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 accurate upper bounds on numerical errors, including roundoffs due to the use of a finite-precision representation for real numbers such as floating point or fixed-point arithmetic. Previous approaches for this task are limited in their accuracy and scalability, especially in the presence of nonlinear arithmetic. Our main idea is to decouple the computation of newly introduced roundoff errors from the amplification of existing errors. To characterize the amplification of existing errors, we use the derivatives of functions corresponding to program fragments. We implemented this technique in an analysis for programs containing nonlinear computation, conditionals, and a certain class of loops. We evaluate our system on a number of benchmarks from embedded systems and scientific computation, showing substantial improvements in accuracy and scalability over the state of the art.

1. Introduction

Numerical software, common in scientific computing and embedded systems, inevitably uses floating points or other approximations of the real arithmetic in which its algorithms are typically designed. Many problem domains come with additional sources of imprecision, such as measurement and truncation errors, increasing the uncertainty on the computed results. We need adequate tools to help developers understand whether the computed values meet the accuracy requirements and remain meaningful in the presence of the errors. This is particularly important for safety-critical systems.

Precise and sound error estimation is hard particularly in the presence of nonlinear arithmetic. Roundoff errors and error propagation depend on the ranges of variables in complex and non-obvious ways; even determining these ranges precisely for nonlinear code poses a challenge. Furthermore, due to numerical errors, the control flow in the finite-precision implementation may diverge from the ideal real-valued one, taking a different branch and producing a result that is far off the expected one. Quantifying discontinuity errors is hard due to many correlations and nonlinearity but also due to lack of smoothness or continuity of the underlying functions that arise in practice [7]. In loops, roundoff errors grow, in general, unboundedly. Even if an iteration bound is known, loop unrolling approaches scale poorly when applied to nonlinear code.

Existing state-of-the-art sound and automated error estimation techniques rely on stepwise application of affine arithmetic (AA) [11]. Fluctuat [17] uses abstract interpretation whose domain leverages affine arithmetic both for the range of variables and for the error computation. Fluctual has successfully applied this techniques to code containing linear arithmetic. On the other hand, AA inevitably introduces over-approximations when applied to *nonlinear* functions. Fluctuat deals with this by adding constraints on the noise terms of AA to improve the ranges computed by AA [16]. Our previously developed tool Rosa [10] 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, linearizing each time, and thus fails to capture the overall effect of a nonlinear function on uncertainties. As a result, we found the accuracy of the computed error bounds for both tools unsatisfactory. Both Fluctuat and Rosa also include a procedure to soundly estimate discontinuity errors, but, again, the approaches work well only for linear or simple functions, severely limiting the analysis of numerical code containing branches.

1.1 Examples

We illustrate these challenges and give a high-level overview of our solutions on several examples. The new techniques we propose build on Rosa [10] and also use its functional specification language, written in a subset of Scala. We denote the new tool presented in this paper by Rosa*.

1.1.1 Propagation of Errors in Nonlinear Codes

Figure 1 shows the code of a jet engine controller benchmark [1]. The initial errors of 1e-11 model possible noise on the sensors. This example is challenging to analyze because of the complexity of the function, and in particular because of the large number of correlations between the two input variables [10]. Fluctuat and Rosa compute an error bound of 4.67e-4 and 1.40e-4 respectively. Through simulation, we have determined an approximate *lower* bound on the error of 3.64e-8, suggesting a large over-approximation by current techniques. We propose a new error computation based on *separating* the propagation of *initial* errors from the roundoff 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, instead of using affine arithmetic as previously to propagate existing errors, we use the (partial) derivative of the underlying real-valued function to compute how the initial errors are magnified. We apply this new error propagation to the computation of the temporary variable t and the final expression, considering the entire arithmetic expression each time. This allows us to compute an approximation of the *global* effect of the function on the input errors. This is in contrast to the local linear approximations that affine arithmetic performs at each

```
def jetEngineRefactored(x: Real, y: Real): Real = {
  require(-5<=x && x<=5 && -20<=y && y<=5 &&
          x +/- 1e-11 && y +/- 1e-11)
  val t = (3*x*x + 2*y - x)
  x + ((2*x*(t/(x*x + 1))*(t/(x*x + 1) - 3) +
    x*x*(4*(t/(x*x + 1))-6))*(x*x + 1) +
  3*x*x*(t/(x*x + 1))+x*x*x+x+3*((3*x*x + 2*y -x)/(x*x + 1)))
}
```

Figure 1: Jet engine benchmark

```
def sine(x: Real): Real = {
  require(-5 <= x && x <= 5)
  x - x*x*x/6 + x*x*x*x*x/120
}

def pendulum(t:Real, w:Real, n:LoopCounter):(Real,Real)={
  require(-2 <= t && t <= 2 && -5 <= w && w <= 5 &&
    -2.01 <= ~t && ~t <= 2.01 && -5.01 <= ~w && ~w <= 5.01)
  if (n < 100) {
    val h:Real=0.01
    val L:Real=2.0
    val m:Real=1.5
    val g:Real=9.80665
    val k1t = w
    val k1w = -g/L * sine(t)
    val k2t = w + h/2*k1w
    val k2w = -g/L * sine(t + h/2*k1t)
    val tNew = t + h*k2t; val wNew = w + h*k2w
    pendulum(tNew, wNew, n + 1)
  } else { (t, w) }
}
```

Figure 2: Simulation of a pendulum

arithmetic operation. Additionally, our procedure is backed by a nonlinear SMT-solver to compute a guaranteed upper bound on the derivative for all possible inputs, yielding a fully automated sound approach, which can still capture nonlinear correlations accurately. Using this technique, Rosa* computes an upper bound on the error of 3.36e-7 which is orders of magnitude more accurate. The analysis takes a similar time as in existing tools.

Application to Loops with Constant Ranges. In general, numerical errors in loops grow unboundedly and the state-of-the-art to compute sound error bounds in complex code is by unrolling. It turns out, however, that our separation of errors allows us to express the error as a function of the number of loop iterations. We have identified a class of loops which allows us to derive a closed-form expression on the loop error bounds. This expression, on one hand, constitutes an inductive invariant, and, on the other hand, can be used to compute concrete error bounds. While this approach is limited to loops where the variable ranges are bounded, our experiments show that this approach can already analyze interesting loops that are out of reach for current tools. Figure 2 shows such an example: a Runge Kutta order 2 simulation of a pendulum. `t` and `w` are the angle the pendulum forms with the vertical and the angular velocity respectively. We approximate the sine function with its order 5 Taylor series polynomial. We focus on *roundoff* errors between the system following the real-valued dynamics and the system following the same dynamics but implemented in finite precision (we do not attempt to capture truncation errors due to the numerical integration, nor due to Taylor approximation of sine). After 100 iterations, Rosa* determines that the error on the result is at most 8.82e-14. Fluctuat uses unrolling and, for 100 iterations, computes an error bound of $[-\infty, \infty]$, while Rosa times out.

