Trustworthy Numerical Computation in Scala

Eva Darulova Viktor Kuncak *

School of Computer and Communication Sciences (I&C) - Swiss Federal Institute of Technology (EPFL), Switzerland

firstname.lastname@epfl.ch

Abstract

Modern computing has adopted the floating point type as a default way to describe computations with real numbers. Thanks to dedicated hardware support, such computations are efficient on modern architectures, even in double precision. However, rigorous reasoning about the resulting programs remains difficult. This is in part due to a large gap between the finite floating point representation and the infinite-precision real-number semantics that serves as the developers' mental model. Because programming languages do not provide support for estimating errors, some computations in practice are performed more and some less precisely than needed.

We present a library solution for rigorous arithmetic computation. Our numerical data type library tracks a (double) floating point value, but also a guaranteed upper bound on the error between this value and the ideal value that would be computed in the real-value semantics. Our implementation involves a set of linear approximations based on an extension of affine arithmetic. The derived approximations cover most of the standard mathematical operations, including trigonometric functions, and are more comprehensive than any publicly available ones. Moreover, while interval arithmetic rapidly yields overly pessimistic estimates, our approach remains precise for several computational tasks of interest. We evaluate the library on a number of examples from numerical analysis and physical simulations. We found it to be a useful tool for gaining confidence in the correctness of the computation.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification

General Terms Algorithms, Languages, Verification

* This research is supported by the Swiss NSF Grant #200021_132176.

OOPSLA'11, October 22–27, 2011, Portland, Oregon, USA.

1. Introduction

Numerical computation has been one of the driving forces in the early development of computation devices. Floating point representations have established themselves as a default data type for implementing software that approximates real-valued computations. Today, floating-point-based computations form an important part of scientific computing applications, but also of cyber-physical systems, which reason about the quantities describing the physical world in which they are embedded.

The IEEE standard [59] establishes a precise interface for floating point computation. Over the past years, it has become a common practice to formally verify the hardware implementing this standard [28, 47, 55]. On the other hand, the software using floating point arithmetic remains difficult to reason about. As an example, consider the experiment in N-version programming [29], in which the largest discrepancies among different software versions were found in numerical computation code.

One of the main difficulties in dealing with numerical code is understanding how the approximations performed by the individual arithmetic operation steps (precisely specified by the standard) compose into an overall error of a complex computation. Such roundoff errors can accumulate to the point where the computed value is no longer a precise enough approximation of the real value. Currently, the developers have no reliable automated method to determine these approximation errors. It is striking that, for many important program correctness properties we now have type systems and static analyzers that can establish them over *all* executions [7, 15, 19, 22, 36, 37, 54], whereas for numerical errors, we lack practical methods to estimate errors even for *one, given,* execution. Namely, it is difficult to determine whether a given execution is correct in the sense that its final result is close to the result of the corresponding sequence of operations in real analysis. In program analysis terminology, we observe a great desire in the community to expand *verification* techniques to numerical code [39]. At the same time, we do not even know yet how to *test* numerical code. Compounding the problem is that the correctness of these applications is difficult to asses by manual inspection (whether we try to inspect the source code or the computed result). Both formal and informal reasoning about programs with floating-points is therefore challenging. As a result, we have little confidence in the correctness of an increasingly important set of applications that reason about the real world.

To remedy this unfortunate situation, we introduce an easy-to-use system for estimating roundoff errors. Our system comes in the form of new data types for the Scala [49] programming language. These data types act as a drop-in replacement for the standard floating point data types, such as Double. They offer support for a comprehensive range of operations, with a greater precision than in any previously documented solution. We deploy our techniques in Scala for easy use on many platforms, although they apply in any programming language using floating-point computation.

When faced with the problem of floating point approximation errors, many existing approaches propose interval arithmetic (IA) as a solution. However, intervals give too pessimistic estimates in many cases. The problem is easy to demonstrate; its essence can be seen already on a very simple example. If x is an interval $[0, a]$, then interval arithmetic approximates the expression $x - x$ with $[-a, a]$, although it is, in fact, always equal to zero. Essentially, interval arithmetic approximates $x-x$ in the same way as it would approximate $x - y$ when x and y are unrelated variables that both belong to $[0, a]$. Furthermore, when such approaches are used to estimate the behavior over a range of input values, they fail to distinguish two sources of uncertainty:

- uncertainty in the error between the ideal and the floating point value;
- uncertainty in the actual values of floating point variables when analyzing code, if the initial values can belong to any point in a given interval.

Any approach that lumps together these two sources of uncertainty will quickly become imprecise.

To avoid the above problems, we first examine *affine arithmetic*, which was introduced in [17] and can more precisely track the relations between variables. It turns out, however, that affine arithmetic by itself cannot be as easily adapted for reasoning about roundoff errors as interval arithmetic, because it uses mid-points of intervals for its estimates of nonlinear functions, and roundoff errors in the end-points of intervals can be greater than for the mid-point value. We describe a methodology that we used to derive the appropriate sound approximations. (The actual approximation rules that we use are publicly available in our system's source code.) Building on these, we define a data type that tracks a floating-point computation and provides, in addition to the computed value, a guaranteed estimate on the roundoff error committed. Furthermore, we introduce an approach that allows the library to track errors over a range of values. We can therefore answer both of the following questions:

- What is an upper bound on the roundoff error of the result of a floating-point computation run, for a concrete input?
- What is the maximum roundoff error of the result, for inputs ranging over a given input interval?

By introducing a freely available library that addresses these questions, we provided developers and researchers with an easy-to-use tool that helps them understand floating-point properties of code, a tool that provides sound guarantees on the floating-point roundoff errors committed. We expect that our system can be easily integrated into verification and testing systems in the future.

Contributions. We make the following contributions:

- We develop and implement an AffineFloat data type that supports testing of concrete numerical computations against their real-valued semantics. Our data type computes practically useful error bounds while retaining compatibility with the standard Double data type: not only are the operations entirely analogous, but the underlying Double value that it computes is identical to the one computed with the standard Double type alone. This compatibility is important in practice, but requires changes to the way roundoff errors and affine forms are supported compared to the existing techniques. As a safe-guard, our technique falls back onto intervals when the linear approximation is not appropriate. Furthermore, our solution goes beyond the (very few) available affine arithmetic implementations by accurately supporting a substantial set of non-linear and transcendental functions. The library also implements a technique to soundly bound the number of affine error terms, ensuring predictable performance without sacrificing much precision.
- We develop and implement a SmartFloat data type that generalizes AffineFloat to estimate upper bounds on roundoff errors over an *entire range of input values*. SmartFloat also accepts user-specified errors on input variables (arising from, e.g. physical measurements, or iterative numerical methods). Thanks to SmartFloat, the developer can show, using a single program run, that the roundoff error within the entire interval remains small. Existing methods that merge initial interval width with roundoff estimates cannot perform such estimates. We also provide a *nested affine* implementation, which uses a linear function of input to represent error terms themselves. This technique provides an improved estimate of relative errors for the input ranges that contain zero.
- We evaluate the precision and performance of our implementation on a number of benchmarks from physics simulations and numerical analysis, including: Nbody and spring simulations, spectral norm computation, the Scimark, Fbench and Whetstone benchmarks [4, 42, 50, 61]. The results show that our library produces (possibly after an initial interval subdivision) precise estimates that would otherwise require expensive constraint solving techniques. It also shows that the library scales to long-running computations.

Our implementation is available at `http://lara.epfl.ch`.

Paper outline. We continue by illustrating our system through two examples. We then provide a quick overview of the basic affine arithmetic approach (Section 3). This background gives the high-level idea of the approach, but is not sufficient to obtain our results. We characterize the precision and the performance of our implementation in Section 4. We show further applications enabled by our system in Section 5.

We then present the new techniques that we introduced to achieve our results: first for AffineFloat (Section 6) and then for SmartFloat (Section 7). We describe the integration into Scala in Section 8, finishing with related work and conclusions.

2. Examples

Cube root. Intervals have the unfortunate property of ignoring correlations between variables and thus often over-approximate roundoff errors by far too much to be useful. As an illustration, consider the following code fragment that uses Halley's method [57] to compute the cube root of a = 10, starting from an initial value of xn = 1.6:

```
for (i ← 1 until 5)
  xn = xn ∗ ((xn∗xn∗xn + 2.0∗a) / (2.0∗xn∗xn∗xn + a))
```

Compare the results computed with Double against the result to 30 digits precision from a popular computer algebra system (denoted CAS), and the result returned by interval arithmetic:

Double	2.15443469003**18834**
CAS	2.15443469003**1883721**...
Interval	[2.15443469003**17617**,
	2.154434690032**2006**]
Affine	2.15443469003**18834** $\pm 1.34 \cdot 10^{-15}$

It turns out that the Double value differs from the true real result only in the very last digit, which amounts to an absolute error on the order of unit in the last place, $\approx 4.44 * 10^{-16}$. Interval arithmetic however, would quantify this error as $\approx 1.23 * 10^{-13}$. On the other hand, using our affine-arithmetic-based type we compute an absolute error of $1.34 * 10^{-15}$, which is (by the correctness of our approach) sound, yet two decimal orders of magnitude more precise than the result in interval arithmetic. If we relied only on intervals, we might be led to believe that we cannot compute the value with the desired precision using Halley's method. We might have thus decided to (unnecessarily) adopt a more expensive computational method, even though Halley's method actually worked fine.

Area of a triangle. As another example, consider the code in Figure 1. triangleTextbook computes the area of a triangle using the well-known textbook formula. On the other hand, triangleKahan uses an improved version by Kahan [34]. Running both versions with our SmartFloat type and with intervals, we get the results listed in Table 1.

```
def triangleTextbook(a: SmartFloat,
                     b: SmartFloat,
                     c: SmartFloat): SmartFloat = {
  val s = (a + b + c)/2.0
  sqrt(s ∗ (s − a) ∗ (s − b) ∗ (s − c))
}

def triangleKahan(a: SmartFloat, b: SmartFloat,
                  c: SmartFloat): SmartFloat = {
  if(b < a) {
    val t = a
    if (c < b) { a = c; c = t }
    else {
      if (c < a) { a = b; b = c; c = t }
      else { a = b; b = t }
  } } else if (c < b) {
    val t = c; c = b;
    if (c < a) { b = a; a = t }
    else { b = t }
  }
  sqrt((a+(b+c)) ∗ (c−(a−b)) ∗ (c+(a−b))
                  ∗ (a+(b−c))) / 4.0
}
```

Figure 1. Code for computing the area of a triangle using the classic textbook formula and Kahan's improved version. The latter sorts the triangle sides by their lengths (a being the smallest) and refactors the final formula such that computations are minimized, and performed in an order that minimizes precision loss.

Interval Arithmetic	area	rel.roundoff
triangleTextbook		
a = 9.0, b = c = [4.71, 4.89]	[6.00, 8.96]	?
a = 9.0, b = c = [4.61, 4.79]	[4.32, 7.69]	?
a = 9.0, b = c = [4.501, 4.581]	[0.42, 3.93]	?
triangleKahan		
a = 9.0, b = c = [4.71, 4.89]	[6.13, 8.79]	?
a = 9.0, b = c = [4.61, 4.79]	[4.41, 7.54]	?
a = 9.0, b = c = [4.501, 4.581]	[0.42, 3.89]	?
SmartFloat		
triangleTextbook		
a = 9.0, b = c = [4.71, 4.89]	[6.25, 8.62]	1.10e-14
a = 9.0, b = c = [4.61, 4.79]	[4.50, 7.39]	1.97e-14
a = 9.0, b = c = [4.501, 4.581]	[0.41, 3.86]	1.95e-12
triangleKahan		
a = 9.0, b = c = [4.71, 4.89]	[6.25, 8.62]	3.11e-15
a = 9.0, b = c = [4.61, 4.79]	[4.49, 7.39]	5.26e-15
a = 9.0, b = c = [4.501, 4.581]	[0.39, 3.86]	5.07e-13

Table 1. Area and relative roundoffs computed on the code from Figure 1 with SmartFloat and intervals for selected values.

