Sound Probabilistic Numerical Error Analysis

Debasmita Lohar[1][0000−0001−8639−4116]⋆, Milos Prokop[2], and Eva Darulova[1]

[1] MPI-SWS, Saarland Informatics Campus, dlohar@mpi-sws.org, eva@mpi-sws.org
[2] University of Edinburgh, m.prokop@sms.ed.ac.uk

Abstract. Numerical software uses floating-point arithmetic to implement real-valued algorithms which inevitably introduces roundoff errors. Additionally, in an effort to reduce energy consumption, approximate hardware introduces further errors. As errors are propagated through a computation, the result of the approximated floating-point program can be vastly different from the real-valued ideal one. Previous work on soundly bounding (roundoff) errors has focused on worst-case absolute error analysis. However, not all inputs and not all errors are equally likely such that these methods can lead to overly pessimistic error bounds.
In this paper, we present a sound probabilistic static analysis which takes into account the probability distributions of inputs and propagates roundoff and approximation errors probabilistically through the program. We observe that the computed probability distributions of errors are hard to interpret, and propose an alternative metric and computation of refined error bounds which are valid with some probability.

Keywords: Probabilistic Analysis, Floating-point, Approximate Computing

1 Introduction

Many programs compute only approximate results; sometimes because exact solutions do not exist, or because resource constraints mandate using a cheaper and inexact algorithm or hardware. Approximations can thus increase the efficiency of a program, but they also introduce errors. Many applications are designed to be robust to some noise and tolerate errors—as long as they remain within some acceptable bounds. However, automatically determining how individual approximation errors influence overall program accuracy remains a challenge.

Finite precision, which efficiently approximates real-valued arithmetic, is widely used across a variety of domains from embedded systems to machine learning applications. Verification of the roundoff errors that finite-precision introduces has thus attracted significant interest in the recent past [8,15,31,19,11]. These tools compute *worst-case* absolute error bounds fully automatically. Such bounds, however, may often be pessimistic. They may not be achieved in practice and furthermore many applications are tolerant to somewhat larger, but

⋆ The author is supported by DFG grant DA 1898/2-1.

infrequent, errors (e.g. control systems, where one feedback iteration compensates for a larger error of a previous one). In order to capture such differentiated behaviour, we need to compute the *probability* of certain error bounds.

Such an analysis is even more relevant with advances in approximate computing [33], which introduces approximate architectures and storage. These hardware components are more resource efficient but often have probabilistic error behaviours: operations return a large error with a certain probability.

While techniques [5,28,21,25,6,20,26] exist which track or compute probability distributions, they do not consider and support reasoning about *errors* due to finite-precision arithmetic or approximate computing, or they only compute coarse-grained error estimates, instead of error probability distributions [22].

In this paper, we present a sound probabilistic error analysis for numerical kernels which computes uncertain distributions of errors, which soundly overapproximate all possible distributions. Our analysis extends the abstract domain of probabilistic affine forms [4] for error analysis, and combines it with a novel probabilistic version of interval subdivision to obtain both an efficient analysis, as well as useful accuracy.

We instantiate our analysis to track two kinds of errors. First, we consider standard floating-point arithmetic, where inputs are distributed according to a user-given distribution. Since roundoff errors depend on the values' magnitudes, we effectively obtain a probability distribution of errors, even though the error of each individual operation is still specified as worst-case. Secondly, inspired by approximate hardware specifications, in addition to probabilistic input ranges, we consider errors which themselves have probabilistic specifications. That is, with a certain probability the error for each operation can be larger than usual [22,27].

We observe that the computed (discretized) probability distributions of errors are hard to interpret and use by a programmer. We thus propose a new metric which states that with probability p, the error is at most C_p, where C_p should be smaller than the overall worst-case error. We show how to compute C_p from the probability distributions of errors, given a particular p by the user.

We have implemented this analysis in the prototype tool called PrAn and evaluated it on several benchmarks from literature. The results show that the proposed probabilistic error analysis refines worst-case errors and soundly computes a smaller error which holds with a predefined probability. For the standard floating-point error specification, PrAn computes refined error bounds which are on average 17% and 16.2% and up to 49.8% and 45.1% lower than worst-case errors for gaussian and uniform input distributions, respectively. These error bounds hold with at least probability 0.85. For an approximate error specification which assumes individual operations to commit an error of larger magnitude with probability 0.1, the refined errors are, again with probability 0.85, on average even 30.6% and up to 73.1% smaller than worst-case.

Contributions In summary, in this paper we present:

- the first static analysis for probability distributions of numerical errors,
- an instantiation of this analysis for worst-case floating-point and probabilistic approximate error specifications,

- a procedure to extract useful and human-readable refined error specifications,
- an experimental evaluation demonstrating the effectiveness of our analysis,
- an implementation, which we will release as open source.

2 Background

Floating-point Arithmetic is a widely used representation of real numbers on digital computers. Due to optimized hardware or software support, it is efficient, but due to its finite nature, every operation introduces roundoff errors. These errors are individually small, but can potentially accumulate over the course of a computation. To ensure the correctness of systems using floating-point arithmetic, we have to bound roundoff errors at the program's output.

An exact formalization of floating-point arithmetic is too complex for reasoning about larger programs, such that most automated tools for bounding roundoff errors use the following abstraction which is based on the specification of the IEEE 754 standard [1]:

$$x \circ_{fl} y = (x \circ y)(1 + e) + d \ , \ \ |e| \leq \epsilon_m, |d| \leq \delta_m \tag{1}$$

where $\circ \in \{+, -, *, /\}$ and $\circ_{fl}$ denotes the respective floating-point version. Square root follows similarly and unary minus does not introduce roundoff errors. The machine epsilon ϵ_m bounds the maximum relative error for normal values and for subnormal values the round-off error is expressed as δ_m. The values of ϵ_m and δ_m for single and double precision are 2^{-24}, 2^{-150} and 2^{-53} and 2^{-1075} respectively. Equation 1 assumes rounding-to-nearest rounding mode, which is the usual default, and no overflow (the analyses prove that this cannot occur).

Worst-case Error Analysis Existing work on bounding roundoff errors computes worst-case absolute errors:

$$\max_{x \in [a,b]} \mid f(x) - \tilde{f}(\tilde{x}) \mid$$

where f and x denote the real-valued function and input and $\tilde{f}$ and $\tilde{x}$ their floating-point counterparts. Floating-point roundoffs depend on the magnitude of inputs (see Equation 1), such that tools compute worst-case errors for some bounded, user-provided input domain. There are two different approaches to bounding roundoff errors: dataflow analysis [15,9,8,7,11], and global optimization [31,19,24].