1.1.2 Discontinuities

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 1. We obtained this approximation by fitting a polynomial to a sample of values of the original function. The resulting function is not continuous. A precise constraint encoding the difference between the real-valued and finite-precision computation, if they take different paths, features variables that are tightly correlated. This makes it hard for SMT-solvers to cope with and makes linear approaches

```
def jetApproxGoodFitErr(x: Real, y: Real): Real = {
 require(-5<=x && x<=5 && -5<=y && y<=5 &&
         x +/- 0.001 && y +/- 0.001)
  if (y < x)
    -0.317581 + 0.0563331*x + 0.0966019*x*x + 0.0132828*y +
      0.0372319*x*y + 0.00204579*y*y
  else
    -0.330458 + 0.0478931*x + 0.154893*x*x + 0.0185116*y -
      0.0153842*x*y - 0.00204579*y*y
}
```

Figure 3: Piece-wise approximation of the jet engine controller

imprecise. We explore the separation of errors idea in this scenario as well, to soundly estimate errors due to conditional branches. We separate the real-valued difference from finite-precision artifacts. The individual error components are easier to handle individually, yet preserve enough accuracy. We show in our experimental results that this trade-off between accuracy and scalability can significantly outperform current techniques.

In our example, the real-valued difference between the two branches is bounded by 0.0428 (making it arguably a reasonable approximation given the large possible range of the result). However, this is not a sound estimate for the discontinuity error in the presence of roundoff and initial errors (in our example 0.001). With Rosa*, we can confirm that the discontinuity error is bounded by 0.0450, with all errors taken into account, whereas Fluctuat and Rosa compute two orders of magnitude larger errors of 5.19 and 3.77, respectively.

1.2 Summary of Contributions

The focus of this paper is a *sound* and *automated* technique for numerical error estimation in *nonlinear* finite-precision computations with control flow.

- We propose an approach for automatic error estimation based on the idea of *separation of errors* into propagation errors and roundoff errors. We show how this general idea applies to three challenging dimensions of numerical error estimation: nonlinearity, loops and discontinuities.
- We develop an approach for computing propagation errors using derivatives to characterize the global sensitivity of a function to input changes and apply this approach to non-linear computation.
- We apply the idea of separation of errors to programs with branches to develop a new way of soundly estimating the discontinuity errors arising when the real-valued ideal and the finite-precision computation diverge.
- For loops whose variable ranges are bounded, we derive a technique for computing error bounds as a function of the number of iterations.
- 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. The source code of our tool and the benchmarks are publicly available at

  `https://github.com/malyzajko/rosa`

  Our techniques remain applicable to any floating-point arithmetic whose basic arithmetic operations are rounded according to the IEEE754 standard [32] (rounding to nearest). Furthermore, we also support fixed-point arithmetic with truncation to any bitlength.

1.3 Problem Definition and Notation

We consider nonlinear computations in functions given by straight-line code, conditionals, and simple loops. An input program is given by the following grammar.

```
P ::= def mName(args): res = {
        require(A_1 ∧ ... ∧ A_n)
        ( L | D | B ) }
A ::= C | x +/- const | S
S ::= S ∧ S | S ∨ S | ¬ S | C
L ::= if (n < const) mName(B, n + 1) else args
D ::= if (C) D else D | B
B ::= val x = F; B | F
F ::= F + F | F - F | F * F | F / F | √F | X
C ::= F ≤ F | F < F | F ≥ F | F > F
X ::= x | const
```

`args` denotes possibly multiple arguments and `res` can be a tuple. The specification language is functional, so we represent loops as recursive functions (denoted `L`), where `n` denotes the loop iteration count. For loop-free code `D`, note that more complex conditions on branches can be expressed with nesting.

Let us denote by P the real-valued function representing our program and by x its input. Denote by $\tilde{P}$ the corresponding finite-precision implementation of the program, which has the same syntax tree but with operations interpreted in finite-precision arithmetic. Let $\tilde{x}$ denote the input to this finite-precision program. The goal in this paper is to estimate the difference:

$$\max_{x,\tilde{x}} \ |P(x) - \tilde{P}(\tilde{x})| \tag{1}$$

The domains of x and $\tilde{x}$, over which this expression is to be evaluated, are given by the user-provided precondition in the **`require`** clause. It defines range bounds $x_i \in [a_i, b_i], \tilde{x}_i \in [c_i, d_i]$ for each component of the possibly multivariate input, as well as absolute error bounds on the inputs of the form x_i `+/-` λ_i that define the relationship $|x - \tilde{x}| \leq \lambda$, understood component-wise. If no errors are given explicitly, we assume roundoff as the initial error. We give more details about the semantics of programs and specifications in the appendix; see also [10].

Corresponding to the syntactic program is a real-valued mathematical expression which is the input to our core error computation procedure. Concretely, the input consists of one or several real-valued functions $f : \mathbb{R}^m \to \mathbb{R}^n$ over some inputs $x_i \in \mathbb{R}$, representing the arithmetic expressions `F`. We denote by f and x the exact *ideal* real-valued function and variables and by $\tilde{f} : \mathbb{R}^m \to \mathbb{R}^n, \tilde{x}_i \in \mathbb{R}$ their *actual* finite-precision counter-parts. Note that for our analysis all variables are real-valued; the finite-precision variable $\tilde{x}$ is considered as a noisy versions of x. We perform the error computation with respect to some fixed target precision in floating-point or fixed-point arithmetic; this choice gives error bounds for each individual arithmetic operation.