Although interval arithmetic does not over-approximate the range by much more than affine arithmetic on this particular example, it fails to quantify the roundoff errors. Based only on intervals, it is impossible to tell that one version of the code behaves better than the other. Our SmartFloat on the other hand shows an improvement of about one order of magnitude in favor of Kahan's formula. Also note that the computed roundoff errors indicate that for thin triangles relative roundoff errors grow, which is indeed what happens. This illustrates that our library allows for both formal reasoning (by establishing correspondence to real-valued semantics), as well as high-level informal reasoning and analysis.

Using our implementation of SmartFloat's, we obtain not only a more accurate interval for the result, but in fact an upper bound on the error across the entire input interval. In interval arithmetic, one could in principle use the width of the actual interval as the roundoff error bound, but this would yield unrealistically large errors. In this particular example the bound on roundoff errors is more than 10^{14} times smaller than the actual width of the interval in which the output ranges! Therefore, any attempt to use an interval-like abstraction to simultaneously represent 1) the input range and 2) the error bound, will spectacularly fail. In contrast, our technique distinguishes these different quantities, and is among the first ones to do so. Thanks to this separation, it can establish that roundoff error is small even though the interval is relatively large.

3. A Quick Tour of Interval and Affine Arithmetic

Throughout this paper, we use the following general notation:

- $\mathbb{F}$ denotes floating-point values; if not otherwise stated, in double precision (64 bit).
- $\mathbb{R}$ denotes (mathematical) real numbers.
- $\mathbb{IF}, \mathbb{IR}$ denote the sets of all closed intervals of floating-point and real numbers, respectively; an interval is given by its two endpoints.
- $[a]$ is a notation to denote the interval represented by an expression a, according to some specified semantics.
- $\downarrow x \downarrow$ and $\uparrow x \uparrow$ denote the result of some expression x rounded towards $-\infty$ or $+\infty$ respectively. That is, if $x \in \mathbb{R}$ is a number not representable in a given floating point format, $\downarrow x \downarrow$ evaluates to the next smaller number representable in binary. Similarly, $\uparrow x \uparrow$ evaluates to the nearest larger floating-point number. If x has a floating point representation, then $\downarrow x \downarrow = \uparrow x \uparrow = x$.

3.1 IEEE Floating-point Arithmetic

Throughout this paper we assume that floating-point arithmetic conforms to the IEEE 754 floating-point standard [59]. Recent general-purpose CPUs conform to it, and it is also generally respected in main programming languages. The JVM (Java Virtual Machine), on which Scala runs, supports single- and double-precision floating-point values according to the standard, as well as rounding-to-nearest rounding mode [41]. Also by the standard, the basic arithmetic operations $\{+, -, *, /, \sqrt{}\}$ are rounded correctly, which means that the result from any such operation must be the closest representable floating-point number. Hence, provided there is no overflow, the result of a binary operation in floating-point arithmetic $\circ_F$ satisfies

$$x \circ_F y = (x \circ_R y)(1+\delta), \;\; |\delta| \leq \epsilon_M, \;\; \circ \in \{+,-,*,/\} \quad (1)$$

where $\circ_R$ is the ideal value in real numbers and ϵ_M is the machine epsilon that determines the upper bound on the relative error. This model provides a basis for our roundoff error estimates.

Thanks to dedicated hardware floating-point units, floating-point computations are fast, and our library is currently set up for double-precision floating-point values (i.e. $\epsilon_M = 2^{-53}$). This is also the precision of choice for most numerical algorithms. It is straightforward to adapt our techniques for single precision, or any other precision with an analogous semantics.

3.2 Interval Arithmetic

One possibility to perform guaranteed computations in floating-point arithmetic is to use standard interval arithmetic [48]. Interval arithmetic computes a bounding interval for each basic operation as

$$x \circ_F y = [\downarrow(x \circ y)\downarrow, \uparrow(x \circ y)\uparrow] \quad (2)$$

Rounding outwards guarantees that the interval always contains the real result and thus ensures soundness. The error for square root is computed analogously.

Section 2 already illustrated how quickly interval arithmetic becomes imprecise. This is a widely recognized phenomenon; to obtain a more precise approximation, we therefore use affine arithmetic.

3.3 Affine Arithmetic

Affine arithmetic was originally introduced in [17] and developed to compute ranges of functions over the domain of reals, with the actual calculations done in double (finite) precision. Affine arithmetic addresses the difficulty of interval arithmetic in handling correlations between variables. It is one possible *range-based method* to address this task; we discuss further methods in Section 9.

Given a function $f : \mathbb{R} \rightarrow \mathbb{R}$, we wish to compute its approximation in terms of floating-point numbers. Let A be a set of representations of intervals, with $[a] \in \mathbb{IR}$ for $a \in A$. The goal is then to compute an approximation of f by a function $g : A \rightarrow A$ that satisfies the *fundamental invariant of range analysis*:

PROPERTY 1. *For $a \in A$, $x \in \mathbb{R}$, if $x \in [a]$, then*

$$f(x) \in [g(a)]$$

Note that a range-based arithmetic as such does necessarily attempt to quantify the roundoff errors itself. A possible application of affine arithmetic, as originally proposed, is finding zeroes of a function in a given initial interval. The idea is to bisect the initial interval and to compute an estimate of the function value over each subdomain. If the output range for one subdomain does not include a zero, then that part of the domain can be safely discarded.

Affine arithmetic represents possible values of variables as affine forms

$$\hat{x} = x_0 + \sum_{i=1}^{n} x_i \epsilon_i$$

where x_0 denotes the *central value* (of the represented interval) and each *noise symbol* ϵ_i is a formal variable denoting a deviation from the central value, intended to range over $[-1, 1]$. The maximum magnitude of each *noise term* is given by the corresponding x_i. Note that the sign of x_i does not matter in isolation, it does, however, reflect the relative dependence between values. For example, take $x = x_0 + x_1\epsilon_1$, then in real number semantics,

$$\begin{aligned} x - x &= x_0 + x_1\epsilon_1 - (x_0 + x_1\epsilon_1) \\ &= x_0 - x_0 + x_1\epsilon_1 - x_1\epsilon_1 = 0 \end{aligned}$$

If we subtracted $x = x_0 - x_1\epsilon_1$ instead, the resulting interval would have width $2 * x_1$ and not zero.

The range represented by an affine form is computed as

$$[\hat{x}] = [x_0 - rad(\hat{x}), x_0 + rad(\hat{x})], \qquad rad(\hat{x}) = \sum_{i=1}^{n} |x_i|$$

When implementing affine forms in a program, we need to take into account that some operations are not performed exactly, because the central value and the coefficients need to be represented in some finite (e.g. double) precision. As suggested in [17], the roundoff errors committed during the computation can be added with a new fresh noise symbol to the final affine form. A general affine operation $\alpha\hat{x} + \beta\hat{y} + \zeta$ consists of addition, subtraction, addition of a constant (ζ) or multiplication by a constant (α, β). Expanding the affine forms $\hat{x}$ and $\hat{y}$ we get

$$\alpha\hat{x} + \beta\hat{y} + \zeta = (\alpha x_0 + \beta y_0 + \zeta) + \sum_{i=1}^{n} (\alpha x_i + \beta y_i)\epsilon_i + \iota\epsilon_{n+1} \tag{3}$$

with $\alpha, \beta, \zeta \in \mathbb{F}$ and where ι denotes the accumulated *internal errors*, that is, the roundoff errors committed when computing the individual terms of the new affine form.

Each operation carries a roundoff error and all of them must be taken into account to achieve the parameter ι for the rigorous bound. The challenge hereby consists of accounting for all roundoff errors, but still creating a tight approximation. While for the basic arithmetic operations the roundoff can be computed with Equation 1, there is no such simple formula for calculating the roundoff for composed expressions (e.g. $\alpha * x_0 + \zeta$). We determine the maximum roundoff error of an expression $f(v_1, \ldots, v_m)$ using the following procedure [17]:

$$\begin{aligned} z &= f(v_1, v_2, \ldots, v_m) \\ z_{-\infty} &= \downarrow f(v_1, v_2, \ldots, v_m) \downarrow \\ z_{+\infty} &= \uparrow f(v_1, v_2, \ldots, v_m) \uparrow \\ \iota &= \max(z_{+\infty} - z, z - z_{-\infty}) \end{aligned}$$

That is, the program computes three results: (1) the floating-point result z using rounding to-nearest, (2) the result $z_{-\infty}$ assuming worst-case roundoff errors when rounding towards $-\infty$, and the analogous result $z_{+\infty}$ with rounding towards $+\infty$ at each step. As the worst-case committed roundoff error ι we use the maximum difference (ι) between these values.

A possible use of affine arithmetic for keeping track of roundoff errors is to represent each double precision value by an affine form. That is, the actually computed double precision value is equal to the central value and the noise terms collect the accumulating roundoff errors. One expects to obtain tighter bounds than with interval arithmetic, especially when a computation exhibits many correlations between variables. However, a straightforward application of affine arithmetic in the original formulation is not always sound, as we show in Section 6. Namely, the standard affine arithmetic takes the liberty of choosing a convenient central value in a range, which does not preserve the compatibility with Double. Using such computation on non-affine operations (such as division or trigonometric functions) can shift the central value away from the actually computed Double value. Roundoff errors computed using this method would be those of a different computation and would thus be unsound. Our implementation therefore provides a modified approximation that ensures soundness.

Before proceeding with the description of the technique we use in our solution, we show in the next section how the library behaves in practice. The library provides two data types, AffineFloat and SmartFloat, that replace ordinary floating-point numbers in a program, track a computation and provide estimates on the roundoff errors committed. An AffineFloat variable represents exactly one floating-point number and thus replaces the computation one-to-one, a SmartFloat variable represents a *range* of values and computes the maximum roundoff error over this range. The technical implications of this difference will be described in detail in Section 6 and Section 7. Note that whereas SmartFloat must compute worst-case roundoff errors over the entire interval, AffineFloat only needs to do this for one value. AffineFloat can therefore generally provide a more precise estimate.

4. Evaluation of Precision and Performance

We have selected several benchmarks for evaluating our library. Many of them were originally written in Java or C; we ported them to Scala as faithfully as possible. Once written in Scala, we found that changing the code to use our AffineFloat type instead of Double is a straightforward process and needs only few manual edits. Scala compiler's type checker was particularly helpful in this process.

Many of the existing benchmarks we adopted were originally developed for *performance* and not *numerical precision* evaluation. We hope that our library and examples will stimulate further benchmarking with precision in mind.

The benchmarks we present are the following:[1]

Nbody simulation is a benchmark from [4] and is a simulation that "should model the orbits of Jovian planets, using [a] (...) simple symplectic-integrator".

Spectral norm is a benchmark from [4] and "should calculate the spectral norm of an infinite matrix A, with entries $a_{11} = 1, a_{12} = \frac{1}{2}, a_{21} = \frac{1}{3}, a_{13} = \frac{1}{4}, a_{22} = \frac{1}{5}, a_{31} = \frac{1}{6}$, etc."

Scimark [50] is a set of Java benchmarks for scientific computations. We selected three benchmarks that best suit our purpose: the Fast Fourier Transform (FFT), Jacobi Successive Over-relaxation (SOR) and a dense LU matrix factorization to solve the matrix equation $Ax = b$. The exact dimensions of the problems we used are noted in Table 6.

Fbench was orginally written by Walker [60] as a "Trigonometry Intense Floating Point Benchmark". We used the Java port [61] for our tests.

