On the Design and Implementation of SmartFloat and AffineFloat

Eva Darulová 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. However, rigorous reasoning about the resulting programs remains difficult, because of a large gap between the finite floating point representation and the infinite-precision real-number semantics that serves as the mental model for the developers. 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 a range of 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.

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 [51] 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 [23, 40, 47]. 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 [24], 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 operations (precisely specified by the standard) compose into an *overall* error of a complex computation relative to a hypothetical ideal value. Such roundoff errors can accumulate to the point where the computed value is not a precise enough approximation of the real value. Currently, the developers have no reliable automated method to determine the error incurred by such approximation. Whereas for other important properties we currently have type systems and static analysis techniques that can establish them, for errors we do not even have available practical methods to accurately estimate errors in individual computations.

While there is general agreement that we need to expand verification to be practical for numerical computation [32], we do not even know today how to *test* numerical codes. Indeed, the very first problem we face is that, for a given execution, we do not know if the execution is correct in the sense of being close to the expected meaning of operations in real analysis. This makes formal or informal reasoning about programs with floating-points very difficult. As a result, we have little confidence in the correctness of an increasingly important set of applications.

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 [42] programming language. These data types act as a drop-in replacement for the standard floating point data types, such as Double, and offer a support for a comprehensive range of operations with a greater precision than in any previously documented solution.

When faced with these problems, many approaches use interval arithmetic (IA) as a solution. However, in many cases intervals give too pessimistic estimates. The problem is easy to demonstrate, and its essence can be seen already on a very simple example: if x is an interval $[0, a]$, then the expression $x-x$ is approximated by $[-a, a]$, although it is, in fact, equal to zero. 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 floating point value;
- uncertainty in the actual values of floating point variables given that the initial values can be anywhere from the initial interval.

As a result, the computed estimates quickly become imprecise. To avoid these problems, we first examine affine arithmetic, which was first introduced in [15] and can more precisely track the correlations 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, 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 we use in our system are also available in the publicly available source code. Furthermore, we introduce an approach that allows the library to track errors over a range of values. We are therefore able to give answers to the following questions:

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

By providing a freely available library that addresses these questions, we hope to provide users with a tool that is easy to use and that helps to gain understanding about the floating-point properties of code as well as a tool that provides sound guarantees on the floating-point roundoff errors committed. Our system can be integrated as part of larger verification and testing systems.

Contributions. This paper makes 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 computed is identical to the one computed with the standard data 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 supporting a number of non-linear and transcendental functions reliably. Finally, the implementation contains a technique to soundly bound the number of affine error terms, ensuring predictable performance and precision without sacrificing much precision.
- We develop and implement a SmartFloat data type that generalizes AffineFloat and can estimate upper bounds on roundoff errors over an *entire range of input values* and 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 linear approximation of error estimates for input ranges including zero, providing an improved description of relative errors for such ranges.
- We evaluate the precision and performance of our implementation on a number of benchmarks from physics and the numerical analysis field.The results show that our library produces, possibly after initial interval subdivision, precise estimates that otherwise require theorem proving techniques. It also shows that the library scales to long-running computation.

Our implementation is publicly available at:

`http://lara.epfl.ch/w/smartfloat`

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), which gives the high-level idea of the approach (but is not sufficient to obtain our results). We then characterize the precision and performance of our implementation in Section 4. We show further applications enabled by our system in Section 5, and describe integration into Scala in Section 6. We then describe the design and implementation of AffineFloat (Section 7) and SmartFloat (Section 8). We finally present related work and conclusions.

2. Examples

Cube root. Intervals have the unfortunate property of ignoring correlations between variables and thus often over-approximate the errors by far too much to be useful. As an illustration, consider the following code fragment that uses Halley's method [49] to compute the cube root of 10, starting from an initial value of xn = 1.6:

```
for (i ← 1 until 5)
```

```
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 with the classic textbook formula and Kahan's improved version.

```
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 Mathematica, and the result returned by interval arithmetic:

Double	2.1544346900318834
Math	2.154434690031883721...
Interval	$[2.1544346900317617, 2.154434690032006]$
Affine	$2.1544346900318834 \pm 1.34 \cdot 10^{-15}$

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 several 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 and decide to e.g. adopt a more expensive computation method instead.

Area of a triangle. As another example, consider the code in Figure 1. triangleTextbook computes the area of a triangle with the well-known textbook formula. On the other hand, triangleKahan uses an improved version by Kahan [29]. Running both versions with our SmartFloat type and with intervals, we get the results listed in Figure 2. Although interval arithmetic does not over-approximate the range by much

	area	rel. roundoff
Interval Arithmetic		
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]	-
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]	-
SmartFloat		
triangleTextbook		
a = 9.0, b, c = SmartFloat([4.71, 4.89])	[6.25, 8.62]	1.10e-14
a = 9.0, b, c = SmartFloat([4.61, 4.79])	[4.50, 7.39]	1.97e-14
triangleKahan		
a = 9.0, b, c = SmartFloat([4.71, 4.89])	[6.25, 8.62]	3.11e-15
a = 9.0, b, c = SmartFloat([4.61, 4.79])	[4.49, 7.39]	5.26e-15

Figure 2. Area and relative roundoffs computed on the code from Figure 1 with SmartFloat and intervals.

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 roundoffs indicate that for thin triangles relative roundoff errors grow, which is indeed what happens. This illustrates that our library allows not only formal reasoning (by establishing correspondence to real-valued semantics), but also 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 the input range and the error bound will fail. Our technique therefore 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 through the interval is relatively large.

3. 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 (64 bit) precision, $\mathbb{R}$ the real numbers.
- $\mathbb{IF}, \mathbb{IR}$ denote intervals of floating-point and real numbers respectively.
- $[i]$ denotes the interval represented by an expression a.
- $\downarrow x \downarrow$ and $\uparrow x \uparrow$ denotes the result of some expression x rounded towards $+\infty$ or $-\infty$ respectively.

3.1 IEEE Floating-point Arithmetic

Throughout this paper we assume that floating-point arithmetic conforms to the IEEE 754 floating-point standard [51]. All recent hardware conforms to it and it is also generally respected in some subset by main programming languages. The JVM (Java Virtual Machine), on which Scala's code is run, supports single and double precision floating-point values according to the standard as well as the rounding-to-nearest rounding mode [34]. 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, it follows for binary operations that the result in floating-point arithmetic satisfies

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

where ϵ_M is the machine epsilon and determines the upper bound on the relative error of a floating-point computation. This model provides a basis for our roundoff error estimates.

Thanks to dedicated floating-point units in most hardware, floating-point computations are fast, so our library is aimed at double precision floating-point values only (i.e. $\epsilon_M = 2^{-53}$). This is also the precision of choice for most numerical algorithms. It is straight-forward to adapt the error estimation for single precision, or any other precision with a corresponding semantics.

3.2 Interval Arithmetic

One possibility to perform guaranteed computations in floating-point arithmetic is to use standard interval arithmetic proposed already in [41]. Then a bounding interval for each basic operation is computed as

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

where outwards rounding ensures soundness. The error for square root follows similarly.

We have already seen in Section 2 an illustration of 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 [15] and developed to compute ranges of values over the domain of reals, with the actual calculations done in double (finite) precision. In particular, it addresses the problem of intervals to handle correlations between variables. Affine arithmetic is one possible *range-based method* to address this task, we discuss further methods in Section 9.