When P consists of a **nonlinear arithmetic expression** alone (`F`), then Equation 1 reduces to bounding the absolute error on the result of evaluating $f(x)$ in finite precision arithmetic: $\max_{x,\tilde{x}} |f(x) - \tilde{f}(\tilde{x})|$. When the body of P is a **loop** (`L`), then the constraint reduces to computing the overall error after k-fold iteration f^k of f, where f corresponds to the loop body. We define for any function H: $H^0(x) = x$, $H^{k+1}(x) = H(H^k(x))$. We are then interested in bounding:

$$\max_{x,\tilde{x}} |f^k(x) - \tilde{f}^k(\tilde{x})|$$

. For code containing branches (grammar rule `D`), Equation 1 accounts also for the **discontinuity** error. For example, if we let f_1 and f_2 be the real-valued functions corresponding to the **`if`** and the **`else`** branch respectively with the **`if`** condition c, then, if $c(x) \wedge \neg c(\tilde{x})$, the discontinuity error is given by $|f_1(x) - \tilde{f}_2(\tilde{x})|$, i.e., it accounts for the case where the real computation takes the if-branch, and the finite-precision one takes the else branch. The overall error on P from Equation 1 in this case must account for the maximum of discontinuity errors between all pairs of paths, as well as propagation and roundoff errors for each path.

2. Propagation of Errors in Nonlinear Arithmetic

The first challenge we address is the error estimation for a loop-free nonlinear function without branches: $|f(x) - \tilde{f}(\tilde{x})|$ where $|x - \tilde{x}| \leq \lambda, f : \mathbb{R}^m \to \mathbb{R}^n$ and where the ranges for x and $\tilde{x}$ are given by the precondition.

2.1 Separation of Errors

Approaches based on interval or affine arithmetic treat all errors equally: the 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:

$$\begin{aligned} |f(x) - \tilde{f}(\tilde{x})| &= |f(x) - f(\tilde{x}) + f(\tilde{x}) - \tilde{f}(\tilde{x})| \\ &\leq |f(x) - f(\tilde{x})| + |f(\tilde{x}) - \tilde{f}(\tilde{x})| \end{aligned} \tag{2}$$

The first term, $|f(x) - f(\tilde{x})|$, captures the error on the result of f caused by the initial error between x and $\tilde{x}$. The second term, $|f(\tilde{x}) - \tilde{f}(\tilde{x})|$, covers the roundoff error committed when evaluating f in finite precision, but note that we can now compute this roundoff error on the same input $\tilde{x}$. Thus, we separate the overall error into the propagation of existing errors, and the newly committed roundoff errors. We denote by $\sigma_f : \mathbb{R}^m \to \mathbb{R}^n$ the function which returns the roundoff error committed when evaluating an expression f in finite-precision arithmetic: $\sigma_f(\tilde{x}) = |f(\tilde{x}) - \tilde{f}(\tilde{x})|$. We omit the subscript $_f$, when it is clear from the context. Further, $g : \mathbb{R}^m \to \mathbb{R}^n$ denotes a function which bounds the difference in f, given a difference in its inputs: $|f(x) - f(y)| \leq g(|x - y|)$. When $m, n > 1$, the absolute values are component-wise, e.g. $g(|x_1 - y_1|, \ldots, |x_m - y_m|)$, but when it is clear from the context, we will write $g(|x - y|)$ for clarity. Thus, the overall numerical error is given by:

$$|f(x) - \tilde{f}(\tilde{x})| \leq g(|x - \tilde{x}|) + \sigma(\tilde{x}) \tag{3}$$

One alternative to Equation 2 would be to bound the error by $|f(x) - \tilde{f}(x)| + |\tilde{f}(x) - \tilde{f}(\tilde{x})|$. The first term now corresponds to roundoff errors, but the second requires bounding the difference of $\tilde{f}$ over a certain input interval. In the separation that we have chosen, we need to compute the difference over the real-valued f. Note that f is a simpler function than its finite-precision counterpart, and its analysis is reusable across different concrete implementations.

2.2 Computing New Roundoff Errors

For computing the newly committed roundoff errors (the function σ), we use our existing procedure from Rosa [10], which is based on affine arithmetic (AA). We briefly review it here for completeness. Rosa represents roundoff errors as an affine form $\hat{x} = x_\circ + \sum_{i=1}^{k} x_i \epsilon_i, \ \epsilon_i \in [-1, 1]$, where each x_i represents the magnitude of a deviation from the central value $x_\circ$. For each arithmetic operation, Rosa adds a new linear form $x_{k+1}\epsilon_{k+1}$ with x_{k+1} the magnitude of the roundoff error committed at that operation and ϵ_{k+1} is a formal variable. Existing errors are propagated with the standard rules of affine arithmetic [10, 11]. The total error represented by an affine form is the maximum absolute value of the interval $[x_\circ - rad(\hat{x}), x_\circ + rad(\hat{x})], rad(\hat{x}) = \sum_i |x_i|$.Note that we use AA only for estimating the newly committed roundoff errors (σ). Since these errors are local, we found AA suitable for this purpose. In contrast, the propagation of existing errors (function g above) depends highly on the steepness of the function, so we want to capture as much global information, such as correlations,

as possible. This is only feasible when looking at the function as a whole, and we describe it in the sequel.

2.3 Computing Propagation Coefficients

We instantiate Equation 3 with $g(x) = K \cdot x$, i.e. $|f(x) - f(y)| \leq K|x-y|$ which bounds the deviation on the result due to a difference in the input by a linear function in the input errors. The constant K (or vector of constants K_i in the case of a multivariate function) is to be determined for each function f 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 $\tilde{f}$.

Error amplification or diminution depends on the derivative of the function at the *value of the inputs*. The steeper the function, i.e. the larger the derivative, the more the errors are magnified. For $f : \mathbb{R}^m \to \mathbb{R}$ we have

$$|f(x) - f(\tilde{x})| \leq \sum_{i=1}^{m} K_i \lambda_i, \qquad \text{where } K_i = \sup_{x,\tilde{x}} \left| \frac{\partial f}{\partial w_i} \right| \quad (4)$$

where λ_i are the initial errors and w_i denote the formal parameters of f. This computation naturally extends component-wise to multiple outputs. Thus, the propagation coefficients are computed as a sound bound on the Jacobian.

