
</tr>
<tr>
<td>squareRoot</td>
<td>0.0394</td>
<td>0.02365</td>
<td>0.02382</td>
</tr>
<tr>
<td>squareRoot3 invalid</td>
<td>0.429</td>
<td>1.308e-09</td>
<td>1.313e-09</td>
</tr>
<tr>
<td>cubic spline</td>
<td>12.01</td>
<td>1.499e-15</td>
<td>1.499e-15</td>
</tr>
<tr>
<td>linear fit</td>
<td>1.721</td>
<td>0.6374</td>
<td>0.6374*</td>
</tr>
<tr>
<td>quadratic fit</td>
<td>10.61</td>
<td>3.218</td>
<td>0.2548*</td>
</tr>
<tr>
<td>quadratic fit (0.1)</td>
<td>11.25</td>
<td>3.226</td>
<td>0.2904</td>
</tr>
<tr>
<td>quadratic fit2 (X)</td>
<td>0.6322</td>
<td>9.195e-16</td>
<td>1.094e-15</td>
</tr>
<tr>
<td>quadratic fit2 (X, 0.1)</td>
<td>0.7781</td>
<td>0.05583</td>
<td>0.08554</td>
</tr>
<tr>
<td>styblinski</td>
<td>223.5</td>
<td>70.41</td>
<td>0.6320*</td>
</tr>
<tr>
<td>styblinski (0.1)</td>
<td>239.91</td>
<td>71.01</td>
<td>3.250</td>
</tr>
<tr>
<td>styblinski2 (X)</td>
<td>30.495</td>
<td>18.69</td>
<td>3.665*</td>
</tr>
<tr>
<td>styblinski2 (X, 0.1)</td>
<td>33.19</td>
<td>19.36</td>
<td>4.863</td>
</tr>
<tr>
<td>jetApprox</td>
<td>25.4</td>
<td>10.91</td>
<td>0.1702*</td>
</tr>
<tr>
<td>jetApprox - good fit (X)</td>
<td>5.73</td>
<td>4.255</td>
<td>0.2121*</td>
</tr>
<tr>
<td>jetApprox - bad fit (X)</td>
<td>15.32</td>
<td>3.822</td>
<td>1.358*</td>
</tr>
</tbody>
</table>

Finally, note that while Fluctuat’s loop unrolling again fails to provide results in acceptable times. Fluctuat is able to compute error bounds for smaller running times, although our bounds are more precise nearly throughout. For larger numbers of iterations, our tool is the only one that can still compute meaningful error bounds. Also note, that while Fluctuat’s computation time grows with the number of iterations, our time is constant, since the propagation coefficient matrix is the same for all iterations in this case, and using our closed-form expression for the errors, the final loop bound computation is fast.
work in the area of numerical error estimation for straight-line code with and without branches.

Abstract Interpretation We are not aware of other techniques beyond Fluctuat in the domain of abstract interpretation that can quantify numerical errors including roundoffs. Abstract domains exist however, that are sound with respect to floating-points and that can be used to prove the absence of runtime errors such as division by zero [5][11][29], but they do not characterize the magnitude of errors. The advantage of quantifying errors is that, once we have ran the analysis and obtained good bounds, we can transfer many forms of subsequent reasoning about real-valued functions to the concrete software implementation. Such separation of concerns was successfully applied to reason about the stability of control systems [3].

Testing Testing is a suitable approach to generate inputs which violate precision requirements or to establish error bounds which need to be only probabilistically correct. [30] tests numerical programs by perturbation of low-order bits and rewriting, whereas the CADNA approach [33] perturbs the rounding modes. A common theme is to run a higher-precision program alongside the original one. [5] does so by instrumentation, [30] generates constraints which are then discharged with a floating-point arithmetic solver and [12] developed a guided search to find inputs which maximize errors. Further approach include [25], which uses instrumentation to detect cancellation and thus loss of precision. [31] uses user-defined inputs as test vectors to determine approximate error bounds which are then used to optimize program efficiency by choosing lower floating-point precision data types. While our implementation is currently limited to a fixed precision for the whole program, the techniques work, in principle, for mixed precision as well, and could be used to certify programs returned from optimization procedures.

Decision Procedures Floating-point decision procedures are an alternative to prove properties about finite-precision programs. Floating-points have been formalized in the SMT-LIB format [32], and approaches exist which deal with the prohibiting complexity of bit-precise techniques via approximations [9][22]. A combination of theories is problematic, and we are not aware of an approach that is able to quantify the deviation of finite-precision computations with respect to reals.

Theorem Proving Floating-point precision assertions can also be proven using an interactive theorem prover [4][7][23][26]. These tools can reason about ranges and errors of finite-precision implementations, but target specialized and precise properties, which, in general, require an expert user and interactively guiding the proof. Such tools can prove already given error bounds, whereas our tool fully automatically computes them from the initial conditions. Furthermore, the techniques also rely on range arithmetics to propagate errors and could, in principle, also profit from the techniques presented in this paper.