Given a function $f : \mathbb{R} \to \mathbb{R}$, we wish to compute it in some discrete number representation (in this case double floating-point precision). 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 the function value with a function $g : A \to A$ that satisfies the *fundamental invariant of range analysis*:

PROPERTY 1. *If $a \in A$, $x \in \mathbb{R}$, $x \in [a]$, then $f(x) \in [g(a)]$.*

Note, that a range-based arithmetic as such does not attempt to quantify the roundoff errors itself, it only tries to compute results in a rigorous way.

A possible application of affine arithmetic as originally proposed is finding the zeroes of functions by bisecting the domain and computing a (rough) estimate of the function value over each subdomain. If the output range for one subdomain does not include zero, then that part of the domain can then be safely discarded. Affine arithmetic represents variables as affine forms

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

where x_0 denotes the *central value* and each *noise symbol* ϵ_i is a formal variable denoting a deviation from the center, and 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, however it reflects the relative dependence between values. The range represented by an affine form is computed as

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

If we compute with affine forms in (finite) double precision values, we need to take into account that some operations are not performed exactly. As suggested in [15], the roundoff errors commited during the computation can be added with a new fresh noise symbol to the final affine form. Hence, affine operations are computed as

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

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 truly rigorous bounds. 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. $\alpha * x_0 + \zeta$). These errors can be determined by the following procedure [15]:

$$\begin{aligned} z &= \ f(x_1, x_2, ...) \\ a &= \ \downarrow f(x_1, x_2, ...) \downarrow \\ b &= \ \uparrow f(x_1, x_2, ...) \uparrow \\ \iota &= \ \max(b - z, z - a) \end{aligned}$$

This suggests the use of affine arithmetic for keeping track of roundoff errors by representing 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 accumulated roundoff errors. One expects to obtain tighter bounds than with interval arithmetic, especially when a computation exhibits many correlations between variables. However, a straighforward application of affine arithmetic in the original formulation is not always sound, as we will show in section 7.

Our solution follows the 'soft' policy advocated in [15], whereby slight domain violations for functions that work only on certain domains are attributed to the inaccuracy of our over-approximations and are ignored. For example, the square root of $[-1, 4]$ results in the interval $[0, 2]$. This behavior is important in reducing false alarms due to over-approximation.

Before proceeding with the description of the technique we use in our solution, we show how library behaves in practice.

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. Overall, we found that with the help of the Scala compiler's typechecker, changing the code then to use our AffineFloat type instead of Double is a straightforward process and needs only few manual edits. We want to remark that the benchmarks have been developed with *performance* evaluation in mind and not accuracy. We have yet to find a comprehensive benchmark whose goal is to evaluate the *precision* of numerical error estimates. 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 $a_1 1 = 1$, $a_1 2 = \frac{1}{2}$, $a_2 1 = \frac{1}{3}$, $a_1 3 = \frac{1}{4}$, $a_2 2 = \frac{1}{5}$, $a_3 1 = \frac{1}{6}$, etc."

Scimark [43] is a set of Java benchmarks for scientific computations and we selected three benchmarks that best suited 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 Figure 7.

Fbench was orginally written by Walker [52] as a "Trigonometry Intense Floating Point Benchmark". We used the Java port [53] for our tests.

Whetstone [35] is a classic benchmark for performance evaluation of floating-point computations.

Spring simulation is our own code from Figure 9, however for benchmarking we removed the added method errors.

In addition, we have written an implementation of interval arithmetic in Scala that otherwise behaves exactly like AffineFloat or SmartFloat for comparison purposes.

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

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

Figure 3. Comparison of the relative errors computed by AffineFloat and interval arithmetic.

4.1 AffineFloat Precision

Figure 3 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. Since 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 unsta-

	double	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

Figure 4. Maximum absolute errors computed by Double, AffineFloat and interval versions for the LU factorisation and FFT benchmarks. The matrices were random matrices with entries between 0 and 1.

ble for floating-point errors, so that we cannot observe some particularly bad behaviour. 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.

Figure 4.1 shows measurements of precision with `AffineFloat` for those benchmarks. These results can be actually check knowing the properties of this particular application. Namely, for the LU factorization if the matrix A, we can compute Ax and compare it against the wanted result b. For the FFT benchmark, we can compute the transform and its inverse and compare it to the original input. 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.) 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.

4.2 SmartFloat Precision

Doppler example. For an evaluation of the SmartFloat type, consider again the Doppler frequency shift example from subsection 8.4. The results are summarized in Figure 5. We can not only compare the range bounds but also the roundoff errors to the minimum number of bits required as determined in [30]. 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. Consider the B-spline basic functions commonly used in image processing [28]

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

with $u \in [0, 1]$. Zhang et al. [54] use these functions to test their new and more complex way to approximate nonlinear functions in affine arithmetic. In light of the possible testing procedure we outline in subsection 5.1, we use our SmartFloat to estimate the ranges and roundoffs of these functions on the given input interval. For this, we divide the input interval twice and four times respectively and observe the results in section 4.2, where we compare the computed bounds against the ones from [54], and using the same dividing procedure with intervals. Note, that only SmartFloat is able to bound the roundoff errors of the results. We can see 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 exploring smaller program fragments at a time.

The runtimes of AffineFloat and SmartFloat are summarized in Figure 7. The SmartFloat uses the extra higher-order information as described in subsection 8.4, which accounts for the larger runtimes. Note, that the operation count is considerable, more than any of the tools we know can handle, yet the runtimes remain acceptable.

4.4 Compacting of Noise Terms

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 7.9. In this paragraph we want to briefly describe its effect on performance. We ran experiments with AffineFloat on all our benchmarks and concluded that in general, the maximum number of noise symbols allowed is proportional to the runtime and inversely proportional to the precision. The results are summarized in Figure 8. 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

	AA [30]	SMT [30]	bits [30]	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

Figure 5. Doppler example from [30]. 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 [54]	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	

Figure 6. B-splines with SmartFloat compared against intervals and [54]. The errors given are absolute errors.

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

Figure 7. Running times 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. Tests were run on a Linux machine with 2.66GHz and 227MB of heap memory available.

can influence running times. It is thus necessary that the user has some control over the compacting procedure. Note, that the precision is not significantly affected in general, so that we decided on a default limit of around 40 noise symbols as a good compromise between performance and accuracy, but the developer may change this value for particular calculations.[3]

5. Further Applications

Our `SmartFloat` type can be used to soundly estimate ranges of floating-point numbers, or to soundly estimate the roundoff errors in an entire range of floating-point numbers, or both. So far, we have only discussed immediate applications of these types. In this section we would like to suggest possible integrations of our tool into larger frameworks.