Whetstone [42] is a classic benchmark for performance evaluation of floating-point computations.

Spring simulation is our own code from Figure 3, however for benchmarking we removed the added method errors.

We have also implemented an interval arithmetic type that can, in the same way as AffineFloat and SmartFloat, replace all Double types in a program. This type is used throughout this paper when comparing our library to interval arithmetic.

4.1 AffineFloat Precision

Because AffineFloats represent exactly one floating-point value, we can compare its precision in computing roundoff errors to that of interval arithmetic, where each value is analogously represented by one interval. The width of the resulting interval provides the roundoff error.

Table 2 presents our measurements of precision on three of our benchmarks. These results provide an idea on the order of magnitude of roundoff error estimates, as well as the scalability of our approach. For the Nbody problem we compute the energy at each step, which changes due to method errors but also due to accumulated roundoffs. For the Spectral norm we measure the roundoff error of the result after different numbers of iterations. In the case of SOR, the reported errors are average relative errors for the matrix entries. Because we do not have a possibility to obtain the hypothetical real-semantics results, we compare the errors against the errors that would be computed with interval arithmetic. Note that none of these benchmarks is known to be particularly unstable for floating-point errors, so that we cannot observe some particularly bad behavior. We can see though that except for the second and third (short) run of the Nbody benchmark our AffineFloat gives consistently better bounds on the roundoff errors. The numbers for the SOR benchmark also suggest that the library scales better on longer computations.

Benchmark	rel. error AF	rel. error IA
SOR 5 iter.	2.327e-14	4.869e-14
SOR 10 iter	4.618e-13	3.214e-12
SOR 15 iter	8.854e-12	2.100e-10
SOR 20 iter	1.677e-10	1.377e-8
NBody, initial energy	5.9e-15	6.40e-15
Nbody, 1s, h=0.01	1.58e-13	1.28e-13
Nbody, 1s, h=0.0156	1.04e-13	8.32e-14
Nbody, 5s, h=0.01	2.44e-10	7.17e-10
Nbody, 5s, h=0.015625	1.42e-10	4.67e-10
Spectral norm 2 iter	1.8764e-15	7.1303e-15
Spectral norm 5 iter	4.9296e-15	2.4824e-14
Spectral norm 10 iter	7.5071e-15	5.6216e-14
Spectral norm 15 iter	1.0114e-14	8.8058e-14
Spectral norm 20 iter	1.7083e-14	1.1905e-13

Table 2. Comparison of the relative errors computed by AffineFloat and interval arithmetic.

	actual error	AffineFloat	IA
with pivoting			
LU 5x5	2.22e-16	1.04e-13	6.69e-13
LU 10x10	8.88e-16	7.75e-12	2.13e-10
LU 15x15	4.44e-16	6.10e-10	1.92e-8
no pivoting			
LU 5x5	1.78e-15	2.50e-11	1.24e-9
LU 10x 10	5.77e-15	2.38e-10	4.89e-6
LU 15x15	7.15e-13	-	-
FFT 512	1.11e-15	9.73e-13	6.43e-12
FFT 256	6.66e-16	3.03e-13	2.38e-12

Table 3. Maximum absolute errors computed by Double, AffineFloat and interval versions for the LU factorization and FFT benchmarks. The matrices were random matrices with entries between 0 and 1.

Table 3 shows measurements of precision with `AffineFloat` for those benchmarks. These results can actually be checked knowing the properties of this particular

[1] All benchmarks are available from `http://lara.epfl.ch/w/smartfloat` .

application. In our example application, an LU factorization of the matrix A is used to compute the solution to the system of linear equations $Ax = b$, with b a vector. From the solution x we can compute Ax and the actual roundoff errors committed as $Ax - b$. Note, that because $Ax - b$ is a vector, we only consider the maximum roundoff error from the entries. This error is then compared to the maximum roundoff error attached to x when the solution is computed with AffineFloats and intervals. For the FFT benchmark, we can compute the transform and its inverse and compare it to the original input. We again compare the maximum roundoff errors from the matrix entries. We applied the LU factorization to random matrices with and without pivoting.[2] We compared the error bounds against interval arithmetic and the actual error. (Note that the computation of the error for the LU transform involves some multiplication, hence these error bounds are not very precise themselves.) Our AffineForm can show the pivoting approach to be clearly more accurate and provides consistently better bounds than interval arithmetic. For LU factorization of size 15x15 both affine and interval arithmetic compute bounds that are too large to be useful.

In general, the type of computation has a strong influence on how fast the over-approximation of error bounds grows. Affine as well as interval arithmetic compute larger roundoff bounds for longer computations, because they accumulate worst-case errors at each step. We have shown that AffineFloats limit this over-approximation better and provide smaller bounds than interval arithmetic. In addition, our library detects the rare cases when a computation is precise and then includes no new error.

4.2 SmartFloat Precision

In contrast to AffineFloat, one SmartFloat variable represents a whole interval of values and computes the worst-case roundoff error over the entire interval.

Doppler example. For an evaluation of the SmartFloat type, consider the Doppler frequency shift. The following equation computes the frequency change due to the Doppler effect

$$z = \frac{dv}{du} = \frac{-(331.4 + 0.6T)v}{(331.4 + 0.6T + u)^2}$$

by decomposing it into the following sub-calculations: $q_1 = 331.4 + 0.6T$, $q_2 = q_1 v$, $q_3 = q_1 + u$, $q_4 = q_3^2$, $z = q_2/q_4$. The parameters used are $-30°C \leq T \leq 50°C$, $20Hz \leq v \leq 20000Hz$ and $-100m/s \leq u \leq 100m/s$. We compare the results to [35], who chose an SMT-based approach, and summarize them in Table 4. We can compare not only the range bounds but also the roundoff errors to the minimum number of bits required as determined in [35]. Our estimates show precisely which calculations require more precision, namely the ones with the largest roundoff errors.

[2] Pivoting attempts to select larger elements in the matrix during factorization to avoid numerical instability.

B-splines example. Now consider the B-spline basic functions commonly used in image processing [33]

$$\begin{aligned} B_0(u) &= (1-u)^3/6 \\ B_1(u) &= (3u^3 - 6u^2 + 4)/6 \\ B_2(u) &= (-3u^3 + 3u^2 + 3u + 1)/6 \\ B_3(u) &= u^3/6 \end{aligned}$$

with $u \in [0, 1]$. Zhang et al. [62] use these functions to test their new way to approximate non-linear multiplication in affine arithmetic. In line with the use case for testing (subsection 5.3), we use SmartFloat to estimate the ranges and roundoffs of these functions on the given input interval. For this purpose, we divide the input interval twice and four times respectively. Observe the results in Table 5, where we compare the computed bounds against the ones from [62], and intervals (with the same dividing procedure). [62] improves the multiplication algorithm for affine arithmetic, but can only provide the final ranges, whereas our SmartFloat is able to bound the roundoff errors of the results as well. Their strategy is sophisticated, but also computationally expensive. However, Table 5 shows that with a suitable strategy, SmartFloat can indeed produce very useful and precise results while at the same time being efficient.

4.3 Performance

Our technique aims to provide much more information than ordinary floating point execution while using essentially the same concrete execution. We therefore do not expect the performance to be comparable to that of an individual double precision computation on dedicated floating-point units. Nonetheless, our technique is effective for unit testing and for exploring smaller program fragments one at a time.

The runtimes of AffineFloat and SmartFloat are summarized in Table 6. The SmartFloat uses the extra higher-order information as described in subsection 7.4, which accounts for the larger runtimes. Note that the computations are long running (indicated by the operation count of a double precision computation). This is more than any of the tools we know can handle, yet the runtimes remain acceptable. Similarly, total memory consumption in these benchmarks was not measurably large.

4.4 Compacting of Noise Terms

Existing affine arithmetic descriptions generally give no guidelines on how to choose bounds on the number of linear terms used in the approximation and how to compact them once this number is exceeded. Our algorithm for compacting noise symbols is described in subsection 6.10. In this paragraph we briefly describe its effect on performance. We ran experiments with AffineFloat on all our benchmarks and concluded that in general the runtime grows and precision increases according to the maximum number of noise symbols allowed. The results are summarized in Table 7 and Figure 2.

	AA [35]	SMT [35]	bits [35]	SmartFloat (outward-rounded)	abs. roundoff
q1	[313, 362]	[313, 362]	6	[313.3999,361.4000]	8.6908e-14
q2	[-473252, 7228000]	[6267, 7228000]	23	[6267.9999,7228000.0000]	3.3431e-09
q3	[213, 462]	[213, 462]	8	[213.3999,461.4000]	1.4924e-13
q4	[25363, 212890]	[45539, 212890]	18	[44387.5599,212889.9600]	1.6135e-10
z	[-80, 229]	[0, 138]	8	[-13.3398,162.7365]	6.8184e-13

Table 4. Doppler example from [35]. Our values were rounded outwards to 4 digits. The third column indicates the minimum number of bits needed to compute the result.

	true ranges	ranges [62]	Intervals 2 div.	Intervals 4 div.	SmartFloat 2 div.	SmartFloat 4 div.	errors for 4 div.
B_0	$[0, \frac{1}{6}]$	[-0.05, 0.17]	[0, 0.1667]	[0, 0.1667]	[-0.0079, 0.1667]	$[-3.25 \cdot 10^{-4}, 0.1667]$	1.43e-16
B_1	$[\frac{1}{6}, \frac{2}{3}]$	[-0.05, 0.98]	[-0.2709, 0.9167]	[-0.1223, 0.6745]	[0.0885, 0.8073]	[0.1442, 0.6999]	6.98e-16
B_2	$[\frac{1}{6}, \frac{2}{3}]$	[-0.02, 0.89]	[0.0417, 1.1042]	[0.1588, 0.9558]	[0.1510, 0.8230]	[0.1647, 0.7097]	7.2e-16
B_3	$[-\frac{1}{6}, 0]$	[-0.17, 0.05]	[-0.1667, 0]	[-0.1667, 0]	[-0.1667, 0.0261]	[-0.1667, 0.0033]	1.3e-16
time		358s	< 1s	< 1s	< 1s	< 1s	

Table 5. B-splines with SmartFloat compared against intervals and [62]. The errors given are absolute errors.

	double	interval	AffineFloat	SmartFloat	+	-	*	/, $\sqrt{}$	trig
Nbody (100 steps)	2.1	21	779	33756	9530	3000	14542	2006	0
Spectral norm (10 iter.)	0.6	31	198	778	4020	0	4020	4002	0
Whetstone (10 repeats)	1.2	2	59	680	1470	510	600	110	0
Fbench	0.2	1.3	10	1082		115	120	89	94
Scimark - FFT (512x512)	1.2	18	1220	39987	13806	15814	19438	37	36
Scimark - SOR (100x100)	0.8	25	698	127168	8416	1	19209	0	0
Scimark - LU (50x50)	2.6	30	2419	4914	0	45425	44100	99	0
Spring sim. (10000 steps)	0.2	46	1283	4086	20002	20003	30007	10002	0

Table 6. Running times (in ms) of our set of benchmarks, compared against the running time in pure doubles. The numbers on the right give the numerical operation count of each benchmark, i.e. the number of operations a double precision computation performs. Tests were run on a Linux machine with 2.66GHz and 227MB of heap memory available.