Our analysis for floating-point precisions relies on the hardware and software conformance to the IEEE 754 standard. Furthermore, we rely on the compiler to not have bugs or re-order arithmetic operations arbitrarily. CompCert is a verified C compiler and [8] presents a recent extension of the proof floating-point arithmetic as well and thus able to ensure that our assumptions are valid. While our tool currently generates only Scala code, this is not a fundamental restriction and our analysis technique equally applies to IEEE 754 compliant C programs.

Discontinuity Errors [10] develops a framework for showing programs robust in the sense of k-Lipschitz continuity and [17] relaxes this strict definition of robustness to programs with specified uncertainties and presents a framework for proving while-loops with a particular structure robust. Neither of these approaches quantifies

---

<table>
<thead>
<tr>
<th>benchmark</th>
<th>Rosa</th>
<th>Our tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>simpleInterpolator (float)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>squareRoot (F)</td>
<td>3</td>
<td>22</td>
</tr>
<tr>
<td>squareRoot3</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>squareRoot3 invalid</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>cubic spline</td>
<td>14</td>
<td>17</td>
</tr>
<tr>
<td>natural spline</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>linear fit</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>quadratic fit</td>
<td>45</td>
<td>54</td>
</tr>
<tr>
<td>quadratic fit (0.1)</td>
<td>66</td>
<td>70</td>
</tr>
<tr>
<td>quadratic fit2</td>
<td>14</td>
<td>54</td>
</tr>
<tr>
<td>quadratic fit2 (0.1)</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>styblinski</td>
<td>118</td>
<td>52</td>
</tr>
<tr>
<td>styblinski (0.1)</td>
<td>220</td>
<td>74</td>
</tr>
<tr>
<td>sortOfStyblinski</td>
<td>53</td>
<td>33</td>
</tr>
<tr>
<td>sortOfStyblinski (0.1)</td>
<td>73</td>
<td>25</td>
</tr>
<tr>
<td>jetApprox</td>
<td>126</td>
<td>139</td>
</tr>
<tr>
<td>jetApprox (0.1)</td>
<td>161</td>
<td>236</td>
</tr>
<tr>
<td>jetApprox - good fit</td>
<td>46</td>
<td>33</td>
</tr>
<tr>
<td>jetApprox - good fit (0.1)</td>
<td>38</td>
<td>26</td>
</tr>
<tr>
<td>jetApprox - bad fit</td>
<td>108</td>
<td>219</td>
</tr>
<tr>
<td>jetApprox - bad fit (0.1)</td>
<td>105</td>
<td>157</td>
</tr>
</tbody>
</table>

Figure 14: Comparison of runtimes on discontinuity benchmarks. Running times are in seconds.
numerical errors arising from the arithmetic computations. Our tool quantifies the error and leaves it up to the user and its application to determine whether it is sufficiently small, without considering notions such as robustness or continuity.\[27\] shows programs robust by considering all possible execution paths. The paths are obtained by a symbolic execution engine and the robustness property is proven by showing that each pair of paths does not differ more than a specified amount. While their approach can produce witnesses for a violation of this property, this work only considers integer programs and relies entirely on a solver. Our approach is similar in that we also consider pairwise difference between paths, but we further present an approach how to scale the solving part to finite-precision using separation of errors and approximation. \[24\] combines abstract interpretation with model checking to check the stability of programs, tracking one input at a time. \[25\] uses concolic execution to find two sets of inputs which maximize the difference in the outputs. These approach are based on testing, however, and cannot prove sound bounds.

\[21\] presents techniques for arbitrary-precision and adaptive precision arithmetic, aimed at geometric applications with the goal to decide geometric predicates robustly. For our applications, we would like to statically prove that a certain data type precision is sufficient and take advantage of efficient hardware during the execution.

8. Conclusion

By taking the idea of linear approximation to entire code fragments (instead of applying it stepwise as in affine arithmetic), we have obtained a precise yet reasonably scalable approach to estimate errors in complex numerical code. The key to making this possible is symbolic computation, present in our analysis as well as in the non-linear constraint solving of the Z3 SMT solver. In a range of benchmarks, our implementation handled non-linear computation, conditionals and loops, for which it also generated closed-form descriptions of bounds on errors. We thus believe we have developed an interesting approach, as well as a tool for sound and automated descriptions of bounds on errors. We therefore believe we have developed benchmarks, our implementation handled non-linear computation, non-linear constraint solving of the Z3 SMT solver. In a range of

is symbolic computation, present in our analysis as well as in the

taken a precision floating-point arithmetic and fast

approximately, but further present an approach how to scale the

approximation.

approach is similar in that we also consider pairwise difference

and take advantage of efficient hardware during the execution.

References