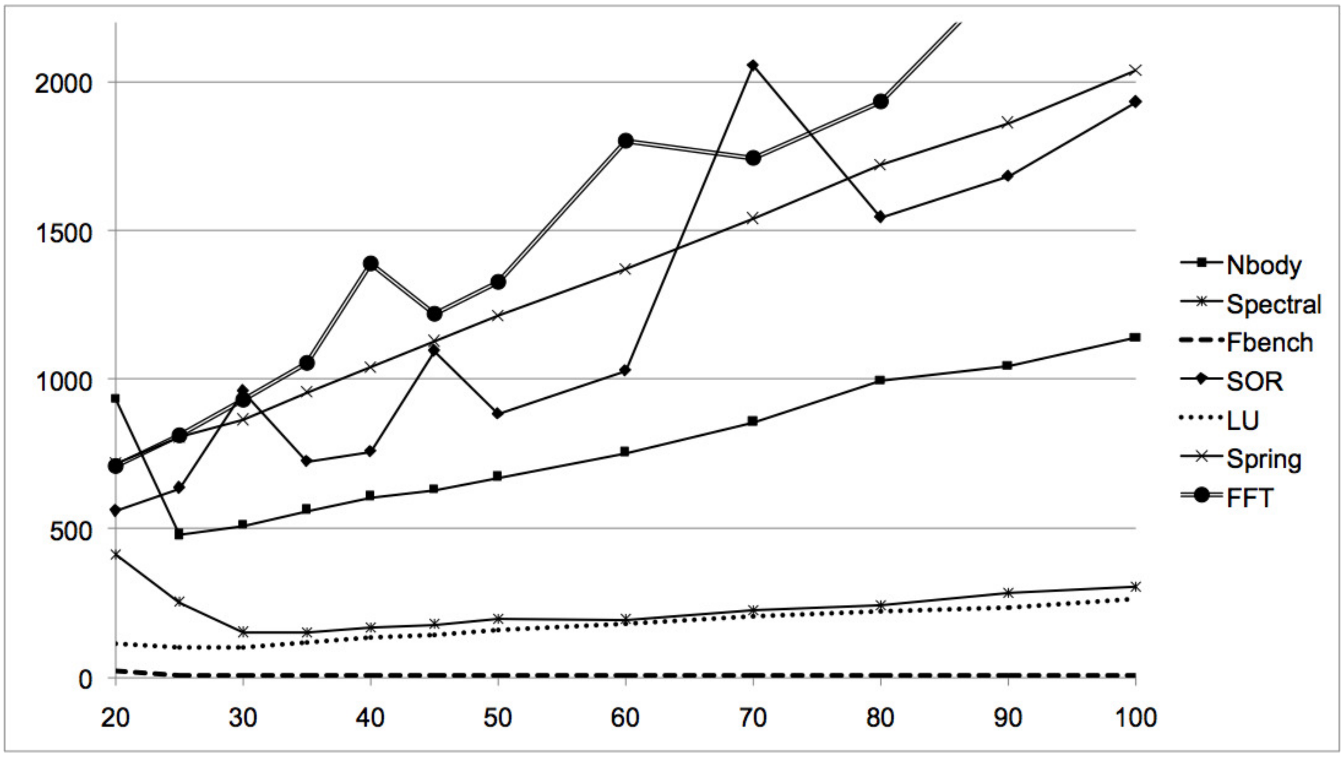
5.1 Testing Numerical Code

We can take SmartFloat's ability to detect when different paths in a program are taken 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 no robustness violation is signaled, we are done and can read off the maximum roundoff error incurred. If a possible violation is detected, split the interval and rerun the program on each of the new input intervals. Repeat until no violation occurs, or until an error in the program is found. In addition to test inputs, the SmartFloats also provide guaranteed bounds on the errors for each of the paths. The splitting can, in addition to control flow diversions, also be triggered on too large roundoff error bounds. In this way, the testing procedure can be refined for a desired precision.

5.2 User-Defined Error Terms

The ranges of a computation are determined chiefly by input intervals and roundoff errors incurred along a computation path. However, it is also possible that the source of uncertainty accumulates *during* a computation, for instance during the integration of an ordinary differential equation.

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



(a) Max. number of noise terms vs. average running time (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

(b) Max. number of noise terms vs. accuracy.

Figure 8. The effect of the number of noise symbols.

One such, albeit simple, example is the simulation of a (undamped and unforced) spring in Figure 9. For simplicity, we use Euler's method and 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 5.3, for now note the method `addError` in line 15. 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. we add a new noise term in addition to the errors already computed.

Now consider the output of the simulation using our library. Notice that when using stepsizes 0.1 and 0.01, time t cannot be computed precisely, whereas using $t = 0.125$, which is representable in binary, we get an exact result. 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, which is also to be expected, since we have to execute more computations.

Note that this precise analysis of roundoff errors is only possible by 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.

5.3 Robustness

As a side-effect, our library can also show code to be robust in certain cases. A computation is robust, if small changes in the input only cause small changes in the output. There are two ways in which we can get a change in the output, starting from some given input.

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

Change in input causes the control flow to change. In this case, a different branch can be taken or a loop executed more or less many times, so that actually different code is executed, causing the output to differ. In each case, a comparison in a guard is involved. These comparisons are handled in a special way in our library. The library keeps a global boolean flag which is set if a comparison fails. What we mean by a comparison failing is that given the error bounds associated with an affine form, it cannot be unambigously decided whether this value is smaller or bigger than another one. Hence, for the comparison $x < y$, if we compute the difference $x - y$, then this difference includes zero.
The user may, when notified by such an 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 9.

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. This case can obviously be also detected with our library, as one 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 gives tighter bounds.

The same comparison function is also used for the methods `abs, max, min` so that the flag will be set in these cases, if a violation occurs, as well.

Illustration of control-flow robustness check. As an example for the first case of robustness violation, consider again the code in Figure 9. One can see that the setting of the global comparison flag was triggered in two of the three runs. Since there is only one comparison in this code, it is clear that the (possible) violation occured 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 $h = 0.125$, since the computation of time is exact, the flag is correctly not triggered.

6. Integration into a Programming Language

This section explains how our types are integrated 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, since 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.

6.1 Our Deployment as a Scala Library

Our library provides a wrapper type SmartFloat (AffineFloat correspondingly) that tracks all errors and that is meant to replace all 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 SmartFloat. Any remaining conflicts are signaled by the compiler's strong type-checker. By now the library handles definitions of variables and the standard arithmetic operations. In addition to this, our library supports a subset of the scala.math library functions, which we consider the most useful:

- log, expr, pow, cos, sin, tan acos, asin, atan
- abs, max, min

- constants Pi and E

The goal is to make the library as applicable for real applications as possible. That is, for any common code the developer should be able to easily adapt it, so that it will be also bounding its roundoff errors. This also includes support for the special values NaN and $\pm\infty$ with the same behavior as the original code. To accomplish such an integration, we had 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 [42] , 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 [20] interface) ScalaNumber which has a special semantics in comparisons with ==. If y is of type ScalaNumber, then both x == y and y == x delegates to y.equals(x) and thus the comparison is symmetric [45].

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 [42]. 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.

6.2 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 [46] can be used to ultimately generate code for such alternative platforms instead of JVM.

7. AffineFloat Design and Implementation

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

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

7.1 Different Interpretations of Computations

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

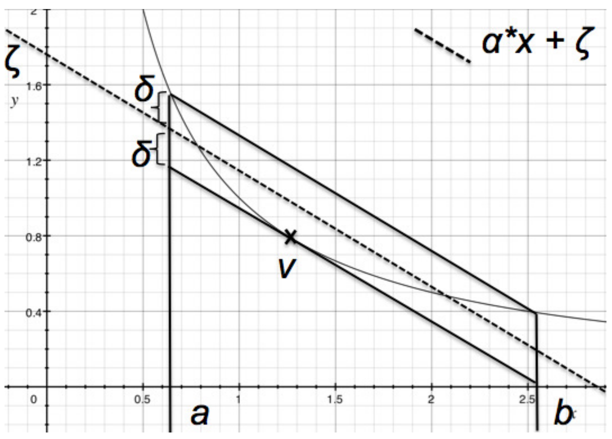
DEFINITION 2 (Original Affine Arithmetic). *In original affine arithmetic, an affine form $\hat{x}$ represents the range of real values, that is $[\hat{x}] \sim [a, b],\ a, b \in \mathbb{R}$.*

This is also the interpretation from in [15].