Dataflow analysis, which is relevant to our approach, tracks real-valued ranges and finite-precision errors at each abstract syntax tree (AST) node using the domains of interval arithmetic (IA) [23] or affine arithmetic (AA) [14]. Interval arithmetic computes a bounding interval for each basic operation as $x \circ y = [\min(x \circ y), \max(x \circ y)]$ where $\circ \in \{+, -, *, /\}$ and analogously for square root. It is widely used and efficient to bound ranges of variables, but results in overapproximations, because it does not track correlations between variables.

Affine arithmetic represents a range of possible values by an affine form: $\hat{x} = x_0 + \sum_{i=1}^{n} x_i \eta_i$ where $\eta_i \in [-1, 1]$, x_0 is the mid-point of the range and each *noise term* $x_i \eta_i$ represents a deviation from this mid-point. The interval of values represented by an affine form is given by $[\hat{x}] = [x_0 - rad(\hat{x}), x_0 + rad(\hat{x})]$, $rad(\hat{x}) = \sum_{i=1}^{n} |x_i|$. The symbolic variables η_i track linear correlations between variables, allowing AA to often compute tighter ranges than IA. In AA, linear arithmetic operations are computed term-wise, while nonlinear operations are approximated and thus result in a certain imprecision of the analysis.

IA and AA can be combined with interval subdivision, which subdivides input domains into subintervals of usually equal width and runs the analysis separately on each. The overall error is computed as the maximum error over all subintervals. Interval subdivision is, for instance, implemented in the tools Fluctuat [15] and Daisy [8]. It provides tighter error bounds as smaller input domains usually result in smaller over-approximations.

Probabilistic Affine Arithmetic The worst-case error can be pessimistic as it takes into account only the ranges of the input variables. However, inputs may be distributed according to different distributions. To get a more nuanced view, we need to track probability distributions through the program. One possible approach to track real-valued probability distributions is probabilistic affine arithmetic [4]. Here, we provide a high-level idea of probabilistic AA needed to understand our probabilistic error analysis. For details, we refer the reader to [4].

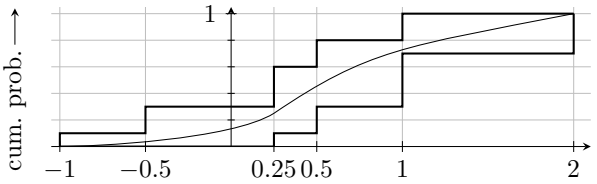
In standard AA, the symbolic noise terms η_i nondeterministically take a value in the interval $[-1, 1]$. Probabilistic AA extends each noise term η_i to carry a *probability distribution* d_{η_i} with support $[-1, 1]$. Thus, a probabilistic affine form represents a probability distribution which is computed by summing the weighted distributions from each noise term:

$$x_0 + \sum_{i=1}^{n} x_i d_{\eta_i} \tag{2}$$

For representing the probability distributions, probabilistic AA uses Interval Dempster-Shafer structures [30,13] (DSI). DSIs discretize a distribution and represent it as a finite list of focal elements: $d = \{\langle \mathbf{x_1}, w_1 \rangle, \langle \mathbf{x_2}, w_2 \rangle, \cdots, \langle \mathbf{x_n}, w_n \rangle\}$ where $\mathbf{x_i}$ is a closed non-empty interval and $w_i \in]0, 1]$ is the associated probability and $\sum_{k=1}^{n} w_k = 1$. That is, the value of a variable x is (nondeterministically) within the interval $\mathbf{x_1}$ with probability w_1, in interval $\mathbf{x_2}$ with probability w_2, and so on. For example, the DSI

$$d=\{\langle[-1,0.25],0.1\rangle,\langle[-0.5,0.5],0.2\rangle,\langle[0.25,1],0.3\rangle,\langle[0.5,1],0.1\rangle,\langle[0.5,2],0.1\rangle,\langle[1,2],0.2\rangle\}$$

represents the distribution where the probability of selecting a value in the interval [-1, 0.25] is 0.1, in the interval [-0.5, 0.5] is 0.2 and so on. Graphically, this DSI looks as follows:



DSIs represent *uncertain* distributions, that is, the set of focal elements in general represents not a single exact distribution, but a set of distributions. Thus DSIs allow us to efficiently and soundly represent the continuous distribution such that the true distribution(s) are guaranteed to be inside the computed DSI. Naturally, the abstraction comes at a cost of certain over-approximations.

To propagate probabilistic affine forms through the program, we note that the affine portion remains the same as standard AA. However, the noise symbols now carry a distribution in the form of a DSI so that we need to define arithmetic operations over DSIs.

The affine form tracks (unknown) dependencies between variables and thus also between DSIs. When two DSIs are independent of each other, arithmetic operations can be computed using standard interval arithmetic. For example, given two independent DSIs, $d_X = \{\langle \mathbf{x_i}, w_i \rangle \mid i \in [1,n]\}$ and $d_Y = \{\langle \mathbf{y_j}, v_j \rangle \mid j \in [1,m]\}$, the resultant DSI structure is: $d_Z = \{\langle \mathbf{z_{i,j}}, r_{i,j} \rangle \mid i \in [1,n], j \in [1,m]\}$ with $\mathbf{z_{i,j}} = \mathbf{x_i} \odot \mathbf{y_j}$ where $\odot \in \{+,-,*,/\}$ and $r_{i,j} = w_i \times v_j$.

If d_X and d_Y are dependent with some unknown dependency, probabilistic AA computes the intervals ($\mathbf{z_{i,j}}$) of d_Z as for the independent case. To compute the corresponding weights, we need to take into account all possible dependencies to obtain a sound over-approximation of the probability distribution. Formally, this means that we need to compute an upper and a lower bound on the cumulative distribution function at every point in the domain. Practically, since our DSIs are discretized, we need to do this computation at every 'step', i.e. for each lower and upper bound in d_Z. For each such 'step', we need to solve an optimization problem, which encodes constraints due to the unknown dependency, effectively encoding all possible dependencies. The resulting linear programming problems can be solved using a Simplex solver.

Probabilistic AA [4] thus propagates discretized uncertain probability distributions through arithmetic computations and provides a sound enclosure on the real-valued probability distribution. So far, probabilistic AA has not been extended to support error propagation.