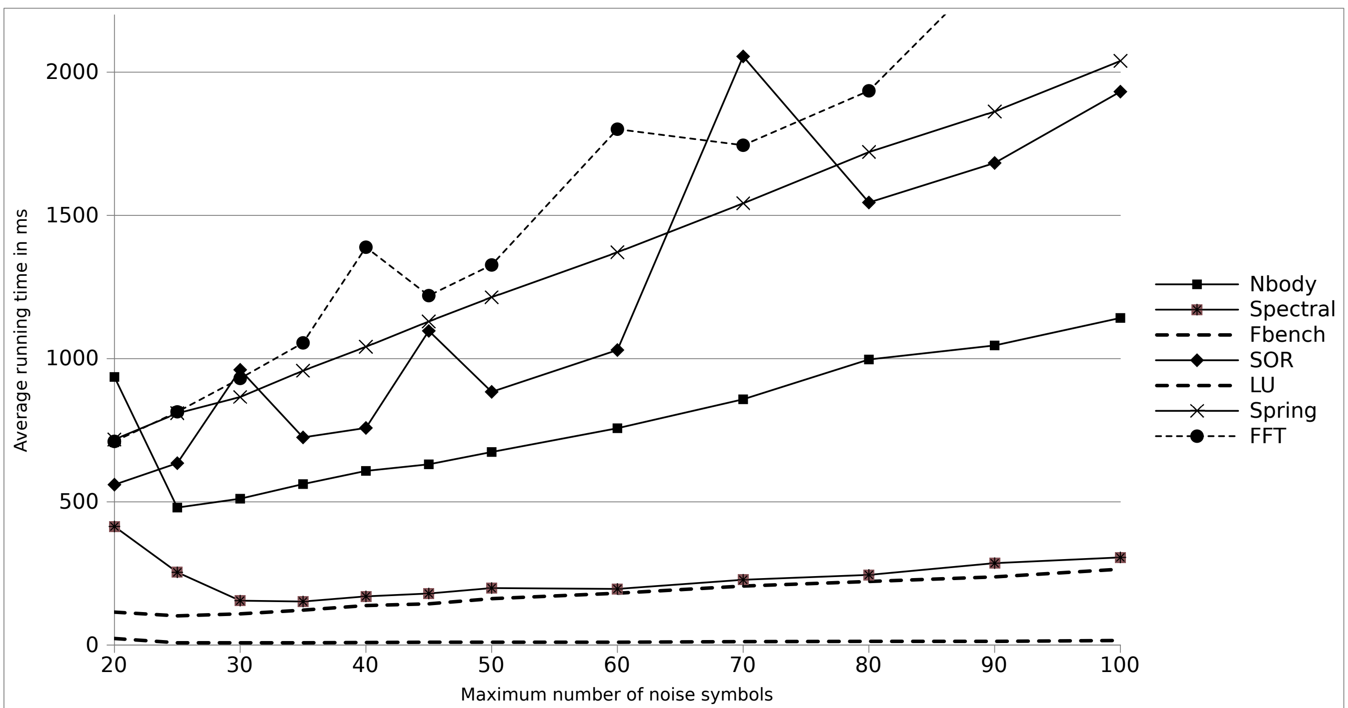


Figure 2. The effect of the number of noise symbols on the average running time (in ms).

	20	40	60	80	100
Nbody	1.60e-13	1.58e-13	1.57e-13	1.55e-13	1.56e-13
Spectral	1.94e-14	9.25e-15	6.63e-15	8.73e-15	5.94e-15
Fbench	3.09e-13	1.28e-13	9.48e-14	1.20e-13	5.22e-14
FFT	1.17e-16	1.63e-16	1.17e-16	1.57e-16	2.04e-16
SOR	1.17e-16	1.17e-16	1.32e-16	1.32e-16	1.32e-16
LU	1.16e-16	1.18e-16	1.54e-16	1.14e-16	2.09e-16
Spring	1.81e-09	1.77e-09	1.73e-09	1.69e-09	1.64e-09

Table 7. The effect of the number of noise symbols on accuracy.

The peaks in the runtime graph for very small thresholds can be explained by the library spending too much time compacting than doing actual computations. The irregular peaks for the SOR and FFT benchmarks illustrate that sometimes the precise characteristics of the calculation problem can influence running times. Every operation's runtime is roughly proportional to the number of noise symbols, and different compacting thresholds change the number of noise symbols at each operation. It can then happen that with a larger threshold the compacting happens just before a critical operation, e.g. one that is executed often, but for a lower threshold the compacting happens earlier and by the time that operation is reached the number of symbols has grown again and thus the overall running time is longer. It is thus necessary that the user has some control over the compacting procedure. Note that the precision is not significantly affected in most of our experiments, hence the library has a default limit of around 40 noise symbols as a good compromise between performance and accuracy.[3] The developer may change this value for particular calculations.

Summary. We have demonstrated in this section that our library provides an improvement over currently used techniques. In the examples we examined, it provided comparable results to those from a SMT solver, at a fraction of the time (see Table 4). Furthermore, it often provides dramatically better results compared to intervals. An interval subdivision benefits our approach just as it benefits interval analysis. Note, however, that systematic interval subdivision is not feasible when the inputs are, e.g., matrices, as in Table 3. Namely, the total number of subdivided points increases exponentially with the number of elements in the matrix, so the total cost quickly dwarfs the added computational cost of affine arithmetic. In such cases, our affine arithmetic approach is essential to obtain additional orders of magnitude more precision over interval arithmetic.

5. Further Applications

Our `SmartFloat` type can be used to soundly estimate ranges of floating-point numbers, roundoff errors over an entire range of floating-point numbers, or both. So far, we have only discussed immediate applications of these types. In this section we suggest further possible use cases, which also point to a possible integration of our tool into larger frameworks.

5.1 User-Defined Error Terms

The ranges of a computation are determined chiefly by input intervals and roundoff errors incurred along a computation path. However, errors can come from different sources as well. For instance, during the integration of an ordinary differential equation the numerical algorithm accumulates method errors, i.e. errors due to the discrete nature of the integration algorithm. One, albeit simple, example where this is the case is the simulation of a (undamped and unforced) spring in Figure 3. For simplicity, we use Euler's method. Although this method is known to be too inaccurate for many applications, it provides a good application showcase for our library. The `comparison failed!` line is explained in subsection 5.2, for now note the method `addError` in line 14. In this example, we compute a coarse approximation of the method error by computing the maximum error over the whole execution. What happens behind the scenes is that our library adds an additional error to the affine form representing x, i.e. it adds a new noise term in addition to the errors already computed.

Now consider the output of the simulation using our library. Notice that using step sizes 0.1 and 0.01, time t cannot be computed precisely, whereas using $t = 0.125$, which is representable in binary, the result is exact. Now consider x. We can see that choosing smaller step sizes, the enclosure of the result becomes smaller and thus more accurate, as expected. But note also, that the use of a smaller step size also increases the overall roundoff errors. This is also to be expected, because we have to execute more computations.

Note that this precise analysis of roundoff errors is only possible with the separation of roundoff errors from other uncertainties. Our SmartFloat type can thus be used in a more general framework that guarantees soundness with respect to a floating-point implementation but that also includes other sources of errors.

5.2 Robustness

Our library can also show code to be robust in certain cases. A computation is robust if small changes in the input cause only small changes in the output. There are two ways in which a program, starting from some given input, can change the output.

Change in input causes the control flow to change. In this case, the program takes a different branch or executes a loop more or less many times, so that it actually executes different code, causing the output to differ. In each case, this involves a comparison in a guard. Our library handles these comparisons in a special way, by keeping a global boolean flag which is set if a comparison fails. What we mean by a comparison failing is that the information computed by

[3] Actually, the number used by default is 42.

```
   def springSimulation(h: SmartFloat) = {
2    val k: SmartFloat = 1.0
     val m: SmartFloat = 1.0
4    val xmax: SmartFloat = 5.0
     var x: SmartFloat = xmax  //curr. horiz. position
6    var vx: SmartFloat = 0.0  //curr. velocity
     var t: SmartFloat = 0.0  //curr. 'time'
8
     var methodError = k*m*xmax * (h*h)/2.0
10
     while(t < 1.0) {
12     val x_next = x + h * vx
       val vx_next = vx − h * k/m * x
14     x = x_next.addError(methodError)
       vx = vx_next
16     t = t + h
     }
18   println("t: " + t + ", x: " + x)
   }
```

Spring simulation for h = 0.1:

```
comparison failed!
t: [1.099,1.101] (8.55e-16)
x: [2.174, 2.651] (7.4158e-15)
```

Spring simulation for h = 0.125:

```
t: [1.0,1.0] (0.00e+0)
x: [2.618, 3.177] (4.04e-15)
```

Spring simulation for h = 0.01:

```
comparison failed!
t: [0.999, 1.001] (5.57e-14)
x: [2.699, 2.706] (6.52e-13)
```

Figure 3. Simulation of a spring with Euler's method. The numbers in parentheses are the maximum absolute roundoff errors committed. We have rounded the output outwards for readability reasons.

affine forms is not sufficient to determine whether this value is smaller or bigger than another one. Hence, for the comparison $x < y$, the difference $x - y$ includes zero.

The user may, when notified by such a situation, choose to refine the input intervals until no such warning occurs. In addition, the user may choose that the library emits a warning (`comparison failed!`) as seen in Figure 3. The same comparison procedure is also used for the methods `abs`, `max`, `min` so the library sets the flag, if it detects a robustness problem, for these functions as well.

Computation is numerically unstable. In this case, the control flow may stay the same, but the input range of the variables gets amplified, yielding a much larger output interval. The programmer can also detect this case with our library, as he only needs to compare the input to the output widths of the intervals. Note that our library only gives estimates on the upper bounds on roundoff errors, but not on the lower bounds. That is, our library makes inevitably over-approximations, so the computed output interval may be larger than the true interval. However, the user can, if such a case is suspected, rerun the computation using `AffineFloats`, which in general give tighter bounds.

Illustration of control-flow robustness check. As an example for the first case of robustness problem, consider again the code in Figure 3. Our library set the global comparison flag in two of the three runs. Because there is only one comparison in this code, it is clear that the (possible) problem occurred in line 12. And in fact, we can see that in the first case for $h = 0.1$, the loop was actually executed once too many, thus also giving a wrong result for the value of x. In the second case $h = 0.125$, because the computation of time is exact, the flag is correctly not set.

5.3 Testing Numerical Code

We can take SmartFloat's ability to detect when a program takes different paths within an input interval even further. Suppose we have a piece of code and, for simplicity, one input variable, for which we assume that the input is within some finite range $[a, b]$. To generate a set of input intervals that exercise all possible paths through the program, we propose the following procedure: Start with the entire interval $[a, b]$ and run the program with our SmartFloat type. If the library does not indicate any robustness problem, we are done and can read off the maximum roundoff error incurred. If it detects a possible problem, split the interval and rerun the program on each of the new input intervals. Repeat until no problem occurs, or until an error in the program is found. In addition to test inputs, SmartFloats also provide guaranteed bounds on the errors for each of the paths. The splitting can, in addition to control flow changes, also be triggered on too large roundoff error bounds. In this way, the testing procedure can be refined to obtain error bounds for each run to a desired precision.