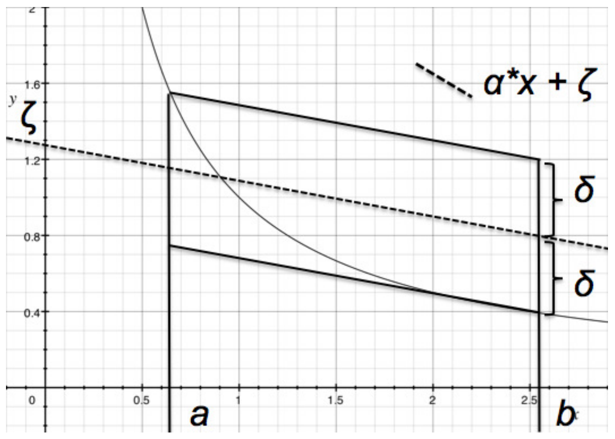
DEFINITION 3 (Exact Affine Arithmetic). *In exact affine arithmetic $\bar{x}$ represents* **one** *floating-point value and its deviation from an ideal real value. That is, if a real valued computation computed $x \in \mathbb{R}$ as the result, then it holds for the corresponding computation in floating-points that $x \in [\bar{x}]$.*

The difference to Definition 2 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 7.3.

DEFINITION 4 (Floating-point Affine Arithmetic). *In floating-point affine arithmetic $\tilde{x}$ represents a range of floating-point values, that is $[\tilde{x}] \sim [a, b],\ a, b \in \mathbb{F}$.*



(a) Chebyshev approximation



(b) Min-range approximation

Figure 10. Linear pproximations of the inverse function.

Definition 4 corresponds to our SmartFloat type and Definition 3 to AffineFloat, the details of which will be 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.

7.2 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 multiplying two affine forms:

$$\hat{x}\hat{y} = x_0y_0 + \sum_{i=1}^{n}(x_0y_i + y_0x_i)\epsilon_i + (\eta + \iota)\epsilon_{n+1}$$

where ι contains the internal errors and η an over-approximation of the nonlinear contribution. To compute the latter, several possibilities exists of varying degree of accuracy. In the case of tracking a single floating-point value the most simple way $\eta = rad(\hat{x}) * rad(\hat{y})$ 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 8.4. Division x/y is computed as $x * (1/y)$ 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 α, ζ, δ 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.

In [15] two approximations are suggested for computing $\alpha, \zeta,$ and δ: a Chebyshev (min-max) or a min-range approximation. These two are illustrated on the example of the inverse function$f(\hat{x}) = \hat{x}^{-1}$ in Figure 10. For both approximations, the algorithm first computes the interval represented by $\hat{x}$ and then works with its endpoints a and b. In both cases we want to compute the 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 intersection a line with this slope would have at a and b and fix ζ to be the average of the two. δ is then determined as the maximum deviation, which occurs by construction at either a or b.

Chebyshev Compute the slope as the slope of a line through a and b. This gives one bound on the wanted "box"(parallelepiped). To find the opposite side, we need to compute the point where the curve takes on the same slope again. Again, ζ is computed as the average of the intersections of the two lines. δ can be then computed from the maximum deviation at either the middle point v, a or b.

In general, the Chebyshev approximation computes tighter 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 $acos, asin, ...$ this can involve quite complex computations which are all committing internal roundoff errors. On big intervals, like the one considered in [15] and [16] these are (probably) not very significant. However, when keeping track of roundoff errors, we are dealing 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 [16], 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 $\log, \mathsf{exp}, \mathsf{cos}, \ldots$ requires specialized rounding, since these are correct to 1 ulp (unit in the last place) only [1], and hence less accurate than the 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.

7.3 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 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 Definition 3. For this, we assume that the central value x_0 is the floating-point value involved in the computation 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 the affine arithmetic from section 7 is not sound as the following observation shows.

OBSERVATION 5. *The algorithm for approximating non-affine operations using the min-range approximation as defined in subsection 7.2 is unsound under the interpretation of Definition 3.*

The interpretation of affine arithmetic as in Definition 3 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 round-

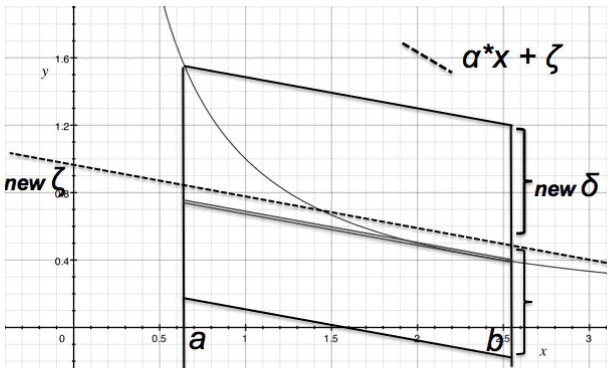


Figure 11. Modified min-range approximation of the inverse function.

off 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 $\hat{z} = \alpha * \hat{x} + zeta + \delta\epsilon_{n+1}$ do not necessarily enforce that $z_0 = \alpha * x_0 + \zeta$ equals the actual double value computed in the operation. 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 11. For non-linear operations, the new central value is computed as $z_0 = \alpha * x_0 + \zeta$. For this to equal the double value that is actually computed by the nonlinear function f (i.e. we want $f(x_0) = \alpha * x_0 + zeta$) we compute ζ as

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

The minrange approximation computes for an input range $[a, b]$ an enclosing parallepiped of a function as $\alpha * x + \zeta \pm \delta$ that is guaranteed to contain the image of the nonlinear function from this interval as computed in floating-point precision. Suppose that we have computed $\zeta = f(x_0) - \alpha * x_0$, with α computed at one of the endpoints of the interval. Since we compute the deviation δ with outwards rounding at both endpoints and keep the maximum, we soundly overapproximate 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 function are also 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 11 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 that the overapproximations do not have a big effect on the precision of our library.

7.4 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, but it is possible to limit the size of δ. In order to avoid arbitrary precision 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 [14, 44] that allow the computation of standard arithmetic operations with only ordinary floating-point operations, making the performance trade-off bearable. In this way, we can compute range reductions for the sine and cosine functions accurately enough. We 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. Currently, this is not possible to enforce in Scala, so that the library uses Java code with the `strictfp` modifier for the calculations.

The library uses the double-double precision types for the noise symbols and computations involving them only. 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.

7.5 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, the following affine form is then created: $0.03125 + (\epsilon_M * 0.03125)\epsilon_n$. This limits the over-approximations committed and provides more precise analyses when possible. For an error estimate according to Definition 3, our runtime library has the exact values available and can thus generally compute tighter bounds compared to a static analysis-based approach.

7.6 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 since 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.

7.7 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 correctness of our implementation is supported by the use of assertions certifying that certain invariants always hold. 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 [8, 39] 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.

7.8 Quadratic Equation Example

A classic example is the quadratic formula in Figure 12, 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 [19]. 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.