3 Tracking Probabilistic Errors

In this work we propose to track roundoff and other approximation errors probabilistically, in order to provide less pessimistic error bounds than only a worst-case analysis. Our probabilistic analysis takes into account the distributions of input variables, tracks errors probabilistically throughout the program for a specified uniform floating-point precision, and computes a *sound* over-approximation

Algorithm 1 Tracking Probabilistic Round-off Errors

```
1: procedure probabilisticError(fnc, E, inDist, L, prec, p)
2:     subDoms = subdivide(E, inDist, L)
3:     errDist = φ
4:     for (dom_i, ρ_i) ← subDoms do
5:         absErr_i = evalProbRoundoff(fnc, dom_i, prec)        ▷ see Algorithm 2
6:         absErr_i = normalizeProb(absErr_i, ρ_i)
7:         errDist = merge(errDist, absErr_i)
8:     (err, prob) = extractErrorMetric(errDist, p)
9:     return (err, prob)                ▷ returns the refined error with probability
```

of the error distribution at the output. We observe that the resulting distributions can be hard to interpret and use. Hence, we further propose and show how to extract an *error metric* from the output distribution. The user of our approach provides a *threshold probability* p, and our technique extracts a tighter refined error bound C_p, which holds with probability p.

We focus here on straight-line arithmetic expressions. Previous techniques for worst-case error bounds reduce reasoning about errors in loops via loop unrolling [7] and loop invariants [9,15,24], and in conditionals by a path-by-path analysis [24,9] to straight-line code. These techniques can also be combined with our probabilistic analysis. We furthermore compute absolute errors. While approaches exist to compute relative errors directly [16], they only compute a result if the output range does not include zero, which however happens often in practice. Whenever the final range does not include zero, our method can compute relative errors from the absolute ones.

Running Example We will use the following program, which computes a third-order approximation of sine, as a running example for this section:

```
x := gaussian(-2.0, 2.0)
0.954929658551372 * x -  0.12900613773279798 * (x * x * x)
```

The user specifies the distribution of input variables: here 'x' is normally distributed in [-2, 2]. Suppose the user has further set the threshold probability to $p = 0.85$. The worst-case error for this program in single precision is 4.62e-7. PrAn computes the output distribution and extracts the following error metric: a smaller error of $C_p = 2.67\text{e-}7$ occurs with at least probability 0.85. □

Algorithm 1 shows the high-level overview of our analysis. It takes as input the following parameters: a program given as an arithmetic expression (fnc), ranges of the input variables (E), the probability distribution of the variables ($inDist$), a limit on the number of total subdivisions (L), a uniform floating-point precision ($prec$), and a threshold probability (p). Our analysis currently considers the same probability distribution for all variables as well as uniform floating-point precision, but it is straight-forward to extend to consider different distributions and mixed precision.

Algorithm 1 first discretizes each input range distribution into subdomains (line 2). For each subdomain it runs the probabilistic error analysis for the specified floating-point precision *prec* (line 5). The computed error distribution is then normalized by multiplying the probability of the subdomain occurring (line 6) and merge it with *errDist* (line 7) that accumulates error distributions for each subdomain to generate the complete error distribution of the output error. Finally from *errDist* we extract the error metric (line 8) and its actual probability, which may be larger than the requested p and return it (line 9).

In Section 3.1 we first describe a simplified version of Algorithm 1, which performs probabilistic interval subdivision but computes errors for each subdomain with standard worst-case roundoff analysis. While interval subdivision is standardly used to decrease overapproximations, to the best of our knowledge, it has not been used previously to compute probabilistic error bounds. Then, we extend this analysis with our novel probabilistic error analysis (Section 3.2). Finally we show how our probabilistic error analysis can be further extended to take into account approximate error specifications common in today's approximate hardware (Section 3.3).

3.1 Probabilistic Interval Subdivision

First, we consider a relatively simple extension of worst-case error analysis which takes into account the distributions of input variables: for now, we let the function `evalProbRoundoff` compute a worst-case error, instead of a probability distribution, and focus on the interval subdivision where the intervals are subdivided probabilistically (line 2 in Algorithm 1).

Our algorithm first subdivides each input interval equally and for multivariate functions takes their Cartesian product to generate subdomains. The probability of each subdomain is computed by taking the product of the probabilities of the corresponding subintervals. In this way the algorithm generates a set of tuples (dom_i, ρ_i) where a value is in subdomain dom_i with probability ρ_i. On each of these subdomains, we can run the standard worst-case dataflow error analysis from Section 2 to compute an error bound $absErr_i$ (here, we use AA for the errors, but IA for the ranges, because it tends to have less over-approximations for nonlinear expressions than standard AA). Hence, the algorithm computes for each subdomain i an error tuple $(absErr_i, \rho_i)$, i.e. with probability ρ_i, the worst-case error is $absErr_i$. From these error tuples (*errDist*), PrAn extracts the error metric.

Extracting the Error Metric We want to extract an error C_p (smaller than the worst-case error), which is satisfied with threshold probability p provided by the user. The function `extractErrorMetric` in Algorithm 1 repeatedly removes tuples with the largest error and subtracts their corresponding probability while the total probability of the remaining tuples remains at least p. Finally, we return the refined error and its probability, which is at least p but may be higher.

Algorithm 2 Probabilistic range and roundoff error analysis

```
1: procedure evalProbRoundoff(node, dom, prec)
2:     if (node = lhs op rhs) then                          ▷ Binary operation
3:         realRange = evalRange(lhs, rhs, op, dom)
4:         errorLhs = evalProbRoundoff(lhs, realRange, prec)
5:         errorRhs = evalProbRoundoff(rhs, realRange, prec)
6:         propagatedErr = propagate(errorLhs, errorRhs, op)
7:         newRange = toDSI(realRange + propagatedErr)
8:     else                                                 ▷ node is a variable
9:         newRange = toDSI(dom)
10:    for (x, w) ← newRange do
11:        roundoff = maxRoundoff(x, prec)
12:        errDist.append(roundoff, w)
13:    return addNoise(propagatedErr, normalize(errDist))
```

Example Recall our running example, which has a worst-case error of 4.62e-7 in single precision. Using probabilistic subdivision PrAn subdivides the input domain into 100 subdomains and computes a reduced error bound of 2.97e-7 which holds with at least probability 0.85 (the worst-case error remains unchanged). If the program is immune to big errors with probability 0.15, the relevant error bound is thus reduced substantially by 35%. □

3.2 Probabilistic Roundoff Error Analysis

The error computed using only probabilistic interval subdivision is pessimistic as the actual error computation still computes worst-case errors. To compute a tighter distribution of errors at the output we propose to track roundoff errors probabilistically throughout the program. To the best of our knowledge, this is the first sound probabilistic analysis of roundoff errors. We first consider an exact error specification, by which we mean that the roundoff error introduced at each individual arithmetic operation is still the worst-case error, following the IEEE754 standard specification (Equation 1). Even with an exact error specification we obtain an error distribution as the error depends on the ranges, and thus not all values, and not all errors, are equally likely. In this section we first present probabilistic roundoff error analysis and then combine it with probabilistic interval subdivision in order to get tighter error bounds.