Derivation We formally derive the computation of the propagation coefficients K_i for a multivariate function $f : \mathbb{R}^m \to \mathbb{R}$ 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: $f(z) - f(y) = h(1) - h(0) = h'(\zeta)$, where $\zeta \in [0,1]$.

$$\begin{aligned}
&|f(z) - f(y)| = |h'(\zeta)| = |\nabla f(y + \zeta(z-y)) \cdot (z-y)| \\
&= \left| \left(\left.\frac{\partial f}{\partial w_1}\right|_s, \ldots, \left.\frac{\partial f}{\partial w_n}\right|_s \right) \cdot (z-y) \right|, \quad s = y + \zeta(z-y) \\
&= \left| \frac{\partial f}{\partial w_1} \cdot (z_1 - y_1) + \cdots + \frac{\partial f}{\partial w_m} \cdot (z_m - y_m) \right| \\
&\leq \sum_{i=1}^{m} \left| \frac{\partial f}{\partial w_i} \right| \cdot |z_i - y_i| \qquad (**)
\end{aligned}$$

where the partial derivatives are evaluated at $s = y + \zeta(z-y)$ (which we omit for readability). The value of s in (**) is constraint to be in $s \in [y, z]$, 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 all possible values of x and $\tilde{x}$. With $|y_i - z_i| \leq \lambda_i$ we obtain

$$|f(x) - f(\tilde{x})| \leq \sum_{i=1}^{m} K_i \lambda_i, \qquad \text{where } K_i = \sup_{x,\tilde{x}} \left| \frac{\partial f}{\partial w_i} \right|$$

Bounding Partial Derivatives We compute the partial derivatives symbolically. Recall that the arithmetic operations permitted are $\{+, -, *, /, \sqrt{}\}$, which leaves the possibility of discontinuities and undefined expressions. We detect these automatically during the bound computation, so we do not need to make or check any assumptions on the derivatives up-front.

We need to soundly bound the partial derivatives over all possible values of x and $\tilde{x}$. Both interval and affine arithmetic suffer from possibly large over-approximations due to nonlinearity and loss of correlations. Furthermore, they cannot take additional constraints into account, for example from branch conditions (e.g. `y < x`) or user defined constraints on the inputs. We use the range computation from Rosa [10] to bound the ranges of the derivatives. This procedure precomputes a range by interval arithmetic and then uses the Z3 SMT solver [12] to narrow down this initial estimate of the range. Using a nonlinear solver allows us to take into account correlations and additional constraints, making the ranges computed much tighter. Note that this computation is over $\mathbb{R}$, and is implemented with rationals to ensure soundness.

Sensitivity to Input Errors Beyond providing a way to compute the propagated initial errors, Equation 4 also makes explicit an upper bound on the sensitivity of the function to input errors. The user can use this knowledge, for example, to determine which inputs need to be determined more precisely, e.g. by more precise measurements or by using a larger number of iterations of a numerical algorithm to find them. We report the values of K back to the user.

2.4 Relationship with Affine Arithmetic

Both our presented propagation procedure and propagation using affine arithmetic perform approximations. The question arises then, when is it preferable to use one over the other? Our experience and experiments show empirically that for longer nonlinear computations, error propagation based on Lipschitz continuity gives better results, whereas for shorter and linear computations this is not the case. In this section, we present an analysis of this phenomenon based on an example.

Suppose we want to compute $x * y - x^2$. For this discussion we consider propagation only and disregard roundoff errors. We consider the case where x and y have an initial error of $\delta_x \epsilon_1$ and $\delta_y \epsilon_2$ respectively, where $\epsilon_i \in [-1, 1]$ are the formal noise symbols of AA. Without loss of generality, we assume $\delta_x, \delta_y \geq 0$. We first derive the expression for the error with affine arithmetic and take the definition of multiplication from [10]. We denote by $[x]$ the evaluation of the *real-valued* range of the variable x.

The total range of x is then the real-valued range plus the error: $[x] + \delta_x \epsilon_1$, where $\epsilon_1 \in [-1, 1]$. Multiplying out, and removing the $[x][y] - [x]^2$ term (since it is no error term), we obtain the expression for the error of $x * y - x^2$:

$$\begin{aligned}
&([y]\delta_x\epsilon_1 + [x]\delta_y\epsilon_2 + \delta_x\delta_y\epsilon_3) - (2[x]\delta_x\epsilon_1 + \delta_x\delta_x\epsilon_4) \\
&\quad = ([y] - 2[x])\delta_x\epsilon_1 + [x]\delta_y\epsilon_2 + \delta_x\delta_y\epsilon_3 + \delta_x\delta_x\epsilon_4
\end{aligned} \quad (5)$$

ϵ_3 and ϵ_4 are fresh noise symbols introduced by the nonlinear approximation. Now we compute the propagation coefficients:

$$\frac{\partial f}{\partial x} = y - 2x \qquad \frac{\partial f}{\partial y} = x$$

so that the error is given by

$$\Big|[y + \delta_y\epsilon_2 - 2(x + \delta_x\epsilon_1)]\Big|\delta_x + \Big|[x + \delta_x\epsilon_1]\Big|\delta_y \quad (6)$$

We obtain this expression by instantiating Equation (**) with the range expressions of x and y. Note that the ranges used in the evaluation of the partial derivatives include the errors. Multiplying out Equation 6 we obtain:

$$\Big|[y - 2x]\Big|\delta_x + \Big|[x]\Big|\delta_x + \delta_x\delta_y + \delta_x\delta_x + \delta_x\delta_x \quad (7)$$

With affine arithmetic we compute ranges for propagation at each computation step, i.e. in Equation 5 we compute $[x]$ and $[y]$ separately. In contrast, with our new technique, the range is computed once, taking all correlations into account between the variables x and y. It is these correlations that improve the computed error bounds. For instance, if we choose $x \in [1, 5]$ and $y \in [-1, 2]$ and we know that $x < y$, then by a step-wise computation we obtain $[y] - 2[x] = [-1, 2] - 2[1, 5] = [-11, 0]$ whereas taking the correlations into account, we can narrow down the range of x to $[1, 2]$

and obtain $[y-2x] = [-1,2] - 2[1,2] = [-5,0]$. Hence, since we compute the maximum absolute value of these ranges for the error computation, AA will use the factor 11, whereas our approach will use 5.

On the other hand, comparing Equation 7 with Equation 5, we see that one term $\delta_x\delta_x$ is included twice with our approach, whereas in the affine propagation it is only included once. We conclude that a Lipschitz-based error propagation is most useful for longer computations where it can leverage correlations. In other cases, we keep the existing affine arithmetic-based technique. It does not require a two-step computation, so we want to use it for smaller expressions. We remark that for linear operations the two approaches are equivalent.

2.5 Higher Order Taylor Approximation

In subsection 2.3 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 $a \in S$ and $a + h \in S$, then

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

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

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

for some $c \in (0,1)$. □

Using Taylor's theorem in several variables, we compute the Taylor expansion of $f(\tilde{x})$ to first order in our setting:

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

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

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 section 5 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. [19] 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.