7.9 Managing Noise Symbols in Long Computations

The runtime performance of our library depends on the number of noise terms in each affine form, since each operation has to access each at least once. Hence, a smart compacting strategy of noise symbols becomes crucial for performance. Compacting too little means that our approach becomes unfeasible, compacting too much means that we loose too much correlation information.

Compacting algorithm. The goal of the algorithm is to take as input a list of noise terms and output a new list with less terms, while preserving the soundness of the roundoff error approximation and ideally keeping the largest correlation information. Compacting in this sense means to add the

```
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 12. Quadratic formula computed in two different ways.

absolute values of the smallest terms and to add them with a fresh noise symbol to the terms that are kept. We propose the following strategy.

- Compact all error terms smaller than 10^{-33}. These errors are smaller than the smallest double value and are thus internal errors.
- Compute the average ($avrg$) and the standard deviation ($stdDev$) of the rest of the terms. Compact all terms smaller than $avrg * a + stdDev * b$ and keep the rest. The factors a and b are user-controllable options and can be chosen separately for each computation.
- In some cases this is still not enough (for example if nearly all errors have the same magnitude). Is this case detected, we repeat this procedure one more time. In the worst case, i.e. should this also fail to produce the desired number of noise terms, the library compacts all noise symbols into a new one. In our experience, this case occurs in very few pathological examples.

8. SmartFloat Design and Implementation

The implementation described in section 7 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 numbers (according to Definition 4)
2. computation of sound roundoff error estimates over this range

Unfortunately a straight-forward re-interpretation of neither the original affine arithmetic, nor the modified version for AffineFloat give a sound range arithmetic for floating-point numbers.

OBSERVATION 6. *The roundoff computation of affine operations as defined in subsection 7.2 is unsound under the interpretation of Definition 4.*

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 values as is the case in subsection 7.2. 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 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. $x_i\epsilon_i$ and $x_i u_i$ are the noise terms characterizing the range, but 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 roundoffs; the noise terms are handled as before.

8.1 Computation of Roundoffs

To compute the roundoff error of an operation, we first compute the new range, either by using subsection 7.2 for affine operations or the min-range approximation and then compute maximum roundoff from the resulting range. Following the definition of roundoff from Equation 1 this 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 $2 * \epsilon_M$.

8.2 Propagation of Roundoffs

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

Affine. The propagation is given straightforwardly by subsection 7.2. 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 commited and κ to the new roundoff error.

Multiplication. The non-linear part in multiplication poses the difficulty that it involves cross-terms between the noise and error terms. We derive the new (albeit admittedly naive) propagation, by appending the error terms to the noise terms and computing the multiplication. Then, removing the resulting terms where the error terms do not appear again, we get new η_e

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

Note that this produces an overapproximation, since some of the errors from the error terms are also included already in the noise terms. The linear part is computed as usual by multiplication by x_0 and y_0.

Non-affine. Since the nonlinear function approximations compute α, ζ and δ, this reduces to an affine propagation of errors. Note that the factor used to propagate the roundoff errors must be the maximum slope of the function on the given range and thus does not necessarily equal α.

8.3 Additional Errors

Additional errors, e.g. method errors can be added to the affine form in the following way. we have a variable x given by $\tilde{x} = (x_0, \sum x_i\epsilon_i, \sum r_i\rho_i)$ and a the error to be added given by $\tilde{y} = (y_0, \sum y_i\epsilon_i, \sum s_i\rho_i)$. The result of adding this error is then 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.

8.4 Treatment of Range Explosion

In section 7 the naive computation of the non-linear part was sufficiently accurate due to 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, $\tilde{z} = \tilde{x} * \tilde{y} = 12 + 8\epsilon_1 + 9\epsilon_2 + 6\epsilon_3$ which gives as resulting interval $[\tilde{z}] = [-11, 35]$. This is unacceptable, if this value is subsequently used in for instance division.

Some approaches have been suggested in [50, 54], however they either change the underlying structure by using matrices instead of affine forms or are simply not scalable enough. Since we do not want to change the underlying data structure, we chose a different solution for this problem. The problem is that by computing $\eta = (\sum_{i=1}^n |x_i|) * (\sum_{i=1}^n |y_i|)$ and appending it with a fresh noise symbol correlation information is lost. Affine forms do not give us the possibility to keep quadratic terms, however, we can keep "source" information with each noise term. For example, if a noise term is computed as $x_1x_2\epsilon_1\epsilon_2$, this will result 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 an affine form represents 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. Since 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.

We will demonstrate the impact of this simple solution on a problem from [30], where an SMT-based approach was chosen, precisely for the reason that affine arithmetic produces too large over-approximations. The example 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 subcalculations: $q_1 = 331.4 + 0.6T$, $q_2 = q_1v$, $q_3 = q_1 + u$, $q_4 = q_3^2$, $z = q_2/q_4$. The parameters used are $-30°C \leq T \leq 50°C$, $20Hz \leq v \leq 20000Hz$ and $-100m/s \leq u \leq 100m/s$. The results reported in [30] for affine arithmetic and their SMT approach, as well as the ranges computed with our SmartFloat are summarized in Figure 5. Note that we obtained our results in under half a second.

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

8.5 Nested Affine Arithmetic

An even smarter version of SmartFloat would also be able to 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, where it keeps only the linear terms in the uncertainties 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 the 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 Figure 2. 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 noise symbols. The final analysis revealed 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.

9. Related Work

Affine Arithmetic. Existing implementations of affine arithmetic include [2][3]. However, they have not been used to quantify roundoff errors, only to compute ranges in the way described by the original affine arithmetic in [15]. 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 [38] in the context of plotting curves. We have decided to use affine arithmetic, since it seems to us to be a good compromise between complexity and functionality. A library based on Chebyshev and Taylor series is presented in [17], 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 [22]. Our library is developed for general purpose calculations and integrated into a programming language to provide information about floating-points for any application domain.

Estimating Roundoff Errors. The Fluctuat static analyzer [21] analyzes numerical code with operations $+, -, *, /$ for roundoff errors and their sources based on abstract interpretation. Because Fluctuat is not publicly available, we were not able to compare it with our system. Further work in abstract interpretation includes the Astrée analyzer [13] and APRON [12, 26]. 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). A recent approach [25] statically detects loss of precision in floating-point computations using bounded model checking with SMT solvers, but uses interval arithmetic for scalability reasons. [18] 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 [36], 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 [27]. However, the stochastic approach does not provide rigorous bounds, since for example in loops, roundoff errors are not uniformly distributed.

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 [11, 37].

Finite-Precision Arithmetic. [31] uses affine arithmetic for bit-width optimization and provides an overview of related approaches. [48] 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. [30] 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. However, a timeout has to be included for when this becomes too expensive.

Theorem Proving Approaches. Researchers have used theorem proving to verify floating-point programs [6, 7, 23, 40, 47]. 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. [33] extend previous work using affine arithmetic by considering the problem of reducing precision for performance reasons, however the work remains interactive. [10] 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 [9], 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, however, 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. Moreover, it computes 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 SmartFloat allows developers to estimate the error behavior on entire classes of inputs using a single run. We have found the performance and the precision of SmartFloat 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/Java SE 6/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] S. Boldo, J.-C. Filliâtre, and G. Melquiond. Combining Coq and Gappa for Certifying Floating-Point Programs. In *CICM*, 2009.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[31] 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.

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

[33] 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.

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

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

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

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

[38] 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.

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

[40] 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.

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

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

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

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

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

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

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

[48] 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.

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

[50] 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.

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

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

[53] 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.

[54] 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.

11. Annex

This section provides the code used for computing the non-linear approximations in AffineFloat. The code given is identical to the one used in AffineFloat, but pretty-printed for readability reasons. SmartFloat uses the same code in principle, except where modifications are necessary for soundness reasons. Please see **??** for details. The following notation is used

- $+^{\downarrow}$ and $+^{\uparrow}$ denote addition rounded towards $-\infty$ and ∞ respectively. Similarly for the other arithmetic operations.
- $+_{DD}$ denotes addition performed in double-double precision.
- $\mathbb{DD}(x)$ denotes a conversion of x into double-double precision. When the conversion is obvious from the context, we omitted explicit conversion for readability reasons.
- The helper functions addQueues, subtractQueues and multiplyQueues are not listed, as they do not do anything particularly interesting. For instance addQueues takes as input two queues representing $\sum x_i \epsilon_i$ and $\sum y_i \epsilon_i$ and computes $\sum (x_i + y_i)\epsilon_i$.
- roundOff is an auxiliary function returning the maximum roundoff error, assuming exact rounding and correctly rounded operations.
- We have implemented our own Queue data structure for performance reasons. The operation queue :+ noiseTerm appends the noiseTerm to the queue (modifies it).
- Since Scala has a very strong type inference, we have added type information in some places for clarity, e.g. **var** delta$_{DD}$ = ... denotes that this variable is of type DoubleDouble.
- By $\downdownarrows$ and $\upuparrows$ we denote the specialized rounding that takes into account the max roundoff specified by the Java java.lang.Math API.

The following class names are used:

- SymAForm is the name of the affine form used by AffineFloat. It stands for Symmetric Affine Form. The class has two fields x0 and xnoise, denoting the central value and the noise terms respectively.
- EmptyForm is a special AffineForm denoting the equivalent of NaN (in Double). FullForm similarly denotes $\pm\infty$.
- Noise terms are represented by the class Noise which has fields for index and value.

11.1 Affine operations