The function `evalProbRoundoff` shown in Algorithm 2 extends probabilistic AA to propagate error distributions through the program, by performing a forward dataflow analysis recursively over the program abstract syntax tree (AST). For each AST node (*node*), an input subdomain (*dom*), and a precision (*prec*), it computes the distribution of accumulated roundoff errors. This error distribution is a sound over-approximation, i.e. the true distribution lies within the computed upper and lower bounds. The soundness of our technique follows from the soundness of the underlying probabilistic AA.

For each binary arithmetic operation (*op*), our analysis first computes the real-valued range (line 3) using probabilistic AA as described in Section 2. Computing the accumulated errors is more involved and constitutes our extension over probabilistic AA and one of our main contributions. To compute the accumulated errors, the algorithm first computes the errors of its operands (line 4-5) and from these computes the propagated error (*propagatedErr*) (line 6). Our probabilistic error propagation extends rules from standard AA to probabilistic AA. The propagated errors are added to the real-valued range distribution to obtain the finite-precision distribution (line 7), from which we can compute roundoff errors. Before doing so, we convert the probabilistic affine form to the DSI representation (`toDSI`), which is more amenable for roundoff error computation.

Each focal element of the finite-precision range DSI assigns a probability w to an interval x. Hence, we can compute the roundoff error for each focal element following the floating-point abstraction in Equation 1, and assign the weight w to that error (line 10-12). That is, our procedure computes roundoff errors separately for each focal element, and appends them to generate a new error DSI (line 12). The newly committed error DSI is then normalized to [-1, 1] and added as a fresh noise term to the probabilistic affine form of the propagated errors (line 13).

Probabilistic Error Analysis with Interval Subdivision This probabilistic analysis computes the distribution of the roundoff error at the output, but we observed that it introduces large over-approximations as intervals of the focal elements of the DSI tend to be wide and almost fully overlap with each other. To reduce the over-approximation, we combine the probabilistic error analysis with our probabilistic interval subdivision from Section 3.1. Since each subdomain is smaller, this reduces over-approximations in each, and thus also overall.

Extracting the Error Metric To extract the error metric, we first transform the probabilistic affine form of the error to a DSI structure. To extract the error metric we sort the focal elements, and then remove the elements with largest error magnitude from the error DSI similarly to Section 3.1, until the total weight of the resultant DSI sums up to the threshold p. PrAn returns the maximum absolute value of the remaining focal elements as the refined error. This extraction effectively tries to find as small error bound as possible which will fall within the probability p.

Example Recall the running sine example. Using only the probabilistic error analysis without subdivision, PrAn first discretizes the input distribution into 100 subdomains as shown in Figure 1. From this, PrAn computes the probability distribution of the error shown in Figure 2 a). As the figure shows, the intervals of the DSI are wide and they almost fully overlap with each other. As a result, PrAn cannot refine the error.

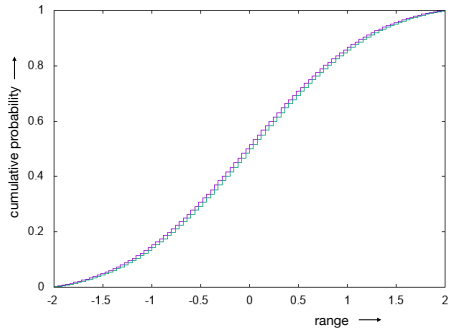


Fig. 1. Input distribution of our example

For the full analysis with probabilistic error analysis and subdivision, we choose 2 initial DSI discretizations and 50 outer interval subdivisions (to keep a fair comparison with 100 overall initial subdivisions). Note that during the computation, the number of DSI discretizations grows and is limited by 100, which we observed to be a good compromise between accuracy and efficiency.

Figure 2 b) shows the resulting error distribution. It is not a step function, because while for each of the subdomain, PrAn does compute the error DSI as a step function, due to the merging *at the end*, the overall distribution does not have this shape. With our full analysis, PrAn computes a refined error of 2.67e-7 (with threshold probability 0.85). This refined error is smaller than the 2.97e-7, which PrAn computed using only probabilistic interval subdivision. □

Alternatively, we could also compute the probability of the largest errors occurring. To extract this information, the error distribution is conceptually subdivided vertically first as shown in Figure 3. In this way we can compute the probability of the large error which will be between the outermost extension and the next subdivision. For this, we need to sum all the weights which intersect with the specified interval. We observed this alternative error specification to be less useful in practice due to large overlaps between the focal elements. We thus focus in the remainder on the first type of error metric.

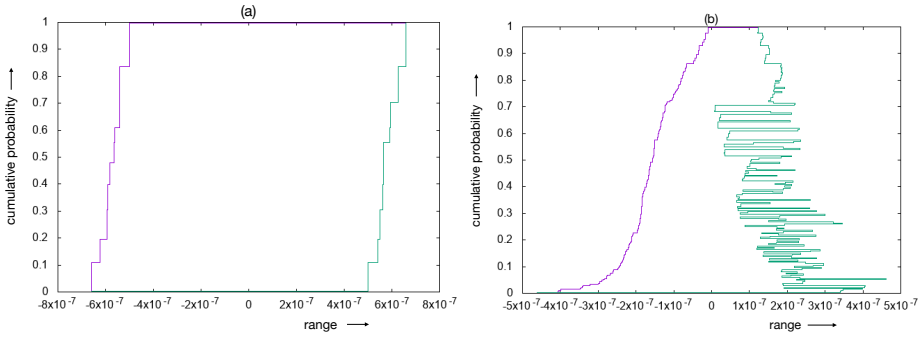


Fig. 2. Error distribution with (a) probabilistic analysis only and (b) full analysis

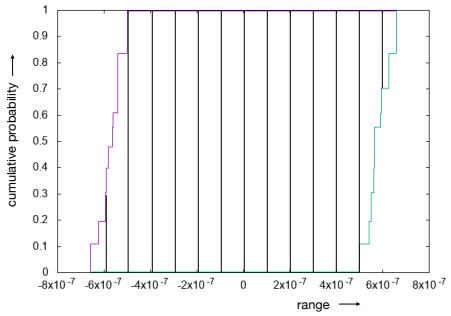


Fig. 3. Alternative vertical subdiv.