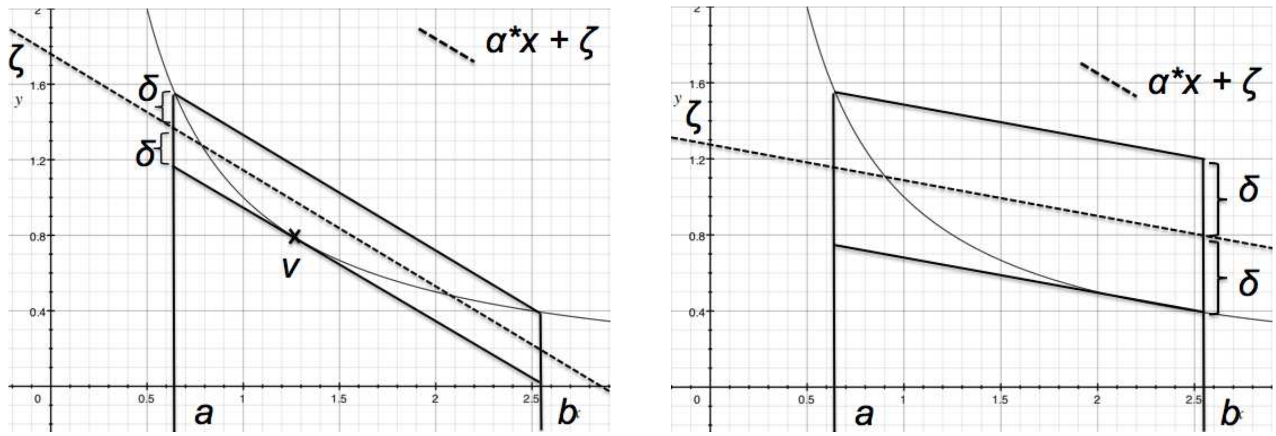
6. AffineFloat Design and Implementation

We will now discuss our contributions in developing an affine arithmetic library suitable for evaluating floating-point computations. The main challenge are non-linear approximations, and this basically for two reasons.

- The precision is unsatisfactory if implemented in a simple way.
- The roundoff error estimation is not sound if using a standard approximation method.

6.1 Different Interpretations of Computations

When using a range-based method like interval or affine arithmetic, it is possible to have different interpretations of what such a range denotes. In this paper we consider the following three different interpretations of affine arithmetic.



(a) Chebyshev approximation

(b) Min-range approximation

Figure 4. Linear pproximations of the inverse function.

INTERPRETATION 1 (Original Affine Arithmetic). *In the original affine arithmetic, an affine form $\hat{x}$ represents a range of real values, that is $[\hat{x}]$ denotes an interval $[a, b]$ for $a, b \in \mathbb{R}$.*

This is also the interpretation from [17].

INTERPRETATION 2 (Exact Affine Arithmetic). *In exact affine arithmetic $\bar{x}$ represents* **one** *floating-point value and its deviation from an ideal real value. That is, if a real valued computation computed $x \in \mathbb{R}$ as the result, then for the corresponding computation in floating-points that $x \in [\bar{x}]$.*

The difference to Interpretation 1 is that the central value x_0 has to be equal to the actually computed double value at all times. We will discuss the reason for this in subsection 6.3.

INTERPRETATION 3 (Floating-point Affine Arithmetic). *In floating-point affine arithmetic $\tilde{x}$ represents a range of floating-point values, that is $[\tilde{x}]$ denotes an interval $[a, b]$ for $a, b \in \mathbb{F}$.*

Interpretation 3 corresponds to our SmartFloat type and Interpretation 2 to AffineFloat, the details of which are discussed in this section.

Usually implementation issues are of minor interest, however in the case of floating-point computations they are an important aspect: our tool itself uses floating-point values to compute roundoff errors, so that we are faced with the very same problems in our own implementation that we are trying to quantify.

6.2 Nonlinear Operations

Affine operations are computed as

$$\alpha\hat{x}+\beta\hat{y}+\zeta = (\alpha x_0+\beta y_0+\zeta)+\sum_{i=1}^{n}(\alpha x_i+\beta y_i)\epsilon_i+\iota\epsilon_{n+1}$$

For nonlinear operations like multiplication, inverse, or square root, this formula is not applicable so that the operations have to be approximated. Multiplication is derived from expanding and multiplying two affine forms:

$$\hat{x}\hat{y} = x_0 y_0 + \sum_{i=1}^{n}(x_0 y_i + y_0 x_i)\epsilon_i + (\eta+\iota)\epsilon_{n+1}$$

where ι contains the internal errors and η an over-approximation of the nonlinear contribution. To compute the latter, several possibilities exist of varying degree of accuracy. In the case of tracking a single floating-point value, the simplest way $\eta = rad(\hat{x}) \cdot rad(\hat{y})$ is sufficient as the radii will be in general several orders of magnitude smaller than the central values. For larger ranges, the nonlinear part of multiplication unfortunately becomes a notable problem and is discussed in subsection 7.4. Division $\hat{x}/\hat{y}$ is computed as $\hat{x} \cdot (1/\hat{y})$ so that it remains only to define unary nonlinear function approximations.

For the approximation of unary functions, the problem is the following: given $f(\hat{x})$, find $\alpha, \zeta, \delta \in \mathbb{F}$ such that

$$[f(\hat{x})] \subset [\alpha\hat{x} + \zeta \pm \delta]$$

α and ζ are determined by a linear approximation of the function f and δ represents all (roundoff and approximation) errors committed, thus yielding a rigorous bound.

[17] suggests two approximations for computing $\alpha, \zeta,$ and δ: a Chebyshev (min-max) or a min-range approximation. Figure 4 illustrates these two on the example of the inverse function $f(\hat{x}) = \hat{x}^{-1}$. For both approximations, the algorithm first computes the interval represented by $\hat{x} = [a, b]$ and then works with its endpoints a and b. In both cases we want to compute a bounding box around the result, by computing the slope (α) of the dashed line, its intersection with the y-axis (ζ) and the maximum deviation from this middle line (δ). This can be done in the following two ways:

Min-range Compute the slope α at one of the endpoints a or b. Compute the intersections of the lines with this slope that go through either a or b. Fix ζ to be the average of the two. Compute δ as the maximum deviation, which occurs by construction at either a or b.

Chebyshev Compute the slope α of the line through both a and b. This gives one bounding side of the wanted 'box' (parallelepiped). To find the opposite side, compute the point where the curve takes on the same slope again. Again, compute ζ as the average of the intersections of the two lines and δ as the maximum deviation at either the middle point v, a or b.

In general, the Chebyshev approximation computes smaller parallelepipeds, especially if the slope is significantly different at a and b. However, it also needs the additional computation of the middle point. Especially for transcendental functions like $acos, asin$, etc., this can involve quite complex computations which are all committing internal roundoff errors. On big intervals, like the one considered in [17] and [18] these are (probably) not very significant. However, when keeping track of roundoff errors, our library deals with intervals on the order of machine epsilon. From the experience with several versions of transcendental function approximations we concluded that min-range is the better choice. Chebyshev approximations kept returning unexpected and wrong results. As discussed in [18], the Chebyshev approximation would be the more accurate one in long running computations, however we simply found it to be too numerically unstable for our purpose. To our knowledge, this problem has not been acknowledged before.

Obviously, any linear approximation is only valid when the input range does not cross any inflection or extreme points of the function. Should this occur, our library resorts to computing the result in interval arithmetic and converting it back into an affine form.

Error estimation for nonlinear library functions like log, exp, cos, etc. requires specialized rounding, because the returned results are correct to 1 ulp (unit in the last place) only [1] (for the standard Scala math library), and hence are less accurate than elementary arithmetic operations, which are correct to within 1/2 ulp. The directed rounding procedure is thus adapted in this case to produce larger error bounds, so that it is possible to analyze code with the usual Scala mathematical library functions without modifications.

6.3 Guaranteeing Soundness of Error Estimates

What we have described so far applies to the original affine arithmetic as well as our AffineFloat. However, our goal is to quantify roundoff errors, and original affine arithmetic has not been developed to quantify them, only to compute sound bounds (i.e. intervals) on output values, interpreted over ranges of real numbers. It turns out that if affine arithmetic is modified appropriately, it can be used for the quantification of roundoff errors as in Interpretation 2. For this, we assume that the central value x_0 is exactly the floating-point value computed with double precision and the noise symbols x_i represent the deviation due to roundoff errors and approximation inaccuracies from non-affine operations. A straight-forward re-interpretation of affine arithmetic from Section 6 is not sound as the following observation shows.

OBSERVATION 2. *The algorithm for approximating non-affine operations using the min-range approximation as defined in subsection 6.2 is unsound under the interpretation of Interpretation 2.*

Namely, the interpretation of affine arithmetic as in Interpretation 2 relies on the assumption that the central value x_0 is equal to the floating-point value of the original computation. This is important, as the roundoff for affine operations is computed according to Equation 1, i.e. by multiplication of the *new* central value by some δ. If the central value does not equal the actual floating-point value, the computed roundoff will be that of a different result. Affine operations maintain this invariant. However, non-affine operations defined by computing α, ζ and δ such that the new affine form is $\hat{z} = \alpha \cdot \hat{x} + \zeta + \delta\epsilon_{n+1}$ do not necessarily enforce that the actual double value computed in the operation is equal to the new central value $z_0 = \alpha \cdot x_0 + \zeta$. That is, in general (and in most cases), the new z_0 will be slightly shifted. In general the shift is not large, however soundness cannot be guaranteed any more.

Fortunately, an easy solution exists and is illustrated in Figure 5. For non-linear operations, the new central value is computed as $z_0 = \alpha \cdot x_0 + \zeta$ and we want $f(x_0) = \alpha \cdot x_0 + \zeta$. Hence, our library computes ζ as

$$\zeta = f(x_0) - \alpha \cdot x_0$$

The min-range approximation computes for an input range $[a, b]$ an enclosing parallepiped of a function as $\alpha \cdot x + \zeta \pm \delta$ that is guaranteed to contain the image of the nonlinear function from this interval as computed in floating-point precision. Suppose that $\zeta = f(x_0) - \alpha \cdot x_0$, with α computed at one of the endpoints of the interval. Because we compute the deviation δ with outwards rounding at both endpoints and keep the maximum, we soundly over-approximate the function f in floating-point semantics. Clearly, this approach only works for input ranges where the function in question is monotonic. By the Java API [1], the implemented library functions are guaranteed to be semi-monotic, i.e. whenever the real function is non-decreasing, so is the floating-point one.

It is clear from Figure 5 that our modified approximation computes a bigger parallelepiped than the original min-range approximation. However, in this case, the intervals are very small to begin with, so the over-approximations do not have a big effect on the precision of our library.

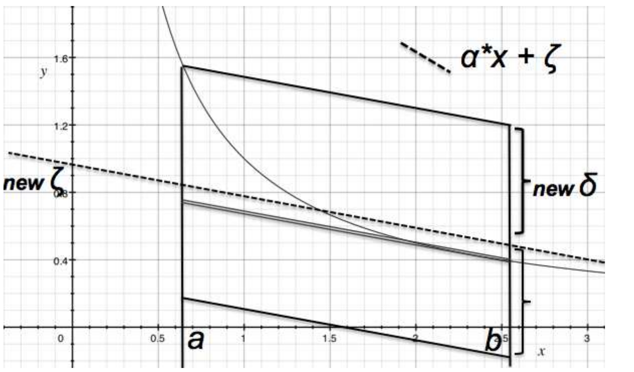


Figure 5. Modified min-range approximation of the inverse function.

6.4 Double-Double Precision for Noise Terms

It turns out that even when choosing the min-range approximation with input ranges with small widths (order 10^{-10} and smaller), computing the result of a nonlinear function in interval arithmetic gives better results. The computation of α and ζ in our approximation cannot be changed for soundness reasons, but it is possible to limit the size of δ. In order to avoid arbitrary precision libraries for performance reasons, our library uses double-double precision (denoted as $\mathbb{DD}$) as a suitable compromise. Each value is represented by two standard double precision values. Algorithms have been developed and implemented [16, 51] that allow the computation of standard arithmetic operations with only ordinary floating-point operations, making the performance trade-off bearable. In this way, our library can compute range reductions for the sine and cosine functions accurately enough. It can also avoid using intervals to bound δ, an approach we found not to be sufficiently effective for our purpose.

One condition for these algorithms to work is that the operations are made in *exactly* the order as given and without optimizations or fused-multiply instructions. We have enforced this for our code (which runs on the JVM) by using the `strictfp` modifier for the calculations.

Our library uses double-double precision types for the noise symbols and computations involving them. Keeping the noise symbols in extended precision and thus reducing also the internal roundoff errors, we have found that the accuracy of our library increased sufficiently for most nonlinear function approximations. To ensure soundness of double-double computations, we use the outward rounding mode.