```
def +(other: AffineForm): AffineForm = other match {
  case SymAForm(y0, ynoise) => {
    val z0 = x0 + y0
    if(z0 == ±∞) return FullForm

    var delta_DD = if ((x0 $+^{\downarrow}$ y0) == (x0 $+^{\uparrow}$ y0)) 0 else roundOff(z0)

    var (deviation, rd) = addQueues(xnoise, ynoise)
    delta = delta $+^{\uparrow}_{DD}$ rd

    if (delta == ∞) return FullForm
    if (delta != 0.0) deviation :+ new Noise(newIndex, delta)
    return new SymAForm(z0, deviation)
  }
  case EmptyForm => return EmptyForm
  case FullForm => return FullForm
}


def −(other: AffineForm): AffineForm = other match {
  case SymAForm(y0, ynoise) => {
    val z0 = x0 − y0
    if(z0 == ±∞) return FullForm

    var delta_DD = if ((x0 $-^{\downarrow}$ y0) == (x0 $-^{\uparrow}$ y0)) 0 else roundOff(z0)

    var (deviation, rd) = subtractQueues(xnoise, ynoise)
    delta = delta $+^{\uparrow}_{DD}$ rd
    if(delta == ∞) return FullForm
    if (delta != 0.0) deviation :+ new Noise(newIndex, delta)
    return new SymAForm(z0, deviation)
  }
  case EmptyForm => return EmptyForm
  case FullForm => return FullForm
}


def unary_−(): AffineForm = {
  var deviation = new Queue
  val iter = xnoise.getIterator
  while(iter.hasNext) {
    val xi = iter.next
    deviation :+ − xi //no round−off error
  }
  new SymAForm(−x0, deviation)
}
```

11.2 Multiplication

```
def $*$(other: AffineForm): AffineForm = other match {
  case SymAForm(y0, ynoise) => {
    //nonlinear part, naive approximation
    var delta = this.radiusExt $*_{DD}^{\uparrow}$ other.radiusExt

    val z0 = x0 $*$ y0
    if(z0 == $\pm\infty$) return FullForm

    if ((x0 $*^{\downarrow}$ y0) != (x0 $*^{\uparrow}$ y0)) delta = delta $+_{DD}^{\uparrow}$ roundOff(z0)

    var (deviation, rd) = multiplyQueues(x0, xnoise, y0, ynoise)
    delta = delta $+_{DD}^{\uparrow}$ rd

    if(delta == $\infty$) return FullForm
    if (delta != 0.0) deviation :+ new Noise(newIndex, delta)
    return new SymAForm(z0, deviation)
  }
  case EmptyForm => return EmptyForm
  case FullForm => return FullForm
}
```

11.3 Division

Division is computed as $x * \frac{1}{y}$. Note that the division procedure calls a special multiplication method that takes a "hint" to make sure that the central value equals the double value that would be computed. This is necessary as $x * \frac{1}{y}$ is not necessarily equal to $\frac{x}{y}$ in floating-point (although the difference is very small).

```
def /(other: AffineForm): AffineForm = other match {
  case SymAForm(y0, ynoise) =>
    val (yloD, yhiD) = other.interval
    val (ylo$_{DD}$, yhi$_{DD}$) = other.interval$_{DD}$
    if(yloD <= 0.0 && yhiD >= 0.0) return FullForm //division by zero

    if(ynoise.size == 0.0) { //exact
      val inv = 1.0/y0
      if((1.0 $/^{\downarrow}$ y0) == (1.0 $/^{\uparrow}$ y0)) return this $*$ (SymAForm(inv, Queue.empty), x0/y0)
      else return this $*$ (SymAForm(inv, new Queue(new Noise(newIndex, roundOff(inv)))) , x0/y0)
    }

    /* Calculate the inverse. */
    val (a, b) = (min(|ylo|, |yhi|), max(|ylo|, |yhi|))
    val (ad, bd) = (min(|yloD|, |yhiD|), max(|yloD|, |yhiD|))

    val alpha = $-$1.0 $/_{DD}$ ( b $*_{DD}$b )

    val dmax = $\mathbb{DD}$(1.0 $/^{\uparrow}$ ad) $-_{DD}^{\uparrow}$ (alpha $*_{DD}^{\downarrow}$ a))
    val dmin = $\mathbb{DD}$(1.0 $/^{\downarrow}$ bd) $-_{DD}^{\downarrow}$ (alpha $*_{DD}^{\uparrow}$ b))

    var (zeta, rdoff) = computeZeta(1.0/y0, alpha, y0)
    if(yloD < 0.0) zeta = $-$zeta

    val delta = computeDelta(zeta, dmin, dmax) $+_{DD}^{\uparrow}$ rdoff
    val inverse = symmetricUnary(1.0/y0, ynoise, alpha, zeta, delta)
    return this $*$(inverse, x0/y0)

  case EmptyForm => return other
  case FullForm => return other
}
```

11.4 Unary functions

The computation of unary functions uses these auxiliary methods

- computeZeta computes soundly: (dmin + dmax) / 2
- computeDelta computes soundly: (dmin − dmax) / 2
- symmetricUnary computes $\alpha * x_0 + \zeta$ and $\sum^n (\alpha * x_i)\epsilon_i + \delta\epsilon_{n+1}$ to obtain a new affine form
- computeExactResult computes the result for the case where the affine form is exact, i.e. the number of noise symbols is zero. The method is similar to the procedure used in square root, except that the rounding takes into account the 1 ulp condition on Java Math API functions.
- interval2affine converts an interval into an affine form.
- math.sqrt calls the Scala library function, i.e. this is really a call to scala.math.sqrt. Similarly for the other methods.

11.5 Square root