3.3 Probabilistic Analysis with Approximate Error Specification

Approximate computing is an emerging design paradigm which exploits approximations to enable better performance in terms of energy or resource utilization. Approximate hardware may, for instance, introduce big, but infrequent, errors in a computation [22,27]. Such approximate error specifications are themselves probabilistic: with a certain probability, the error produced by an operation is larger than the usual worst-case error bound. For such specifications, worst-case analysis considers the large error for all computations, even if it occurs only infrequently, and computes thus highly pessimistic error bounds. With our proposed probabilistic error analysis, we can incorporate such an approximate error specification when the larger errors are bounded with a known upper bound.

To compute a sound probabilistic error bound with approximate specifications we extend the function `maxRoundoff` in Algorithm 2. Previously, we computed one error for each focal element, now we compute multiple errors, depending on the error specification. For example, for a specification where a larger error can occur with a certain probability, we compute two errors and thus two focal elements at line 11 in Algorithm 2: one focal element that keeps track of the big error (with a usually small probability ρ), and one focal element which tracks the regular error (with probability $1 - \rho$).

Example We consider an approximate error specification for our running sine example where an error of size twice the usual machine epsilon ($2 \times \epsilon_m$) occurs with probability $\rho = 0.1$ and the regular roundoff error (ϵ_m) appears with probability 0.9. Our tool PrAn computes 7.34e-7 as the worst-case error with 2 DSI discretizations and 50 outer interval subdivisions. As expected, the absolute error computed with this approximate error specification is larger than the error computed using the exact error specification (4.62e-7) where at each step only an error of ϵ_m appears. Using only probabilistic interval subdivision, PrAn computes a bound of 5.02e-07 with threshold probability 0.85. However, this analysis cannot take advantage of the approximate error specification. Using our full analysis (as described in Section 3.2), PrAn computes a refined error bound of 3.86e-7 with threshold probability 0.85. □

3.4 Implementation

We have implemented our proposed probabilistic error analysis technique(s) in the prototype tool PrAn which is implemented on top of the tool Daisy. The analysis itself is implemented using intervals with arbitrary-precision outer bounds (with outwards rounding) using the GNU MPFR library. This ensures soundness in addition to an efficient implementation. We use GLPK [2] as our simplex solver, which uses floating-point arithmetic internally, and may thus introduce roundoff errors into the analysis. A fully satisfactory solution would use a guaranteed LP-solver as Lurupa [17] or LPex [12], but it is unclear whether the analysis performance would suffer.

4 Experimental Evaluation

Benchmarks We have used various benchmarks from the domains of scientific computing, embedded systems and machine learning. [3]. Most of the benchmarks are widely used in finite-precision roundoff error estimation [31,9,8]: sine, sineOrder3 and sqroot are polynomial approximations of functions, bsplines are used in embedded image applications, train and rigidBody are linear and nonlinear controller respectively. We also have benchmarks where probabilistic AA has been used previously to compute ranges probabilistically [18]: filter is a 2nd order filter unrolled to specified iterations, cubic is obtained from the GNU library, classIDXs are extracted from a linear support vector classifier, neuron is a simplified version of the DNN that learns 'AND' operator. We have further extracted polyIDXs using sklearn-porter [3] from a polynomial support vector classifier trained on the Iris standard data from Python sklearn library.

Experimental Setup The experiments were performed on a Debian desktop computer with a 3.3 GHz i5 processor and 16GB RAM. We use 32 bit single precision as the target precision. We consider a threshold probability of 0.85. We have evaluated PrAn with eight different settings denoted by the letters A-H. Settings (A, B) use only the probabilistic interval subdivision from Section 3.1 and settings (C, D) use our complete probabilistic analysis with subdivision from Section 3.2, assuming an exact error specification. Settings (E, F, G) perform full probabilistic analysis considering an approximate error specification (as in Section 3.3), where with probability 0.9 the error is small (ϵ_m) and with probability 0.1 the error has magnitude $2 \times \epsilon_m$ (settings E, F) or magnitude $4 \times \epsilon_m$ (setting G).

For a fair comparison, in settings A-G, we limit the total number of input subdomains from DSI and input interval subdivision to 100. I.e. when probabilistic AA is used (in C-G), each input DSI is subdivided into 2, and the input interval subdivisions are determined such that the overall number of subdomains remains

[3] All benchmarks are available at `https://people.mpi-sws.org/~dlohar/assets/code/Benchmarks.txt`

	worst-case error			refined error		
	A	C	E	A	C	E
sine	2.40e-7	2.41e-7	4.76e-7	**1.56e-7**	1.68e-7	2.66e-7
sineOrder3	4.62e-7	4.62e-7	7.34e-7	2.97e-7	**2.67e-7**	3.86e-7
sqroot	1.50e-4	1.54e-4	2.42e-4	**8.38e-5**	8.89e-5	1.31e-4
bspline0	8.69e-8	8.69e-8	1.44e-7	**4.36e-8**	4.44e-8	5.51e-8
bspline1	2.09e-7	2.10e-7	3.86e-7	**1.96e-7**	1.97e-7	2.67e-7
bspline2	2.16e-7	2.12e-7	4.21e-7	**1.87e-7**	1.89e-7	2.91e-7
bspline3	5.71e-8	5.71e-8	8.44e-8	**3.33e-8**	3.39e-8	3.72e-8
rigidBody1	1.58e-4	1.73e-4	3.01e-4	**9.99e-5**	1.57e-4	1.98e-4
rigidBody2	1.94e-2	9.70e-3	1.38e-2	1.06e-2	**8.50e-3**	1.05e-2
train1	1.99e-3	2.00e-3	2.94e-3	1.84e-3	**1.67e-3**	2.52e-3
train2	1.37e-3	1.37e-3	2.06e-3	1.32e-3	**1.19e-3**	1.83e-3
train3	2.29e-2	2.29e-2	3.85e-2	2.29e-2*	**2.26e-2**	3.46e-2
train4	2.30e-1	2.30e-1	4.13e-1	2.30e-1*	2.30e-1*	3.75e-1
filter2	1.04e-6	1.04e-6	1.72e-6	8.64e-7	**7.57e-7**	1.13e-6
filter3	2.99e-6	2.87e-6	4.99e-6	2.62e-6	**2.58e-6**	4.52e-6
filter4	6.51e-6	5.20e-6	9.16e-6	6.09e-6	**4.96e-6**	8.69e-6
cubic	1.83e-5	2.02e-5	3.35e-5	**1.73e-5**	1.90e-5	2.80e-5
classIDX0	8.77e-6	9.10e-6	1.45e-5	7.95e-6	**7.92e-6**	1.20e-5
classIDX1	4.63e-6	4.76e-6	7.70e-6	**4.28e-6**	4.38e-6	6.70e-6
classIDX2	7.32e-6	7.60e-6	1.25e-5	**6.35e-6**	6.55e-6	1.02e-5
polyIDX0	5.56e-3	5.80e-3	9.29e-3	2.96e-3	5.32e-3	7.94e-3
polyIDX1	6.81e-4	7.56e-4	1.23e-3	4.51e-4	7.08e-4	1.12e-3
polyIDX2	5.05e-3	5.40e-3	8.73-3	2.84e-3	5.08e-3	7.55e-3
neuron	3.22e-5	7.02e-5	9.87e-5	**3.20e-5**	5.25e-05	7.47e-5