6.5 Precise Handling of Constants

A single value, say 0.03127, is represented in a real valued interval semantics as the point interval $[0.03127, 0.03127]$ or in affine arithmetic as $\hat{x} = 0.03127$, i.e. without noise terms. This no longer holds for floating-point values that cannot be represented exactly in the underlying binary representation. Our library tests each value for whether it can be represented or not and adds noise terms only when necessary. In the case of the above example, it creates the following affine form: $0.03125+(\epsilon_M \cdot 0.03125)\epsilon_n$. This limits the over-approximations committed and provides more precise analyses when possible. For an error estimate according to Interpretation 2, our runtime library has the exact values available and can thus generally compute tighter bounds compared to a static analysis-based approach.

6.6 Computing Roundoff Errors

The JVM does not provide access to the different rounding modes of the floating-point unit, so that the expressions that need directed rounding are implemented as native C methods. It turns out that this approach does not incur a big performance penalty, but provides the needed precision, which cannot be achieved by simulated directed rounding. The native C code has to be compiled for each architecture separately, but because no specialized functionality is needed this is a straightforward process and does not affect the portability of our library. Using directed rounding also enables the library to determine when a calculation is exact so that no unnecessary noise symbols are added.

6.7 Soft Policy to Avoid Too Many False Warnings

Our solution follows the ‘soft’ policy advocated in [17], whereby slight domain breaches for functions that work only on restricted domains are attributed to the inaccuracy of our over-approximations and are ignored. For example, with a ‘hard’ policy computing the square root of $[-1, 4]$ results in a run-time error, as the square root function is not defined on all of the input interval. It is possible however that the true interval (in a real semantics) is $[0, 4]$ and the domain problem is just a result of a previous over-approximation. In order to not interrupt computations unnecessarily with false alarms, a ‘soft’ policy computation will give the result $[0, 2]$. Note, that our library nonetheless generates warnings in these cases if the user chooses so, so that the policy only affects the tool’s ability to continue a computation in ambiguous cases, but not its rigorousness.

6.8 Correctness

The correctness of each step of the interval or affine arithmetic computation implies the correctness of our overall approach: for each operation in interval or affine arithmetic the library computes a rigorous over-approximation, and thus the overall result is an over-approximation. This means, that for all computations, the resulting interval is guaranteed to contain the result that would have been computed on an ideal real-semantics machine.

The use of assertions certifying that certain invariants always hold support the correctness of our implementation. Example invariants for `AffineFloat` include the statement that the computed double precision value has to be exactly the same as the central value of the affine form, a prerequisite for our roundoff analysis.

In addition, we have tested our library extensively on several benchmarks (see Section 4) and our implementation of nonlinear functions against the results from 30 digit precision results from Mathematica.

We are able to avoid several pitfalls related to floating-point numbers [9, 46] by writing our library in Scala and not for example in C, as the JVM is not as permissive to optimizations that may alter the actual execution of code.

6.9 Using a Library on an Example

To illustrate the use of AffineFloat we present its use on a classic example, the quadratic formula in Figure 6. This example illustrates the effect of roundoff errors because it produces less accurate results (two orders of magnitude in this particular case), when one root is much smaller. Our library shows the result of rewriting this code following the method in [24]. Our library confirms that both roots are now computed with approximately the same accuracy:

```
classic r1 = -18.655036847834893 (5.7133e-16),
        r2 = -0.0178874602678082 (1.4081e-13)
smarter r1 = -18.655036847834893 (5.7133e-16),
        r2 = -0.0178874602678077 (7.7584e-16)
```

The values in parentheses give the relative errors. Note that the code looks nearly the same as if it used the standard Double type, thanks to the techniques we used to integrate it into Scala (see section 8).

```
var a = AffineFloat(2.999)
var b = AffineFloat(56.0001)
var c = AffineFloat(1.00074)
val discr = b * b − a * c * 4.0

//classical way
var r2 = (−b + sqrt(discr))/(a * 2.0)
var r1 = (−b − sqrt(discr))/(a * 2.0)
println("classic r1 = " + r1 + ", r2 = " + r2 )

//smarter way
val (rk1: AffineFloat, rk2: AffineFloat) =
if(b*b − a*c > 10.0) {
  if(b > 0.0)
    ((−b − sqrt(discr))/(a * 2.0),
        c * 2.0 /(−b − sqrt(discr)))
  else if(b < 0.0)
    (c * 2.0 /(−b + sqrt(discr)),
    (−b + sqrt(discr))/(a * 2.0))
  else
    ((−b − sqrt(discr))/(a * 2.0),
     (−b + sqrt(discr))/(a * 2.0))
}
else {
  ((−b − sqrt(discr))/(a * 2.0),
   (−b + sqrt(discr))/(a * 2.0))
}
println("smarter r1 = " + rk1 + ", r2 = " + rk2)
```

Figure 6. Quadratic formula computed in two different ways.

6.10 Managing Noise Symbols in Long Computations

The runtime performance of our library depends on the number of noise terms in each affine form, because each operation must access each term at least once. Hence, an appropriate compacting strategy of noise symbols becomes crucial for performance. Compacting too little means that our approach becomes unfeasible, whereas compacting too much means the loss of too much correlation information.

Compacting algorithm. The goal of compaction is to take as input a list of noise terms and output a new list with fewer terms, while preserving the soundness of the roundoff error approximation and, ideally, keeping the most important correlation information. Our library performs compaction by adding up the absolute values of the smallest terms and introducing them as a fresh noise symbol along with the remaining terms. We propose the following strategy to compute the fresh noise term:

- Compact all error terms smaller than 10^{-33}. These errors are smaller than the smallest double value and are thus internal errors. Our library can manipulate such small values because it uses double-double precision internally (section 6.4).
- Compute the average ($avrg$) and the standard deviation ($stdDev$) of the rest of the noise terms. Compact all terms smaller than $avrg \cdot a + stdDev \cdot b$ and keep the rest. The factors a and b are user-controllable positive parameters, and can be chosen separately for each computation. (The result is sound regardless of the particular values.)
- In some cases the above steps are still not enough to ensure that the number of symbols is below the threshhold. This occurs, for example, if nearly all errors have the same magnitude. If our library detects this case, it repeats the above procedure one more time on the newly computed noise terms. In our examples, at most two iterations were sufficient. In pathological cases in which this does not suffice, the library compacts all noise symbols into a single one.

7. SmartFloat Design and Implementation

The implementation described in Section 6 provides a way to estimate roundoff errors for one single computation. It provides reasonably tight bounds for the most common mathematical operations and is fast enough for middle sized computations, hence it can be used to provide some intuition about the behavior of a calculation. It does not provide, however, any guarantee as to how large the errors would be if one chose (even slightly) different input values or constants. In this section we investigate the following two aspects:

1. computation of a rigorous range of floating-point values (according to Interpretation 3);

2. computation of sound roundoff error estimates for this range.

Unfortunately, a straightforward reinterpretation of neither the original affine arithmetic, nor the modified version for AffineFloat gives a sound range arithmetic for floating-point numbers.

OBSERVATION 3. *The roundoff computation of affine operations as defined in Equation 3 is unsound under the interpretation of Interpretation 3.*

Namely, when tracking a *range* of floating-point numbers and computing the roundoff errors of each computation, we need to consider the roundoff errors for *all* values in the range, not only the central value as is the case in Equation 3. In addition, the non-linear approximation algorithm does not explicitly compute the roundoff errors, they are implicitly included in the computed δ. If we now have input values given by (possibly wide) ranges, the computed δ will be so large that no roundoff estimate from them is meaningful.

Our library provides a new type, SmartFloat as a solution for this problem. A SmartFloat can be constructed from a double value or a double value with an uncertainty, providing thus a range of inputs. A SmartFloat variable $\tilde{x}$, then keeps the initial double value and the following tuple

$$\tilde{x} = (x_0, \sum x_i\epsilon_i + \sum x_i u_i, \sum r_i\rho_i), \qquad x_i, r_i \in \mathbb{DD} \tag{4}$$

where $x_0 \in \mathbb{F}$ is the central value as before and $x_i\epsilon_i$ and $x_i u_i$ are the noise terms characterizing the range. We now mark those that come from user-defined uncertainties by special noise symbols u_i, which we call *uncertainties*. We keep these separately, so that for instance during the noise term compacting, these are preserved. $r_i\rho_i$ are the *error terms* quantifying the roundoff errors committed. The sum $\sum |r_i|$ gives a sound estimate on the current maximum committed roundoff error for all values within the range. We now need to define the computation and propagation of roundoff errors; the noise terms are handled as before.

7.1 Computation of Roundoffs

To compute the roundoff error of an operation, our library first computes the new range. It uses either Equation 3 for affine operations or the min-range approximation for non-linear ones and then computes the maximum roundoff from the resulting range. Following the definition of roundoff from Equation 1 the maximum roundoff is the maximum absolute value in the range multiplied by ϵ_M. For the other operations, correct to within 1 ulp, we adjust the factor to $2 \cdot \epsilon_M$.

7.2 Propagation of Roundoffs

The already committed errors $\tilde{e_x} = \sum r_i\rho_i$ in some affine form $\tilde{x}$ have to be propagated correctly for each operation.

Affine. The propagation is given straightforwardly by Equation 3. That is, if the operation involves the computation $\alpha\tilde{x} + \beta\tilde{y} + \zeta$, the errors are transformed as $\alpha\tilde{e_x} + \beta\tilde{e_y} + (\iota + \kappa)\rho_{n+1}$, where ι corresponds to the internal errors committed and κ to the new roundoff error.

Multiplication. The linear part is computed as usual by multiplication by x_0 and y_0. The non-linear part in multiplication poses the difficulty that it involves cross-terms between the noise and error terms. We derive the new propagation, by appending the error terms to the noise term sum and then compute the multiplication. Then, removing those terms not involving any error terms, we get the new η_e:

$$\begin{aligned}\eta_e = \quad & rad(\tilde{x}) \cdot rad(e_y) + rad(\tilde{y}) \cdot rad(e_x) + \\ & rad(e_x) \cdot rad(e_y)\end{aligned}$$

Note that this produces an over-approximation, because some of the errors from the error terms are also included already in the noise terms.

Non-affine. Because the nonlinear function approximations compute α, ζ and δ, the propagation of errors reduces to an affine propagation, with one exception. The factor used to propagate the roundoff errors must be, instead of α, the maximum slope of the function over the given input range to ensure soundness. Because this value does not necessarily equal α, we need to compute that factor separately.

7.3 Additional Errors

Additional errors, e.g. method errors from a numerical integration, can be added to the affine form in the following way. Given $\tilde{x} = (x_0, \sum x_i\epsilon_i, \sum r_i\rho_i)$ and the error to be added $\tilde{y} = (y_0, \sum y_i\epsilon_i, \sum s_i\rho_i)$, the resulting affine form is given by

$$\begin{aligned}\tilde{z} = (x_0, \sum x_i\epsilon_i + (|y_0| + rad(\sum y_i\epsilon_i))\epsilon_{n+1}, \\ \sum r_i\rho_i, +(rad(\sum s_i\rho_i)\rho_{m+1})\end{aligned}$$

That is, the maximum magnitude of the error is added as a new noise term, and the maximum magnitude of the roundoff committed when computing this error is added as a new error term.