```
def squareRoot: AffineForm = {
  var (ad, bd) = interval

  if(bd < 0.0) return EmptyForm
  if(bd == $\infty$) return FullForm

  if(xnoise.size == 0) { //exact
    val sqrt = math.sqrt(x0)
    if(sqrt$^{\downarrow}$(x0) == sqrt$^{\uparrow}$(x0)) return new SymAForm(sqrt, Queue.empty)
    else return new SymAForm(sqrt, new Queue(new Noise(newIndex, roundOff(sqrt))))
  }
  var (a$_{DD}$, b$_{DD}$) = interval$_{DD}$
  if(ad < 0.0) {ad = 0.0; a = zero} //soft policy

  val sqA = $\mathbb{DD}$(math.sqrt$^{\downarrow}$(ad))
  val sqB = $\mathbb{DD}$(math.sqrt$^{\uparrow}$(bd))
  val alpha = 0.5 $/_{DD}$ sqrt$_{DD}$(b)

  val dmin = sqA $-^{\downarrow}_{DD}$ (alpha $*^{\uparrow}_{DD}$ a)
  val dmax = sqB $-^{\uparrow}_{DD}$ (alpha $*^{\downarrow}_{DD}$ b)

  var (zeta, rdoff) = computeZeta(math.sqrt(x0), alpha, x0)
  val delta = computeDelta(zeta, dmin, dmax) $+^{\uparrow}_{DD}$ rdoff
  return symmetricUnary(math.sqrt(x0), xnoise, alpha, zeta, delta)
}
```

11.6 Logarithm

```
def ln: AffineForm = {
    if(xnoise.size == 0) return computeExactResult(x0, math.log)

    val (ad, bd) = interval
    if(ad <= 0.0 || bd < 0.0 || bd == $\infty$) return FullForm

    var (a$_{DD}$, b$_{DD}$) = interval$_{DD}$

    val alpha = 1.0 $/_{DD}$ b
    val dmin = $\mathbb{DD}\downdownarrows$(math.log(ad)) $-^{\downarrow}_{DD}$ (alpha $*^{\uparrow}_{DD}$ a)
    val dmax = $\mathbb{DD}\upuparrows$(math.log(bd)) $-^{\uparrow}_{DD}$ (alpha $*^{\downarrow}_{DD}$b)

    var (zeta, rdoff) = computeZeta(math.log(x0), alpha, x0)
    val delta = computeDelta(zeta, dmin, dmax) $+^{\uparrow}_{DD}$ rdoff

    return symmetricUnary(math.log(x0), xnoise, alpha, zeta, delta)
}
```

11.7 Exponential

```
def exponential: AffineForm = {
    if(xnoise.size == 0) return computeExactResult(x0, math.exp)

    val (ad, bd) = interval
    if(bd == $\infty$) return FullForm

    var (a$_{DD}$, b$_{DD}$) = interval$_{DD}$
    val expA = $\mathbb{DD}\downdownarrows$(math.exp(ad))
    val expB = $\mathbb{DD}\upuparrows$(math.exp(bd))

    val alpha = expA
    val dmin = expA $*^{\downarrow}_{DD}$ (1.0 $-^{\downarrow}_{DD}$ a)
    val dmax = expB $-^{\uparrow}_{DD}$ (alpha $*^{\downarrow}_{DD}$b)

    var (zeta, rdoff) = computeZeta(math.exp(x0), alpha, x0)
    val delta = computeDelta(zeta, dmin, dmax) $+^{\uparrow}_{DD}$ rdoff

    return symmetricUnary(math.exp(x0), xnoise, alpha, zeta, delta)
}
```

11.8 Cosine

```
def cosine: AffineForm = {
    if(radius$_{DD}$ $>$ $2\pi_{DD}$) return new SymAForm(0.0, new Queue(new Noise(newIndex, 1.0)))

    if(xnoise.size == 0) computeExactResult(x0, math.cos)

    var (xlo, xhi) = interval
    var (xloExt$_{DD}$, xhiExt$_{DD}$) = interval$_{DD}$

    // range reduction.
    val ka = if(xlo > 0.0) (2.0*xlo $/^{\downarrow}$ $\pi^{\uparrow}$) else (2.0*xlo $/^{\downarrow}$ $\pi^{\downarrow}$)
    val kb = if(xhi > 0.0) (2.0*xhi $/^{\uparrow}$ $\pi^{\downarrow}$) else (2.0*xhi $/^{\uparrow}$ $\pi^{\uparrow}$)
    val m = math.floor(ka).toLong
    val n = math.ceil(kb).toLong

    if(n−m < 2.0) {
      val k = math.floor(xlo/$2\pi$)
      val kD = $\mathbb{DD}$(k)
      val r = (m % 4).toLong

      val aExt = $2\pi$ $*^{\downarrow}_{DD}$ ((xloExt $/^{\downarrow}_{DD}$ $2\pi$) $-^{\downarrow}_{DD}$ kD)
      val bExt = $2\pi$ $*^{\uparrow}_{DD}$ ((xhiExt $/^{\uparrow}_{DD}$ $2\pi$) $-^{\uparrow}_{DD}$ kD)

      val alpha = $\mathbb{DD}$(− math.sin(xlo))
      val (dmin$_{DD}$, dmax$_{DD}$) =
        if(r == 0 || r == −1 || r == 3) { //y > 0
          (($\mathbb{DD}\downdownarrows$(math.cos(xhi) $-^{\downarrow}_{DD}$(alpha $*^{\uparrow}_{DD}$ bExt)),
           ($\mathbb{DD}\upuparrows$(math.cos(xlo) $-^{\uparrow}_{DD}$(alpha $*^{\downarrow}_{DD}$ aExt)))
        }
        else { //y < 0
          (($\mathbb{DD}\downdownarrows$(math.cos(xlo) $-^{\downarrow}_{DD}$(alpha $*^{\uparrow}_{DD}$ aExt)),
           ($\mathbb{DD}\upuparrows$(math.cos(xhi) $-^{\uparrow}_{DD}$(alpha $*^{\downarrow}_{DD}$ bExt)))
        }

      val x0D = $\mathbb{DD}$(x0)
      val x0_new = $2\pi$ $*_{DD}$((x0D $/_{DD}$ $2\pi$) $-_{DD}$ kD)
      val x0_down = $2\pi$ $*^{\downarrow}_{DD}$((x0D $/^{\downarrow}_{DD}$ $2\pi$) $-^{\downarrow}_{DD}$ kD)
      val x0_up = $2\pi$ $*^{\uparrow}_{DD}$((x0D$/^{\uparrow}_{DD}$ $2\pi$) $-^{\uparrow}_{DD}$ kD)

      var delta = max((x0_up $-^{\uparrow}_{DD}$ x0_new),(x0_new $-^{\uparrow}_{DD}$ x0_down))
      var (zeta, rdoff) = computeZeta(math.cos(x0), alpha, x0_new)
      delta = computeDelta(zeta, dmin, dmax) $+^{\uparrow}_{DD}$ rdoff

      return symmetricUnary(math.cos(x0), xnoise, alpha, zeta, delta)
    }
    else
      return interval2affine(interval.cosine, math.cos(x0))
  }
```

11.9 Sine