3. Loops with Bounded Ranges

We have identified a class of loops for which the propagation of errors idea allows us to express the numerical errors as a function of the number of iterations. Concretely, we assume a single non-nested loop without conditional branches for which the ranges of variables are bounded and fixed statically. We do not attempt to prove that ranges are preserved across loop iteration; we leave the discovery of suitable inductive invariants that implies ranges for future work. Our approach does not include all loops, but it does cover a number of interesting patterns, including simulations of initial value problems in physics. We note that the alternative for analyzing numerical errors in general nonlinear loops is unrolling, which, as our experiments show, does not scale well.

Representing the computation of the loop body by f, we want to compute the overall error after k-fold iteration f^k of f: $|f^k(x) - \tilde{f}^k(\tilde{x})|$. f, g and σ 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 3.

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 monotonic. Further, σ and λ satisfy $\sigma(\tilde{x}) = |f(\tilde{x}) - \tilde{f}(\tilde{x})|$ and $|x - \tilde{x}| \leq \lambda$. The absolute value is taken component-wise. Then the numerical error after m iterations is given by

$$|f^k(x) - \tilde{f}^k(\tilde{x})| \leq g^k(|x-\tilde{x}|) + \sum_{i=0}^{k-1} g^i(\sigma(\tilde{f}^{k-i-1}(\tilde{x}))) \quad (8)$$

Thus, the overall error after k iterations can be decomposed into the initial error propagated through k iterations, and the roundoff error from the i^{th} iteration propagated through the remaining iterations.

Proof: We show this by induction. The base case $m = 1$ has already been covered in subsection 2.1. By adding and subtracting $f(\tilde{f}^{m-1}(\tilde{x}))_1$ we get

$$\begin{pmatrix} |f^m(x)_1 - \tilde{f}^m(\tilde{x})_1| \\ \vdots \\ |f^m(x)_n - \tilde{f}^m(\tilde{x})_n| \end{pmatrix}$$
$$\leq \begin{pmatrix} |f^m(x)_1 - f(\tilde{f}^{m-1}(\tilde{x}))_1| \\ \vdots \\ |f^m(x)_n - f(\tilde{f}^{m-1}(\tilde{x}))_n| \end{pmatrix} + \begin{pmatrix} |f(\tilde{f}^{m-1}(\tilde{x}))_1 - \tilde{f}^m(\tilde{x})_1| \\ \vdots \\ |f(\tilde{f}^{m-1}(\tilde{x}))_n - \tilde{f}^m(\tilde{x})_n| \end{pmatrix}$$

Applying the definitions of g and σ

$$\leq g \begin{pmatrix} |f^{m-1}(x)_1 - \tilde{f}^{m-1}(\tilde{x})_1| \\ \vdots \\ |f^{m-1}(x)_n - \tilde{f}^{m-1}(\tilde{x})_n| \end{pmatrix} + \sigma(\tilde{f}^{m-1}(\tilde{x}))$$

then using the induction hypothesis and monotonicity of g,

$$\leq g\left(g^{m-1}(\vec{\lambda}) + \sum_{i=0}^{m-2} g^i(\sigma(\tilde{f}^{m-i-1}(\tilde{x}))) \right) + \sigma(\tilde{f}^{m-1}(\tilde{x}))$$

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

$$\leq g^m(\vec{\lambda}) + \sum_{i=1}^{m-1} g^i(\sigma(\tilde{f}^{m-i-1}(\tilde{x}))) + \sigma(\tilde{f}^{m-1}(\tilde{x}))$$
$$= g^m(\vec{\lambda}) + \sum_{i=0}^{m-1} g^i(\sigma(\tilde{f}^{m-i-1}(\tilde{x}))) \quad \blacksquare$$

3.1 Closed Form Expression

We instantiate the propagation function g as before using propagation coefficients. Evaluating Equation 8 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. We observe that when the ranges are bounded (as by our assumption), then we can compute K as a matrix of propagation coefficients, and similarly obtain $\sigma(\tilde{f}^i) = \sigma$ as a vector of constants, both *valid for all iterations*. Then we obtain a closed-form for the expression of the error:

$$|f^k(x) - \tilde{f}^k(\tilde{x})| \leq K^k\lambda + \sum_{i=1}^{k-1} K^i\sigma + \sigma = K^k\lambda + \sum_{i=0}^{k-1} K^i\sigma$$

where λ is the vector of initial errors. If $(I-K)^k$ exists,

$$|f^k(x) - \tilde{f}^k(\tilde{x})| \leq K^k\lambda + ((I-K)^{-1}(I-K^k))\sigma$$

We obtain K^k with power-by-squaring and compute the inverse with the Gauss-Jordan method with rational coefficients to obtain sound results (though a closed-form is not strictly necessary for our purpose because we do know the number of iterations k).

Computing K and σ 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 σ. For loops, we require the user to specify both the real-valued ranges of variables (e.g. `a <= x && x <= b`) as well as the actual finite-precision ones (`c <= ~x && ~x <= d`, as in Example 2). We also require that the actual ranges always include the real ones ($[a,b] \subseteq [c,d]$), and we use the actual ranges ($[c,d]$) for the computation of K and σ. We believe that it is reasonable to assume that a user writing these applications to have the domain knowledge to be able to provide these specifications.

4. Errors due to Discontinuities

Recall the piece-wise jet engine approximation from Figure 3. 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*.

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. Existing approaches differ in how they handle the constraints introduced by the branch condition. Fluctuat constrains the affine forms of the real and floating- point computation in its abstract domain based on a logical product with the interval domain [18]. 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 simple functions, but are hard to handle beyond these. In this section, we show how to apply the separation of errors idea and overcome the limitations of these techniques.

Individual branch conditions are of the form `e1`$\circ$`e2`, where $\circ \in \{<, \leq, >, \geq\}$ and `e1`, `e2` are arithmetic expressions. More complex conditions can be obtained by nesting conditionals. We do not assume the function represented by the conditional to be neither smooth nor continuous. We perform our analysis pairwise for each pair of paths in the program. While this gives, in the worst-case, an exponential number of cases to consider, we found that many of these paths are infeasible due to inconsistent branch conditions; such infeasible paths are eliminated early.