Table 1. Worst-case and refined error in different settings with gaussian inputs

below 100. Note that *during* the computation, the number of DSI subdivisions is limited by 100 (not 2).

Setting (H) considers an approximate error specification as in (G), but performs only probabilistic interval subdivision (Section 3.1). We additionally increase the input interval subdomains to 200. With this setting, we evaluate whether probabilistic AA is helpful for approximate error specifications, or simply using more interval subdivisions (which are relatively cheap) is sufficient.

Settings (B, D, F) consider uniformly distributed inputs and the other settings consider gaussian inputs where all inputs are independent. Note that our approach handles any discretized input distribution provided as a DSI, as well as inputs with (unknown) correlations. Our choice of input distribution is arbitrary, as the 'right' distribution is application specific.

Results Table 1 shows the absolute worst-case and refined error values for setting A, C, and E with gaussian inputs. Settings A and C were able to refine the

	worst-case error			refined error		
	B	D	F	B	D	F
sine	2.72e-7	**2.41e-7**	4.76e-07	2.18e-7	**1.83e-7**	2.66e-7
sineOrder3	4.62e-7	4.62e-7	7.34e-7	3.29e-7	**2.84e-7**	4.04e-7
sqroot	**1.50e-4**	1.54e-4	2.42e-4	**9.02e-5**	9.33e-5	1.39e-4
bspline0	8.69e-8	8.69e-8	1.44e-7	**4.60e-8**	4.77e-8	5.80e-8
bspline1	2.12e-7	**2.10e-7**	3.86e-7	2.00e-7	2.00e-7	2.81e-7
bspline2	2.18e-7	**2.12e-7**	4.21e-7	2.08e-7	**1.93e-7**	2.93e-7
bspline3	5.71e-8	5.71e-8	8.44e-8	3.50e-8	3.50e-8	3.99e-8
rigidBody1	**1.58e-4**	1.73e-4	3.01e-4	**1.50e-4**	1.57e-4	1.98e-4
rigidBody2	1.94e-2	**9.70e-3**	1.38e-2	1.71e-2	**8.55e-3**	1.13e-2
train1	2.00e-3	2.00e-3	2.94e-3	1.91e-3	**1.67e-3**	2.48e-3
train2	1.37e-3	1.37e-3	2.06e-3	1.32e-3	**1.19e-3**	1.80e-03
train3	2.29e-2	2.29e-2	3.85e-2	2.29e-2*	**2.26e-2**	3.46e-2
train4	2.30e-1	2.30e-1	4.13e-1	2.30e-1*	2.30e-1*	3.75e-1
filter1	2.03e-7	2.03e-7	2.62e-7	1.96e-7	**1.86e-7**	1.93e-7
filter2	1.04e-6	1.04e-6	1.72e-6	9.08e-7	**7.74e-7**	1.17e-6
filter3	3.07e-6	**2.87e-6**	4.99e-6	2.96e-6	**2.58e-6**	4.26e-6
filter4	8.23e-6	**5.20e-6**	9.16e-6	8.17e-6	**4.95e-6**	8.69e-6
cubic	**1.85e-5**	2.02e-5	3.35e-5	**1.74e-5**	1.90e-5	2.80e-5
classIDX0	9.10e-6	9.10e-6	1.45e-5	8.52e-6	**7.79e-6**	1.20e-05
classIDX1	4.76e-6	4.76e-6	7.70e-6	4.66e-6	**4.35e-6**	6.79e-6
classIDX2	7.60e-6	7.60e-6	1.25e-5	7.44e-6	**6.36e-6**	1.01e-05
polyIDX0	5.80e-3	**5.19e-3**	9.29e-3	**4.17e-3**	4.78e-3	7.94e-3
polyIDX1	7.55e-4	**5.71e-4**	1.23e-3	7.00e-4	**5.04e-4**	5.04e-4
polyIDX2	5.40e-3	**5.19e-3**	8.38e-3	**3.94e-3**	4.78e-3	7.04e-3
neuron	**6.40e-5**	7.02e-5	9.87e-5	**3.94e-5**	4.86e-5	6.86e-5

Table 2. Worst case and refined error in different settings with uniform inputs

error bounds considering an exact error specification in almost all cases, except those marked with '*'. The train4 benchmark has 9 input variables, which means that with our total limit of subdivisions each input is divided too few times to refine the error. For train3, PrAn could refine the worst-case error only with setting C, i.e. using our full analysis. Using an approximate error specification, PrAn was always able to refine the errors.

For some benchmarks, the full probabilistic error analysis outperforms probabilistic interval subdivision, but for others it is the other way around. When probabilistic error analysis (setting C) is better, it tends to be more significantly better than vice-versa.

We show results with uniform input distributions (settings B, D, F) in Table 2. Settings B and D were also able to reduce the error bounds except the train4 benchmark with uniform inputs. Also with uniform inputs setting B could not reduce the error bound of train3.