7.4 Treatment of Range Explosion due to Multiplication

In Section 6 the naive computation of the non-linear part for multiplication was sufficiently accurate due to the relatively small radii of the involved affine forms. This is in general no longer the case if we consider arbitrary ranges of floating-point numbers. To illustrate this problem, consider $\tilde{x} = 3 + 2\epsilon_1$ and $\tilde{y} = 4 + 3\epsilon_2$ Both values are clearly positive, hence their product should be positive as well. Now, $\tilde{z} = \tilde{x} \cdot \tilde{y} = 12 + 8\epsilon_1 + 9\epsilon_2 + 6\epsilon_3$ which gives as resulting interval $[\tilde{z}] = [-11, 35]$. This result is unacceptable, if this value is subsequently used in for instance division.

[58, 62] have suggested some approaches, however they either change the underlying structure by using matrices in-

stead of affine forms or are simply not scalable enough. Because we do not want to change the underlying data structure, we chose a different solution for this problem. The problem is, that by computing $\eta = (\sum_{i=1}^{n} |x_i|) \cdot (\sum_{i=1}^{n} |y_i|)$ and appending it with a fresh noise symbol, correlation information is lost. Affine forms do not provide the possibility to keep quadratic terms. We can keep, however, 'source' information with each noise term. For example, if a noise term is computed as $x_1 x_2 \epsilon_1 \epsilon_2$, it results in a new fresh noise symbol $x_3 \epsilon_3 [1, 2]$, where the indices in brackets denote the information that is additionally stored. Similarly, if the product involves two noise terms that already contain such information, it is combined. Currently, our library supports up to 8 indices, however this value can be extended as needed - at a performance cost of course. Most operations work exactly as before; this information is only used when the interval of an affine form is computed and is essentially an optimization problem. One option is to use a brute force approach and to substitute all possible combinations of $-1, 1, 0$ for all ϵ_i. Because an affine form represents a convex range of values, the maximum and minimum value of this range has to necessarily be at ϵ_i having one of these values. Clearly, this approach is not very efficient, but for up to 11 noise terms is still feasible. We use our compacting algorithm to reduce the number of noise symbols before this optimization is run to make this approach efficient enough in practice.

The example from Section 4.2 demonstrates the impact of this simple solution on the Doppler shift problem. [35] choose an SMT-based approach, precisely for the reason that affine arithmetic produces too large over-approximations. We compare the results from the approaches in Table 4. Note that our library obtains its results in under half a second.

Clearly, the library could compute better bounds if a better optimization method is used, for instance by using a dedicated solver.

7.5 Nested Affine Arithmetic

An even smarter version of SmartFloat can provide information on how the output roundoff error depends on the input error, thus providing additional insight about the computation. This is possible, provided that roundoff errors are computed as *functions* of the initial uncertainties, and not just absolute values. Note that if we restrict ourselves to linear functions, we can use affine forms for the new roundoffs. That is, for the affine form representing the roundoff errors $\tilde{e_x} = \sum r_i \rho_i$, the library now keeps each r_i as an affine form (i.e. affine form of affine forms). It keeps only the linear terms of uncertainties (and not the noise terms from the computation itself) and compacts all other terms for performance reasons. Finally, the computation of the actual roundoff errors becomes an optimization problem similar to the one for multiplication. Our library currently reports the assignment that minimizes and maximizes the roundoff errors. Note that due to over-approximations this reported assignment may not necessarily be the one giving the real smallest or largest roundoff. For this reason, the user may want to examine the, say, three smallest or largest assignments respectively.

Although this feature is so far only experimental, we believe that it can become very useful. We will demonstrate this by returning to the triangle example from Table 1. With the modified SmartFloat, we can now run the following code:

```
val area = triangleArea(9.0,
  SmartFloat(4.7, 0.19), SmartFloat(4.7, 0.19))
area.analyzeRoundoff
```

The output is

```
analyzing the roundoffs...
maximum relative error: 4.728781774296841E-13
maximizing assignment: 10 -> -1.0, 7 -> -1.0
minimum relative error: 8.920060990068312E-14
minimum assignment: 10 -> 1.0, 7 -> 1.0
```

To explain this output, the numbers 10 and 7 denote the indices of the uncertainties that were assigned to b and c respectively, that is, those are the indices of their uncertainty noise symbols. The final analysis reveals that for the assignment of -1.0 to both noise symbols, the roundoff is maximized. Looking back at the definition of the values we can see that the assignment of -1.0 corresponds to the input value of 4.51 for both b and c. This corresponds exactly to the known property that the relative roundoff errors are largest for the thinnest triangles. Similarly, the assignment of 1.0 corresponds to the least thin triangles, as expected.

8. Integration into a Programming Language

This section explains how we integrated AffineFloat and SmartFloat data types into Scala in a seamless way. Our decision to implement a runtime library was influenced by several factors. Firstly, a runtime library is especially useful in the case of floating-point numbers, because the knowledge of exact values enables us to provide a much tighter analysis that cannot be achieved in the general case in static analysis. Also, with our tight integration it is possible to use any Scala construct, thus not restricting the user to some subset that an analyzer can handle.

8.1 Our Deployment as a Scala Library

Our library provides wrapper types AffineFloat and SmartFloat that track the computation errors. These types are meant to replace the Double types in the user-selected parts of a program. All that is needed to put our library into action are two import statements at the beginning of a source file,

```
import smartfloats.SmartFloat
import smartfloats.SmartFloat._
```

and the replacement of Double types by one of the AffineFloat or SmartFloat types. Any remaining conflicts are signaled by the compiler's strong typechecker. The new data types handle definitions of variables and the standard arithmetic operations, as well as many scala.math library functions, including the most useful:

- log, expr, pow, cos, sin, tan acos, asin, atan
- abs, max, min
- constants Pi (π) and E (e)

The library also supports special values NaN and $\pm\infty$ with the same behavior as the original code.

To accomplish such an integration, we needed to address the following issues.

Operator overloading. Developers should still be able to use the usual operators +, −, ∗, / without having to rewrite them as functions, e.g x.add(y). Fortunately, Scala allows x m y as syntax for the statement x.m(y) and (nearly) arbitrary symbols as method names [49] , including +, −, ∗, /.

Equals. Comparisons should be symmetric, i.e., the following should hold

```
val x: SmartFloat = 1.0
val y: Double = 1.0
assert(x == y && y == x)
```

The == will delegate to the equals method, if one of the operands is not a primitive type. However, this does not result in a symmetric comparison, because Double, or any other built-in numeric type, cannot compare itself correctly to a SmartFloat. Fortunately, Scala also provides the trait (similar to a Java [25] interface) ScalaNumber which has a special semantics in comparisons with ==. If y is of type ScalaNumber, then both x == y and y == x delegate to y.equals(x) and thus the comparison is symmetric [52].

Mixed arithmetic. Developers should be able to freely combine our SmartFloats with Scala's built-in primitive types, as in the following example

```
val x: SmartFloat = 1.0
val y = 1.0 + x
if (5.0 < x) {...}
```

This is made possible with Scala's implicit conversions, strong type inference and companion objects [49]. In addition to the class SmartFloat, the library defines the (singleton) object SmartFloat, which contains an implicit conversion similar to

```
implicit def double2SmartFloat(d : Double):
  SmartFloat = new SmartFloat(d)
```

As soon as the Scala compiler encounters an expression that does not type-check, but a suitable conversion is present, the compiler inserts an automatic conversion from the Double type in this case to a SmartFloat. Therefore, implicit conversions allow a SmartFloat to show a very similar behavior to the one exhibited by primitive types and their automatic conversions.

Library functions. Having written code that utilizes the standard mathematical library functions, developers should be able to reuse their code without modification. Our library defines these functions with the same signature (with SmartFloat instead of Double) in the companion SmartFloat object and thus it is possible to write code such as

```
val x: SmartFloat = 0.5
val y = sin(x) * Pi
```

Concise code. For ease of use and general acceptance it is desirable not having to declare new variables always with the **new** keyword, but to simply write SmartFloat(1.0). This is possible as this expression is syntactic sugar for the special apply method which is also placed in the companion object.

8.2 Applicability to Other Languages

The techniques described in this paper can be ported to other languages, such as C/C++, or to languages specifically targeted for instance for GPU's or parallel architectures, provided the semantics of floating-point numbers is well specified. In fact, language virtualization in Scala [53] can be used to ultimately generate code for such alternative platforms instead of the JVM.

9. Related Work

Estimating Roundoff Errors in Fluctuat. Closest to our work is the Fluctuat static analyzer [26] which analyzes numerical code with operations $+, -, *, /$ for roundoff errors and their sources based on abstract interpretation and affine arithmetic. A comparison solely based on the results reported [26] is difficult, because the results are obtained by using different user-supplied settings for different examples (for example, using interval subdivisions, or a larger number of bits for internal computations). If we do the subdivision into tens of sub-intervals in our system (using a simple for-loop in the user's code), we find that the accuracy on many of our examples increases further by a decimal order of magnitude. A direct experimental comparison with the Fluctuat implementation was not possible due to the commercial nature of the tool, but there are several important conceptual differences.

We do not view our approach as abstract interpretation, because our library is closer to the actual program execution, and this makes it more appropriate for (and easier to use for) testing. We never perform joins and never approximate objects and other data structures using alias analysis as in Fluctuat. We do not expect to reach a fixpoint, because we maintain the exact floating-point value of the numbers. Because we only replace the floating-point number data type and keep the structure of the data exact, we generate the same trace and terminate in the same number of steps as the original program. This lack of approximation gives us more precision. Furthermore, our tool supports mathematical functions beyond the standard four arithmetic operators.

No other work we know of documents the actual approximations used. As described in Section 6 in detail, this task turns out to be non-trivial. We also introduced the noise symbol packing technique (subsection 6.10), which allows us to support longer-running computations with many floating-point operations, while having stable time and memory overhead. [26] does not describe how Fluctuat deals with the growing numbers of noise symbols, nor does it indicate whether the computations performed were simply not of the size that would require it, as in our benchmarks.

We note that in addition to SmartFloat, we also introduced AffineFloat, which tracks the error of a specific computation, with concrete input points, instead of an error over an interval. AffineFloat is even closer to concrete execution: it gives more precise results for one execution (with fewer guarantees about other executions). We have further enhanced AffineFloat precision by using directed rounding to recognize the cases where the error is actually zero, instead using the worst case of machine epsilon relative error for each operation. Because of this, we avoid introducing unnecessary error terms. As a result, entire series of basic arithmetic computations can be found to have zero error. We have not seen this precision enhancement mentioned elsewhere, or performed by other tools.

Abstract interpretation Further work in abstract interpretation includes the Astrée analyzer [14] and APRON [13, 31]. These systems provide abstract domains that work correctly in floating-point semantics, but they do not attempt to quantify roundoff errors (an attempt to treat intervals themselves as roundoff errors gives too pessimistic estimates to be useful). In general, tools based on abstract interpretation (including Fluctuat) attempt to rigorously establish global ranges for variables, regardless of what these ranges correspond to. In contrast, we are interested in the correspondence between the computed value and the ideal mathematical result.

Another difference is that, instead of focusing only on embedded domain, we are interested also in scientific computations, which often have transcendental functions (so we needed to develop approximation rules for them), and non-trivial data structures. Data structures contain many floating-point numbers whose values cannot be simply approximated by joint values as in many static analysis approaches, otherwise too much precision would be lost.

A recent approach [30] statically detects loss of precision in floating-point computations using bounded model checking with SMT solvers, but uses interval arithmetic for scalability reasons. [21] uses affine arithmetic to track roundoff errors using a C library; this work is specific to the signal processing domain. Further approaches to quantify roundoff errors in floating-point computations are summarized in [44], of which we believe affine arithmetic to be the most useful one. This also includes stochastic estimations of the error, which have been implemented in the CADNA library [32]. However, the stochastic approach does not provide rigorous bounds, because, for example, in loops, roundoff errors are not uniformly distributed.