4.1 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 **`if`** and the **`else`** branch respectively. Then, the discontinuity error is given by $|f_1(x) - \tilde{f}_2(\tilde{x})|$, i.e. the real computation takes branch f_1, and the finite-precision one f_2. The opposite case is analogous. We again apply the idea of separation of errors:

$$|f_1(x) - \tilde{f}_2(\tilde{x})| \leq |f_1(x) - f_1(\tilde{x})| + |f_1(\tilde{x}) - f_2(\tilde{x})| + |f_2(\tilde{x}) - \tilde{f}_2(\tilde{x})| \quad (9)$$

The individual components are

1. $|f_1(x) - f_1(\tilde{x})|$: the difference in f_1 due to initial errors. We can compute this difference with our propagation coefficients: $|f_1(x) - f_1(\tilde{x})| \leq K|x - \tilde{x}|$.
2. $|f_1(\tilde{x}) - f_2(\tilde{x})|$: the real-valued difference between f_1 and f_2. We can bound this value by the Z3-aided range computation from [10].
3. $|f_2(\tilde{x}) - \tilde{f}_2(\tilde{x})|$: the roundoff error when evaluating f_2 in finite-precision arithmetic. We use the procedure from [10] as before.

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 procedure. On the other hand, we clearly introduce an additional over-approximation, but we observe in our experiments that this is in general small, even for benchmarks where the precise approach of Rosa performs well. For more complex benchmarks our tool outperforms the more precise approach.

A split of the total error into two parts is also possible, e.g. as $|f_1(x) - \tilde{f}_2(\tilde{x})| \leq |f_1(x) - f_2(\tilde{x})| + |f_2(\tilde{x}) - \tilde{f}_2(\tilde{x})|$, which performs one computation less. This split, combined with a precise constraint relating x to $\tilde{x}$ is essentially what Rosa does. As mentioned before, such a precise and complex relation overwhelms the SMT solver quickly. Bounding the ranges without the correlation information yields unsatisfactory results.

4.2 Determining Ranges for x and $\tilde{x}$

As in the previous sections, it is crucial to determine the ranges of $x, \tilde{x} \in \mathbb{R}$ over which to evaluate the individual parts of Equation 9. A sound approach would be to use the same bounds as for the straight-line case, but this would lead to unnecessary over-approximations. In general, not all inputs can exhibit a divergence between the real-valued and the finite-precision computation. They are determined by the branch conditions and the errors on the variables. Consider the branch condition `if (e1 < e2)` and the case where the real-valued path takes the if-branch, i.e. variable x satisfies $e1 < e1$ and $\tilde{x}$ satisfies $e1 \geq e2$. The constraint for the finite-precision variables $\tilde{x}$ is then $e1 + \delta_1 < e2 + \delta_2 \wedge e1 \geq e2$, where δ_1, δ_2 are error intervals on evaluating $e1$ and $e2$ respectively. This constraint expresses that we want those values which satisfy the condition $e1 \geq e2$, but are "close enough" to the boundary such that their corresponding ideal real value could take the other path. 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 9. The procedure for other branch conditions is analogous.

5. Experiments

We have chosen a number of benchmarks from the domains of scientific computing and embedded systems to evaluate our techniques. We show some representative examples in the appendix. [1] We perform our test with double precision, as this is a common choice for numerical programs. Note however, that Rosa* supports both floating-point arithmetic with different precisions, as well as fixed-point arithmetic with different bit lengths. In our experience, while the absolute errors naturally change with varying precisions and data types, relative differences when comparing different tools on the same precision data type remain similar.

We compare our results against those obtained by Fluctuat and Rosa. These are the only available tools that we are aware of that can compute sound numerical error bounds automatically and for general arithmetic expressions. We denote our current extension of Rosa as Rosa*. Experiments were performed on a desktop computer running Ubuntu 14.04.1 with a 3.5GHz i7 processor and 16GB of RAM, and using the unstable branch (as of 10 December 2014) of Z3. Figure 4 summarizes the worst-case absolute errors computed by the three tools as well as the running time of the analyses.

Straight-line Computation The first part of Figure 4 evaluates our new error propagation technique for straight-line nonlinear code on a number of benchmarks from [10]. 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 vs SMT solver). The initial errors in the top benchmarks are roundoff errors only, in the bottom section we add an initial absolute error of 1e-11 to all inputs. We observe that our new technique computes tighter error bounds in most cases, but especially for benchmarks with larger initial errors, confirming that our separation of errors is useful.

Furthermore, we investigate the effect of refactoring expressions and applying our error propagation technique to each subexpression. For example, in the case of the doppler benchmark, we consider two formulations.

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

which is often the formulation produced by code generation tools, and

```
val tmp = 331.4 + 0.6 * T; (-tmp * v) / ((tmp + u)*(tmp + u))
```

In the second case, we apply the error propagation twice, once for computing the error on `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. Rosa* currently performs the compositional error propagation only for expressions defined as **`val`**s or final expressions, as too fine-grained steps would increase the running time unnecessarily or degrade the computed results. The approach can also be applied to function calls.

First-order Method Summaries

Section 2.5 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 [10] 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.

[1] For tool source code and benchmarks, please see `https://github.com/malyzajko/rosa`.

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.

```
def sine(x: Real): Real = {
  require(-3.5 < x && x < 3.5 && x +/- 1e-8)

  x - (x*x*x)/6.0 + (x*x*x*x*x)/120.0 -
     (x*x*x*x*x*x*x)/5040.0
}ensuring(res⇒ -1.0 < ~res && ~res < 1.0 && 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`

```
  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 2.5 will instead use the computed summaries and determine the error for the first case to be 1.000e-8 and for the second case 4.945e-15, improving the error bounds by more than 6 decimal orders of magnitude. This illustrates the benefits of relational summaries that our approach computes.

Loops The second part of Figure 4 shows our results on benchmarks with loops. The mean benchmark computes the running average of values in a range of $[-1200, 1200]$ and the nbody benchmark is a 2-body simulation of Jupiter orbiting around the Sun. Both benchmarks are given in the appendix. While for the mean benchmark Fluctuat computes tighter error bounds, our approach scales much better for large numbers of iterations. This limit in scalability of Fluctuat (and Rosa), due to unrolling, is also apparent for the pendulum and nbody benchmarks, where it returns the trivial error bound $[-\infty, \infty]$.