	A	C	D	E	G	H		A	C	D	E	G	H
sine	34.9	30.3	24.1	44.1	**52.2**	39.6	*train4*	0.0	0.0	0.0	9.2	**8.3**	0.0
sineOrder3	35.7	42.1	38.5	47.4	**52.9**	34.0	*filter2*	16.9	27.5	25.9	34.5	13.9	**29.7**
sqroot	44.3	42.4	39.5	45.6	**56.6**	45.8	*filter3*	12.4	10.0	10.0	9.6	23.0	**23.8**
bspline0	49.8	48.9	45.1	61.7	**73.1**	56.6	*filter4*	6.5	4.6	4.9	5.1	**47.5**	10.4
bspline1	6.2	6.0	4.7	30.8	**40.2**	9.7	*cubic*	5.8	5.8	5.9	16.2	**41.9**	9.3
bspline2	13.5	11.0	9.4	31.0	**40.6**	17.4	*classIDX0*	9.3	13.0	1.5	17.6	**18.7**	13.6
bspline3	41.6	40.7	38.7	56.0	**67.0**	48.6	*classIDX1*	7.5	8.0	8.6	12.9	5.3	**10.1**
rigidBody1	36.9	8.8	8.8	34.2	34.8	**38.6**	*classIDX2*	13.3	13.8	16.3	18.3	**22.2**	14.8
rigidBody2	45.4	12.4	11.8	24.1	13.5	**49.2**	*polyIDX0*	46.8	8.2	8.0	14.5	14.6	**50.1**
train1	7.5	16.4	16.4	14.4	**20.2**	7.2	*polyIDX1*	33.8	6.3	11.7	8.7	10.6	**37.3**
train2	3.3	12.6	12.7	11.3	**13.6**	1.9	*polyIDX2*	43.9	5.9	8.0	13.6	19.6	**49.2**
train3	0.0	1.3	1.3	10.1	**11.2**	2.2	*neuron*	0.9	25.3	30.8	24.3	**41.7**	13.9

Table 3. Reductions in % in errors w.r.t. the standard worst-case in different settings

In general, which analysis is better is application specific: this depends on the over-approximations committed, which in turn depend on the operations and ranges of an application. A user should consider both variants and use the best result in a portfolio-like approach. Note, however, that for approximate error specifications, a probabilistic error analysis is necessary in order to adequately capture the probabilistic error specification.

We show the error reduction between the refined and the corresponding worst-case errors in Table 3. The reductions for the exact error specifications are on average 20.9%, 17% and 16.2% with settings A, C and D, respectively. The reduction with gaussian inputs in setting C is in most cases higher than for uniform inputs. We also see that reductions are application specific, and for many benchmarks more substantial than the averages suggest. In some cases our probabilistic analysis even with exact specifications allows to report refined errors with nearly half the magnitude as the worst-case (up to 49.8%), but still with guaranteed probability of at least 0.85.

Our probabilistic analysis achieves even higher reductions with the approximate error specification (settings E and G) with on average 24.9% and 30.6% smaller refined errors than worst-case, respectively. Even if we double the number of total subdivisions (setting H) probabilistic interval subdivision can only reduce the error on average by 25.4%, compared to 30.6% using our full probabilistic analysis. While for a few cases setting H outperforms setting G (because of overapproximations in the probabilistic analysis), overall probabilistic analysis is successful at capturing the fact that large errors only occur infrequently.

Finally, Table 4 shows the running times of our analysis for settings A, C and E (averaged over 3 runs). The probabilistic error analysis takes more time than the non-probabilistic analysis (i.e. one with only a probabilistic interval subdivision), as expected. We note, however, that the analysis times are nonetheless acceptable for a static analysis which is run only once.

	A	C	E		A	C	E
sine	9s	6m 1s	109m 11s	*train4*	0.1s	22s	36s
sineOrder3	5s	3m 7s	6m 46s	*filter2*	0.9s	1m 47s	3m 2s
sqroot	1s	1m 29s	33m 56s	*filter3*	0.9s	4m 19s	11m 16s
bspline0	0.4s	22s	12m 2s	*filter4*	16s	10m 54s	28m 39s
bspline1	0.6s	42s	13m 37s	*cubic*	16s	3m 11s	14m 24s
bspline2	0.8s	50s	16m 6s	*classIDX0*	23s	3m 39s	7m 44s
bspline3	0.4s	18s	1m 20s	*classIDX1*	2s	3m 56s	7m 53s
rigidBody1	0.1s	35s	1m 16s	*classIDX2*	0.1s	3m 39s	7m 1s
rigidBody2	0.4s	1m 14s	10m 59s	*polyIDX0*	0.5s	32m 52s	181m 45s
train1	0.3s	1m 15s	3m 55s	*polyIDX1*	0.8s	33m 26s	193m 19s
train2	0.3s	6m 29s	11m 22s	*polyIDX2*	3s	38m 29s	177m 38s
train3	0.1s	12s	25s	*neuron*	0.5s	1m 11s	4m 17s

Table 4. Analysis time (averaged over 3 runs) in different settings

5 Related Work

Probabilistic affine arithmetic with Dempster-Shafer Interval structures provide an efficient approach for soundly propagating probability distributions, but it incurs huge over-approximation of the probabilities. An alternative would be exact probabilistic inference [21], however its scalability is very limited [18]. Most probabilistic inference algorithms rely on sampling [25,6,20,26] and thus do not provide guaranteed bounds. Probabilistic affine arithmetic has been also augmented by concentration of measure inequalities [5], which may reduce the amount of over-approximations. It has been used for tracking real-valued ranges in a program (but not errors). This could potentially also help improve the accuracy of our approach, but since the implementation is not available, we leave this to future work.

Sankaranarayanan et al. [28] verify probabilistic properties of programs with many paths with a combination of symbolic execution and volume computation, but do not consider (finite-precision) errors and non-linear programs.

Chisel [22] considers and bounds the probability of errors occurring due to approximate hardware, and is thus in spirit similar to our approach. Chisel, however, only tracks the probability of a large error occurring, whereas our approach provides a more nuanced probability *distribution*, and furthermore tracks the variable ranges probabilistically as well.

Several tools exist for soundly bounding roundoff errors, some are based on affine arithmetic [9,8,15,11,7], and others on a global optimization approach [31,19,24]. The tools only compute worst-case roundoff errors, however.

Daumas et al. [10] compute bounds on the probability that accumulated floating-point roundoff errors exceed a given threshold, using known inequalities on sums of probability distributions, and distributions on the individual errors. This approach requires manual proofs and does not consider input distributions.

Statistical error analyses have also been proposed. For instance, the CADNA approach [29] computes a confidence interval on the number of exact digits using repeated simulation with random rounding modes. An approach similar in spirit is to perturb the low-order bits, or rewrite expressions based on real-valued identities to uncover instabilities of programs [32]. The approaches, however, do not provide sound error guarantees.