Affine Arithmetic. Existing implementations of affine arithmetic include [2] and [3]. Researchers [62] have proposed a solution to the over-approximation problem of multiplication by repeatedly refining the approximation until a desired precision is obtained. While the method provides a possible solution to the over-approximation problem, it is also computationally expensive. None of these implementations can be used to quantify roundoff errors, only to compute ranges in the way described by the original affine arithmetic [17]. As a result, the problems we describe in this paper do not arise, and the existing systems cannot be used as such for our purpose. Other range-based methods are surveyed in [45] in the context of plotting curves. We have decided to use affine arithmetic, because it seems to us to be a good compromise between complexity and functionality. A library based on Chebyshev and Taylor series is presented in [20], however it does not provide correlation information as affine arithmetic does, so its use is directed more towards non-linear solvers. Affine arithmetic is used in several application domains to deal with uncertainties, for example, in signal processing [27]. Our library is developed for general-purpose calculations and integrated into a programming language to provide information about floating-points for any application domain.

Robustness Analysis. Our library can detect the cases when the program would continue to take the same path in the event of small changes to the input, thanks to the use of the global sticky flag set upon the unresolved comparisons. Therefore, we believe that our library can be useful for understanding program robustness and continuity properties, for which sophisticated techniques have been investigated [12, 43].

Finite-Precision Arithmetic. [38] uses affine arithmetic for bit-width optimization and provides an overview of related approaches. [56] uses affine arithmetic with a special model for floating-points to evaluate the difference between a reduced precision implementation and normal float implementation, but uses probabilistic bounding to tackle over-approximations. Furthermore, it only allows addition and multiplication. [35] employs a range refinement method based on SMT solvers and affine arithmetic, which is one way to deal with the division-by-zero problem due to over-approximations. The authors use a timeout to deal with the case when such computation becomes too expensive. Another approach uses automatic differentiation [23] to symbolically compute the sensitivity of the outputs to the inputs. While this approach could be used for roundoff error analysis as well, the symbolic expressions need to be evaluated, and thus need to ultimately rely on methods such as interval or affine arithmetic.

Theorem Proving Approaches. Researchers have used theorem proving to verify floating-point programs [6, 8, 28, 47, 55]. These approaches provide high assurance and guarantee deep properties. Their cost is that they rely on user-provided specifications and often require lengthy user interactions. [40] extend previous work using affine arithmetic by considering the problem of reducing precision for performance reasons. However, the resulting system still requires interactive effort. [11] presents a decision procedure for checking satisfiability of a floating-point formula by encoding into SAT. Even this approach requires the use of approximations, because of the complexity of the resulting formulas. A symbolic execution technique that supports floating-point values was developed [10], but it does not quantify roundoff errors. There is a number of general-purpose approaches for reasoning about formulas in non-linear arithmetic, including the MetiTarski system [5]. Our work can be used as a first step in verification and debugging of numerical algorithms, by providing the correspondence between the approximate and real-valued semantics.

10. Conclusions

We have presented a library that introduces numerical types, SmartFloat and AffineFloat, into Scala. Like the standard Double type, our data type supports a comprehensive set of operators. It subsumes Double in that it does compute the same floating point value. In addition, it also computes a roundoff error—an estimate of the difference between this floating-point value and the value of the computation in an ideal real-number semantics. SmartFloat can compute the roundoff error not only for a given value, but also for the values from a given interval, with the interval being possibly much larger than the roundoff error.

It can be notoriously difficult to reason about computations with floating-point numbers. Running a computation with a few sample values can give us some understanding for the computation at hand. The newly developed data types allow developers to estimate the error behavior on entire classes of inputs using a single run. We have found the performance and the precision of these data types to be appropriate for unit-testing of numerical computations. We are therefore confident that our implementation is already very helpful for reasoning about numerical code, and can be employed for building future validation techniques.

References

[1] DocWeb - Java SE 6 - java.lang.Math. `http://doc.java.sun.com/DocWeb/#r/JavaSE6/java.lang.Math/columnMain`.

[2] J. Stolfi's general-purpose C libraries. `http://www.ic.unicamp.br/~stolfi/EXPORT/software/c/Index.html#libaa`, 2005.

[3] aaflib - An Affine Arithmetic C++ Library. `http://aaflib.sourceforge.net/`, 2010.

[4] The Computer Language Benchmarks Game. `http://shootout.alioth.debian.org/`, Jan 2011.

[5] B. Akbarpour and L. C. Paulson. MetiTarski: An Automatic Theorem Prover for Real-Valued Special Functions. *J. Autom. Reason.*, 44(3), 2010.

[6] A. Ayad and C. Marché. Multi-Prover Verification of Floating-Point Programs. In *IJCAR*, 2010.

[7] T. Ball, E. Bounimova, R. Kumar, and V. Levin. Slam2: Static driver verification with under 4% false alarms. In *FMCAD*, pages 35–42, 2010.

[8] S. Boldo, J.-C. Filliâtre, and G. Melquiond. Combining Coq and Gappa for Certifying Floating-Point Programs. In *CICM*, 2009.

[9] S. Boldo and T. M. T. Nguyen. Hardware-independent proofs of numerical programs. In *Proceedings of the Second NASA Formal Methods Symposium*, 2010.

[10] B. Botella, A. Gotlieb, and C. Michel. Symbolic execution of floating-point computations. *Softw. Test. Verif. Reliab.*, 2006.

[11] A. Brillout, D. Kroening, and T. Wahl. Mixed Abstractions for Floating-Point Arithmetic. In *FMCAD*, 2009.

[12] S. Chaudhuri, S. Gulwani, and R. Lublinerman. Continuity analysis of programs. In *POPL*, 2010.

[13] L. Chen, A. Miné, J. Wang, and P. Cousot. A Sound Floating-Point Polyhedra Abstract Domain. In *APLAS*, 2008.

[14] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE Analyser. In *ESOP*, 2005.

[15] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-sensitive program verification in polynomial time. 2002.

[16] M. Davis. DoubleDouble.java. `http://tsusiatsoftware.net/dd/main.html`.

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

[18] L. H. de Figueiredo and J. Stolfi. Affine Arithmetic: Concepts and Applications. *Numerical Algorithms*, 2004.

[19] Isil Dillig, Thomas Dillig, and Alex Aiken. Sound, complete and scalable path-sensitive analysis. In *PLDI*, 2008.

[20] A. G. Ershov and T. P. Kashevarova. Interval Mathematical Library Based on Chebyshev and Taylor Series Expansion. *Reliable Computing*, 11, 2005.

[21] C.F. Fang, Tsuhan C., and R.A. Rutenbar. Floating-point error analysis based on affine arithmetic. In *ICASSP*, 2003.

[22] Stephen Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanual Geay. Effective typestate verification in the presence of aliasing. In *ISSTA'06*, 2006.

[23] A.A. Gaffar, O. Mencer, and W. Luk. Unifying bit-width optimisation for fixed-point and floating-point designs. In *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, pages 79 – 88, april 2004.

[24] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1), 1991.

[25] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java(TM) Language Specification, The 3rd Edition*. Addison-Wesley, 2005.

[26] E. Goubault and S. Putot. Static Analysis of Finite Precision Computations. In *VMCAI*, 2011.

[27] Ch. Grimm, W. Heupke, and K. Waldschmidt. Refinement of Mixed-Signal Systems with Affine Arithmetic. In *DATE*, 2004.

[28] J. Harrison. Formal Verification at Intel. In *LICS*, 2003.

[29] L. Hatton and A. Roberts. How Accurate is Scientific Software? *IEEE Trans. Softw. Eng.*, 20, 1994.

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

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

[32] F. Jézéquel and J.-M. Chesneaux. CADNA: a library for estimating round-off error propagation. *Computer Physics Communications*, 178(12), 2008.

[33] J. Jiang, W. Luk, and D. Rueckert. FPGA-Based Computation of Free-Form Deformations. In *Field - Programmable Logic and Applications*. 2003.

[34] W. Kahan. Miscalculating Area and Angles of a Needle-like Triangle. Technical report, University of California Berkeley, 2000.

[35] A.B. Kinsman and N. Nicolici. Finite Precision bit-width allocation using SAT-Modulo Theory. In *DATE*, 2009.

[36] Etienne Kneuss, Philippe Suter, and Viktor Kuncak. Runtime instrumentation for precise flow-sensitive type analysis. In *International Conference on Runtime Verification*, 2010.

[37] Viktor Kuncak, Patrick Lam, Karen Zee, and Martin Rinard. Modular pluggable analyses for data structure consistency. *IEEE Transactions on Software Engineering*, 32(12), December 2006.

[38] D.-U. Lee, A. A. Gaffar, R. C. C. Cheung, O. Mencer, W. Luk, and G. A. Constantinides. Accuracy-Guaranteed Bit-Width Optimization. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 25(10), 2006.

[39] X. Leroy. Verified squared: does critical software deserve verified tools? In *POPL*, 2011.

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

[41] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., 2nd edition, 1999.

[42] R. Longbottom. Whetstone Benchmark Java Version. `http://www.roylongbottom.org.uk/online/whetjava.html`, 1997.

[43] R. Majumdar and I. Saha. Symbolic Robustness Analysis. In *IEEE Real-Time Systems Symposium*, 2009.

[44] M. Martel. An overview of semantics for the validation of numerical programs. In *VMCAI*, 2005.

[45] R. Martin, H. Shou, I. Voiculescu, A. Bowyer, and G. Wang. Comparison of interval methods for plotting algebraic curves. *Comput. Aided Geom. Des.*, 19(7), 2002.

[46] D. Monniaux. The pitfalls of verifying floating-point computations. *ACM Trans. Program. Lang. Syst.*, 30(3), 2008.

[47] J. S. Moore, T. W. Lynch, and M. Kaufmann. A Mechanically Checked Proof of the AMD5 K86 Floating Point Division Program. *IEEE Trans. Computers*, 47(9), 1998.

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

[49] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: a comprehensive step-by-step guide*. Artima Press, 2008.

[50] R. Pozo and B. R. Miller. Java SciMark 2.0. http://math.nist.gov/scimark2/about.html, 2004.

[51] D. M. Priest. Algorithms for Arbitrary Precision Floating Point Arithmetic. In *Proceedings of the 10th Symposium on Computer Arithmetic*, 1991.

[52] Scala Programming Language Blog. '==' and equals. `http://scala-programming-language.1934581.n4.nabble.com/and-equals-td2261488.html`, June 2010.

[53] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *GPCE*, 2010.

[54] P. M. Rondon, M. Kawaguci, and R. Jhala. Liquid types. In *PLDI*, pages 159–169, 2008.

[55] D. M. Russinoff. A Mechanically Checked Proof of Correctness of the AMD K5 Floating Point Square Root Microcode. *Formal Methods in System Design*, 1999.

[56] R. A. Rutenbar, C. F. Fang, M. Püschel, and T. Chen. Toward efficient static analysis of finite-precision effects in DSP applications via affine arithmetic modeling. In *DAC*, 2003.

[57] T. R. Scavo and J. B. Thoo. On the Geometry of Halley's Method. *The American Mathematical Monthly*, 102(5), 1995.

[58] H. Shou, R.R. Martin, I. Voiculescu, A. Bowyer, and G. Wang. Affine Arithmetic in Matrix Form for Polynomial Evaluation and Algebraic Curve Drawing. *Progress in Natural Science*, 12, 2002.

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

[60] J. Walker. fbench - Trigonometry Intense Floating Point Benchmark. `http://www.fourmilab.ch/fbench/fbench.html`, 2007.

[61] J. White. Fbench.java. `http://code.google.com/p/geo-reminder/source/browse/trunk/benchmark-android/src/com/benchmark/suite/Fbench.java?r=108`, 2005.

[62] L. Zhang, Y. Zhang, and W. Zhou. Tradeoff between Approximation Accuracy and Complexity for Range Analysis using Affine Arithmetic. *Journal of Signal Processing Systems*, 61, 2010.