Discontinuities The bottom part of Figure 4 compares absolute errors computed for our discontinuity benchmarks by the three tools. We have made an effort to choose our benchmarks such that they cover a variety of characteristics. The first three benchmarks are unary functions taken from [10, 18], while the remaining ones are binary. For the binary benchmarks, we distinguish those whose branch condition is relational (`x < y`, marked with R) as opposed to range-based (`x < 0`), and benchmarks where the arguments have an initial error of 0.001. We have derived these benchmarks by piece-wise approximating a more complex function, a common pattern seen in embedded systems.

We observe that Rosa can leverage its more precise constraint formulation and can compute tighter bounds for some of the examples. On the more complex examples, however, our new technique in Rosa* outperforms both Fluctuat and Rosa significantly. For Rosa we know that this difference is due to the complexity of the constraint constructed, on which Z3 times out so that it falls back to interval arithmetic. For Fluctuat we suspect that the limitation is due to linearity of the underlying domain. In particular, our new technique is better capable to distinguish the relative difference between benchmarks. For example, we can clearly see the effect of added initial errors on the styblinski or jetApproxGoodFitErr benchmark, whereas this difference is barely noticeable for Fluctuat and Rosa. The 'good fit' and 'bad fit' versions of the jet engine benchmark are two approximations where the two branches are more or less close at the boundary. Rosa*'s order of magnitude difference strongly hints at this fact, while Fluctuat's and Rosa's results are

	benchmark	Accuracy Fluctuat	Accuracy Rosa	Accuracy Rosa*	Runtime Fluctuat	Runtime Rosa	Runtime Rosa*
straight-line nonlinear	doppler	**3.90e-13**	4.36e-13	4.29e-13	< 1	2	20
	dopplerRefactored	3.90e-13	4.19e-13	**2.68e-13**	< 1	2	17
	jetengine	4.08e-8	1.16e-8	**5.33e-9**	< 1	40	287
	jetengineRefactored	4.08e-8	1.16e-8	**4.91e-9**	< 1	39	255
	rigidBody	3.65e-11	3.65e-11	3.65e-11	< 1	6	7
	rigidBodyRefactored	3.65e-11	3.65e-11	3.65e-11	< 1	6	7
	sine	7.97e-16	6.40e-16	**5.18e-16**	< 1	3	4
	sineOrder3	1.15e-15	1.23e-15	**9.96e-16**	< 1	< 1	1
	sqroot	3.21e-13	3.09e-13	**2.87e-13**	< 1	1	2
	turbine1	9.20e-14	8.87e-14	**5.99e-14**	< 1	1	18
	turbine1Refactored	9.26e-14	8.87e-14	**5.15e-14**	< 1	1	3
	with added initial errors						
	dopplerRefactored	5.45e-11	5.29e-11	**2.08e-11**	< 1	3	24
	jetengineRefactored	4.67e-4	1.40e-4	**3.36e-7**	< 1	40	251
	turbine1Refactored	1.82e-9	1.88e-9	**4.60e-10**	< 1	1	2
	turbine2Refactored	2.82e-9	2.90e-9	**5.86e-10**	< 1	1	2
	turbine3Refactored	1.24e-9	1.27e-9	**3.32e-10**	< 1	1	5
loops	mean (100 iter.)	**1.52e-11**	-	9.74e-10	0.5	-	8
	mean (1000 iter.)	**1.54e-10**	-	1.32e-7	35	-	9
	mean (4000 iter.)	**6.17e-10**	-	2.06e-7	814	-	8
	pendulum (50 iter)	2.43e-13	-	**2.21e-14**	49	-	8
	pendulum (100 iter)	∞	-	**8.82e-14**	-	-	8
	pendulum (1000 iter)	∞	-	**3.89e-05**	-	-	8
	nbody	∞	-	1.35e-08	-	-	781
discontinuity	squareRoot	0.0394	**0.0236**	0.0238	< 1	3	25
	squareRoot3 invalid	0.429	1.32e-9	**1.31e-9**	< 1	3	7
	linear fit	1.72	**0.637**	**0.637**	< 1	3	4
	quadratic fit	10.6	3.22	**0.255**	< 1	45	57
	quadratic fit (0.001)	11.0	3.22	**0.255**	< 1	65	71
	quadratic fit2 (R)	0.632	**9.19e-16**	1.26e-15	< 1	14	16
	quadratic fit2 (R, 0.001)	0.719	**5.55e-4**	8.52e-4	< 1	17	21
	styblinski	121.0	36.4	**2.31e-14**	< 1	179	60
	styblinski (0.001)	124.0	36.4	**0.0132**	< 1	225	73
	styblinski2 (R)	27.1	20.2	**1.09**	< 1	29	34
	styblinski2 (R, 0.001)	28.8	20.2	**1.10**	< 1	29	28
	jetApprox	18.4	6.83	**0.0232**	< 1	97	175
	jetApprox - good fit (R)	5.19	3.77	**0.0428**	< 1	27	33
	jetApprox - good fit (R, 0.001)	5.19	3.77	**0.045**	< 1	25	26
	jetApprox - bad fit (R)	9.30	4.26	**0.882**	< 1	85	206

Figure 4: Comparison of worst-case absolute errors computed by Fluctuat, our previous work Rosa and our new method implemented in Rosa*. Approximate runtimes are given in seconds. All number are rounded. We mark the best result in bold.

somewhat less clear-cut and may, for example, be caused due to inherent over-approximations.

Running Times Figure 4 also compares the running times of the different techniques. It is apparent that more accuracy in the computed errors comes at the expense of longer analysis time. Most of the difference in running times between the tools is due to the use of the nonlinear SMT solver, which directly accounts for the better accuracy. Since our technique is static and thus need not be run often, we believe that this trade-off between efficiency and accuracy is reasonable.

6. Related Work

To the best of our knowledge, Fluctuat [17, 18] is the most related to our work. We are not aware of other tools or techniques that can soundly and automatically quantify numerical errors in the presence of nonlinearity, branches and loops.

In the context of abstract interpretation, domains exist 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 [4, 8, 13, 28]. [14] presents an abstract domain which associates the ranges with the iteration count, similar to our proposed technique for loops. [27] considers the stability of loops, by proving whether loops can asymptotically diverge. The problem that we are solving is different, however, as we want quantify the *difference* between the real-valued and the finite-precision computation.

Floating-points have been formalized in the SMT-LIB format [30], and approaches exist which deal with the prohibiting complexity of bit-precise techniques via approximations [6, 21]. Efficient combination of theories needed to express roundoff errors is non-trivial, and we are not aware of an approach that is able to quantify the deviation of finite-precision computations with respect to reals. Floating-point precision assertions can also be proven using an interactive theorem prover [2, 5, 22, 25]. 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. Very tight error bounds have been shown by manual proof for certain special computations, such as powers [20]. Our work is on the other side of the trade-off between accuracy and automation as well as generality.