6 Conclusion

We have presented a probabilistic analysis for tracking errors due to finite-precision arithmetic in straight-line code. Instead of worst-case errors, as usual analyses compute, our analysis computes probability distributions and extracts refined error metrics which determine that a potentially smaller error than the worst-case is satisfied with some probability. We believe that this analysis can be useful for applications which can tolerate larger errors, as long as their probability is bounded. We observe that our refinement is even more useful in the case of probabilistic error specifications common in today's approximate hardware.

References

1. IEEE Standard for Floating-Point Arithmetic. IEEE Std 754-2008 (2008)
2. GLPK. `https://www.gnu.org/software/glpk/` (2012)
3. Project Sklearn-porter. `https://github.com/nok/sklearn-porter` (2018)
4. Bouissou, O., Goubault, E., Goubault-Larrecq, J., Putot, S.: A Generalization of P-boxes to Affine Arithmetic. Computing **94**(2-4), 189–201 (2012)
5. Bouissou, O., Goubault, E., Putot, S., Chakarov, A., Sankaranarayanan, S.: Uncertainty Propagation using Probabilistic Affine Forms and Concentration of Measure Inequalities. In: TACAS (2016)
6. Carpenter, B., Gelman, A., Hoffman, M., Lee, D., Goodrich, B., Betancourt, M., Brubaker, M., Guo, J., Li, P., Riddell, A.: Stan: A Probabilistic Programming Language. Journal of Statistical Software, Articles **76**(1), 1–32 (2017)
7. Damouche, N., Martel, M., Chapoutot, A.: Improving the numerical accuracy of programs by automatic transformation. International Journal on Software Tools for Technology Transfer **19**(4), 427–448 (2017)
8. Darulova, E., Izycheva, A., Nasir, F., Ritter, F., Becker, H., Bastian, R.: Daisy - Framework for Analysis and Optimization of Numerical Programs. In: TACAS (2018)
9. Darulova, E., Kuncak, V.: Towards a Compiler for Reals. TOPLAS **39**(2) (2017)
10. Daumas, M., Lester, D., Martin-Dorel, E., Truffert, A.: Improved bound for stochastic formal correctness of numerical algorithms. Innovations in Systems and Software Engineering **6**(3), 173–179 (2010)
11. De Dinechin, F., Lauter, C.Q., Melquiond, G.: Assisted Verification of Elementary Functions using Gappa. In: ACM Symposium on Applied Computing (2006)
12. Dhiflaoui, M., Funke, S., Kwappik, C., Mehlhorn, K., Seel, M., Schömer, E., Schulte, R., Weber, D.: Certifying and Repairing Solutions to Large LPs. How Good are LP-Solvers? In: SODA. pp. 255–256 (2003)

13. Ferson, S., Kreinovich, V., Ginzburg, L., Myers, D.S., Sentz, K.: Constructing Probability Boxes and Dempster-Shafer Structures. Tech. rep., Sandia National Laboratories (2003)
14. de Figueiredo, L.H., Stolfi, J.: Affine Arithmetic: Concepts and Applications. Numerical Algorithms **37**(1-4) (2004)
15. Goubault, E., Putot, S.: Static Analysis of Finite Precision Computations. In: VMCAI (2011)
16. Izycheva, A., Darulova, E.: On sound relative error bounds for floating-point arithmetic. In: FMCAD (2017)
17. Keil, C.: Lurupa - Rigorous Error Bounds in Linear Programming. In: Algebraic and Numerical Algorithms and Computer-assisted Proofs. No. 05391 in Dagstuhl Seminar Proceedings (2006), http://drops.dagstuhl.de/opus/volltexte/2006/445
18. Lohar, D., Darulova, E., Putot, S., Goubault, E.: Discrete Choice in the Presence of Numerical Uncertainties. EMSOFT (2018)
19. Magron, V., Constantinides, G., Donaldson, A.: Certified Roundoff Error Bounds Using Semidefinite Programming. ACM Trans. Math. Softw. **43**(4) (2017)
20. Minka, T., Winn, J., Guiver, J., Webster, S., Zaykov, Y., Yangel, B., Spengler, A., Bronskill, J.: Infer.NET 2.6. http://research.microsoft.com/infernet (2014)
21. Misailovic, S., Vechev, M., Gehr, T.: PSI: Exact Symbolic Inference for Probabilistic Programs . In: CAV (2016)
22. Misailovic, S., Carbin, M., Achour, S., Qi, Z., Rinard, M.C.: Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels. In: OOPSLA (2014)
23. Moore, R.: Interval Analysis. Prentice-Hall (1966)
24. Moscato, M., Titolo, L., Dutle, A., Muñoz, C.: Automatic Estimation of Verified Floating-Point Round-Off Errors via Static Analysis. In: SAFECOMP (2017)
25. Nori, A.V., Hur, C.K., Rajamani, S.K., Samuel, S.: R2: An Efficient MCMC Sampler for Probabilistic Programs. In: AAAI (2014)
26. Sampson, A., Panchekha, P., Mytkowicz, T., McKinley, K.S., Grossman, D., Ceze, L.: Expressing and Verifying Probabilistic Assertions. In: PLDI (2014)
27. Sampson, A., Dietl, W., Fortuna, E., Gnanapragasam, D., Ceze, L., Grossman, D.: Enerj: Approximate data types for safe and general low-power computation. In: PLDI (2011)
28. Sankaranarayanan, S., Chakarov, A., Gulwani, S.: Static Analysis for Probabilistic Programs: Inferring Whole Program Properties from Finitely Many Paths. In: PLDI (2013)
29. Scott, N.S., Jézéquel, F., Denis, C., Chesneaux, J.M.: Numerical 'health check' for scientific codes: the CADNA approach. Computer Physics Communications **176**(8), 507–521 (2007)
30. Shafer, G.: A Mathematical Theory of Evidence. Princeton university press (1976)
31. Solovyev, A., Jacobsen, C., Rakamaric, Z., Gopalakrishnan, G.: Rigorous Estimation of Floating-Point Round-off Errors with Symbolic Taylor Expansions. In: FM (2015)
32. Tang, E., Barr, E., Li, X., Su, Z.: Perturbing numerical calculations for statistical analysis of floating-point program (in)stability. In: ISSTA (2010)
33. Xu, Q., Mytkowicz, T., Kim, N.S.: Approximate computing: A survey. IEEE Design Test **33**(1), 8–22 (Feb 2016)