Several approaches also exist to test the stability of numerical programs, e.g. by perturbation of low-order bits and rewriting [33], or by perturbing the rounding modes [31]. Another common theme is to run a higher-precision program alongside the original one. [3] does so by instrumentation, [29] generates constraints which are then discharged with a floating-point arithmetic solver and [9] developed a guided search to find inputs which maximize errors. [24] uses instrumentation to detect cancellation and thus loss of precision. [23] combines abstract interpretation with model checking to check the stability of programs, tracking one input at a time. [26] 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.

It is natural to use the Jacobian for sensitivity analysis. Related to our work is a proof framework using this idea for showing programs robust in the sense of k-Lipschitz continuity [7]. Note, however, that our approach does not require programs to be continuous. [15] relaxes the strict definition of robustness to programs with specified uncertainties and presents a framework for proving while-loops with a particular structure robust. Our work follows the philosophy of these approaches in leveraging Jacobians of program paths, yet we explicitly incorporate the handling of roundoff errors in a fully automated system.

7. Conclusion

Using the idea of separation of errors proved to be an effective method. Whereas the questions can in principle be formulated as a non-linear constraint solving even without error separation, this technique proves to overcome the scalability and precision limitations of past approaches. 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. In a range of benchmarks, our implementation handled nonlinear computation, conditionals and certain types of loops. We thus believe we have developed an interesting approach, as well as a tool for sound and automated computation of worst-cases bounds on roundoff errors that obtains more precise results within reasonable time bounds than the existing approaches.

References

[1] Adolfo Anta and P. Tabuada. To Sample or not to Sample: Self-Triggered Control for Nonlinear Systems. *Automatic Control, IEEE Transactions on*, 55(9), 2010.

[2] Ali Ayad and Claude Marché. Multi-prover verification of floating-point programs. In *IJCAR*, 2010.

[3] Florian Benz, Andreas Hildebrandt, and Sebastian Hack. A dynamic program analysis to find floating-point accuracy problems. In *PLDI*, 2012.

[4] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *PLDI*, pages 196–207, 2003.

[5] Sylvie Boldo and Claude Marché. Formal verification of numerical programs: from C annotated programs to mechanical proofs. *Mathematics in Computer Science*, 2011.

[6] A. Brillout, D. Kroening, and T. Wahl. Mixed abstractions for floating-point arithmetic. In *FMCAD*, pages 69–76, 2009.

[7] Swarat Chaudhuri, Sumit Gulwani, Roberto Lublinerman, and Sara Navidpour. Proving Programs Robust. In *ESEC/FSE*, 2011.

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

[9] Wei-Fan Chiang, Ganesh Gopalakrishnan, Zvonimir Rakamaric, and Alexey Solovyev. Efficient Search for Inputs Causing High Floating-point Errors. In *PPoPP*, 2014.

[10] Eva Darulova and Viktor Kuncak. Sound Compilation of Reals. In *POPL*, 2014.

[11] L. H. de Figueiredo and J. Stolfi. *Self-Validated Numerical Methods and Applications*. IMPA/CNPq, Brazil, 1997.

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

[13] Jérôme Feret. Static Analysis of Digital Filters. In *ESOP*, 2004.

[14] Jérôme Feret. The Arithmetic-Geometric Progression Abstract Domain. In *VMCAI*, 2005.

[15] Ivan Gazeau, Dale Miller, and Catuscia Palamidessi. A non-local method for robustness analysis of floating point programs. In *QAPL*, 2012.

[16] Khalil Ghorbal, Eric Goubault, and Sylvie Putot. A Logical Product Approach to Zonotope Intersection. In *CAV*, 2010.

[17] Eric Goubault and Sylvie Putot. Static Analysis of Finite Precision Computations. In *VMCAI*, 2011.

[18] Eric Goubault and Sylvie Putot. Robustness Analysis of Finite Precision Implementations. In *APLAS*, 2013.

[19] Eric Goubault, Sylvie Putot, and Franck Védrine. Modular Static Analysis with Zonotopes. In *SAS*, 2012.

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

[21] L. Haller, A. Griggio, M. Brain, and D. Kroening. Deciding floating-point logic with systematic abstraction. In *FMCAD*, 2012.

[22] John Harrison. Floating-Point Verification using Theorem Proving. In *Formal Methods for Hardware Verification*, 2006.

[23] F. Ivancic, M.K. Ganai, S. Sankaranarayanan, and A. Gupta. Numerical stability analysis of floating-point computations using software model checking. In *MEMOCODE*, 2010.

[24] Michael O. Lam, Jeffrey K. Hollingsworth, and G.W. Stewart. Dynamic floating-point cancellation detection. *Parallel Computing*, 39(3), 2013.

[25] Michael D. Linderman, Matthew Ho, David L. Dill, Teresa H. Meng, and Garry P. Nolan. Towards program optimization through automated analysis of numerical precision. In *CGO*, 2010.

[26] R. Majumdar, I. Saha, and Zilong Wang. Systematic Testing for Control Applications. In *MEMOCODE*, 2010.

[27] Matthieu Martel. Static Analysis of the Numerical Stability of Loops. In *SAS*, 2002.

[28] Antoine Miné. Relational Abstract Domains for the Detection of Floating-Point Run-Time Errors. In *ESOP*, 2004.

[29] Gabriele Paganelli and Wolfgang Ahrendt. Verifying (In-)Stability in Floating-point Programs by Increasing Precision, using SMT Solving. In *SYNASC*, 2013.

[30] Philipp Rümmer and Thomas Wahl. An SMT-LIB Theory of Binary Floating-Point Arithmetic. In *Informal proceedings of 8th International Workshop on Satisfiability Modulo Theories (SMT) at FLoC*, 2010.

[31] N.S. Scott, F. Jézéquel, C. Denis, and J.-M. Chesneaux. Numerical 'health check' for scientific codes: the CADNA approach. *Computer Physics Communications*, 2007.

[32] IEEE Computer Society. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, 2008.

[33] Enyi Tang, Earl Barr, Xuandong Li, and Zhendong Su. Perturbing numerical calculations for statistical analysis of floating-point program (in)stability. In *ISSTA*, 2010.