```
def sine: AffineForm = {
  if(radius$_{DD}$ $>$ $2\pi_{DD}$) return new SymAForm(0.0, new Queue(new Noise(newIndex, 1.0)))

  if(xnoise.size == 0) computeExactResult(x0, math.sin)

  var (xlo, xhi) = interval
  var (xloExt$_{DD}$, xhiExt$_{DD}$) = interval$_{DD}$

  // range reduction.
  val ka = if(xlo $>$ 0.0) (2.0$*$xlo $/^{\downarrow}$ $\pi^{\uparrow}$) else(2.0$*$xlo $/^{\downarrow}$ $\pi^{\downarrow}$)
  val kb = if(xhi $>$ 0.0) (2.0$*$xhi$/^{\uparrow}$ $\pi^{\downarrow}$) else(2.0$*$xhi $/^{\uparrow}$ $\pi^{\uparrow}$)
  val m = math.floor(ka).toLong
  val n = math.ceil(kb).toLong


  if(n$-$m $<$ 2.0) {
    val k = math.floor(xlo$/2\pi$)
    val kD = $\mathbb{DD}$(k)
    val r = (m % 4).toLong

    val aExt = $2\pi$ $*^{\downarrow}_{DD}$ ((xloExt $/^{\downarrow}_{DD}$ $2\pi$) $-^{\downarrow}_{DD}$ kD)
    val bExt = $2\pi$ $*^{\uparrow}_{DD}$ ((xhiExt $/^{\uparrow}_{DD}$ $2\pi$) $-^{\uparrow}_{DD}$ kD)


    val alpha = $\mathbb{DD}$(math.cos(xlo))
    val (dmin$_{DD}$, dmax$_{DD}$) =
      if(r == 0 || r == 1 || r == $-3$) { //$y > 0$
        (($\mathbb{DD}\downarrow\downarrow$(math.sin(xhi) $-^{\downarrow}_{DD}$ (alpha $*^{\uparrow}_{DD}$ bExt)),
         ($\mathbb{DD}\uparrow\uparrow$(math.sin(xlo) $-^{\uparrow}_{DD}$ (alpha $*^{\downarrow}_{DD}$ aExt)))
      }
      else { //$y < 0$
        (($\mathbb{DD}\downarrow\downarrow$(math.sin(xlo) $-^{\downarrow}_{DD}$ (alpha $*^{\uparrow}_{DD}$ aExt)),
         ($\mathbb{DD}\uparrow\uparrow$(math.sin(xhi) $-^{\uparrow}_{DD}$ (alpha $*^{\downarrow}_{DD}$ bExt)))
      }

    val x0D = $\mathbb{DD}$(x0)
    val x0_new = $2\pi$ $*_{DD}$((x0D $/_{DD}$ $2\pi$) $-_{DD}$ kD)
    val x0_down = $2\pi$ $*^{\downarrow}_{DD}$((x0D $/^{\downarrow}_{DD}$ $2\pi$) $-^{\downarrow}_{DD}$ kD)
    val x0_up = $2\pi$ $*^{\uparrow}_{DD}$((x0D $/^{\uparrow}_{DD}$ $2\pi$) $-^{\uparrow}_{DD}$ kD)

    var delta = max((x0_up $-^{\uparrow}_{DD}$ x0_new), subUp(x0_new $-^{\uparrow}_{DD}$ x0_down))
    var (zeta, rdoff) = computeZeta(math.sin(x0), alpha, x0_new)
    delta =computeDelta(zeta, dmin, dmax) $+^{\uparrow}_{DD}$ rdoff

    return symmetricUnary(math.sin(x0), xnoise, alpha, zeta, delta)
  }
  else
    return interval2affine(interval.sine, math.sin(x0))
}
```

11.10 Tangent

```
def tangent: AffineForm = {
    if(radius$_{DD}$ > $\pi_{DD}$) return FullForm

    if(xnoise.size == 0) computeExactResult(x0, math.tan)

    var (xlo, xhi) = interval
    var (xloExt$_{DD}$, xhiExt$_{DD}$) = interval$_{DD}$

    val ka = if(xlo > 0.0) (2.0*xlo $/^{\downarrow}$ $\pi^{\uparrow}$) else(2.0*xlo $/^{\downarrow}$ $\pi^{\downarrow}$)
    val kb = if(xhi > 0.0)(2.0*xhi $/^{\uparrow}$ $\pi^{\downarrow}$) else (2.0*xhi $/^{\uparrow}$ $\pi^{\uparrow}$)
    val m = math.floor(ka).toLong
    val n = math.ceil(kb).toLong

    if(n−m < 2.0) {
      val (alpha, dmin, dmax, k) =
        if(m % 2 == 0) { //y > 0
          val kx = math.floor(2.0 * xlo / $\pi$)
          val kD = $\mathbb{DD}$(kx)
          val aExt = $\pi/2$ $*^{\downarrow}_{DD}$ ((xloExt $/^{\downarrow}_{DD}$ $\pi/2$) $-^{\downarrow}_{DD}$ kD)
          val bExt = $\pi/2$ $*^{\uparrow}_{DD}$ ((xhiExt $/^{\uparrow}_{DD}$ $\pi/2$) $-^{\uparrow}_{DD}$ kD)

          val alphax = $\mathbb{DD}$(1.0 + math.tan(xlo)*math.tan(xlo))

          (alphax, ($\mathbb{DD}\downarrow\downarrow$(math.tan(xlo)) $-^{\downarrow}_{DD}$ (alphax $*^{\uparrow}_{DD}$ aExt)),
            ($\mathbb{DD}\uparrow\uparrow$(math.tan(xhi)) $-^{\uparrow}_{DD}$ (alphax $*^{\downarrow}_{DD}$ bExt)), kD)
        }
        else { //y < 0
          val kx = math.floor(2.0 * xlo/$\pi$) + 1.0
          val kD = $\mathbb{DD}$(kx)
          val aExt = $\pi/2$ $*^{\downarrow}_{DD}$ ((xloExt $/^{\downarrow}_{DD}$ $\pi/2$) $-^{\downarrow}_{DD}$ kD)
          val bExt = $\pi/2$ $*^{\uparrow}_{DD}$ ((xhiExt $/^{\uparrow}_{DD}$ $\pi/2$) $-^{\uparrow}_{DD}$ kD)

          val alphax = $\mathbb{DD}$(1.0 + math.tan(xhi)*math.tan(xhi))

          (alphax, ($\mathbb{DD}\downarrow\downarrow$(math.tan(xlo)) $-^{\downarrow}_{DD}$ (alphax $*^{\uparrow}_{DD}$ aExt)),
           ($\mathbb{DD}\uparrow\uparrow$(math.tan(xhi)) $-^{\uparrow}_{DD}$ (alphax $*^{\downarrow}_{DD}$ bExt)), kD)
        }

      val x0D = $\mathbb{DD}$(x0)
      val x0_new = $\pi/2$ $*_{DD}$ ((x0D $/_{DD}$ $\pi/2$) $-_{DD}$ k)
      val x0_down = $\pi/2$ $*^{\downarrow}_{DD}$ ((x0D $/^{\downarrow}_{DD}$ $\pi/2$) $-^{\downarrow}_{DD}$ k)
      val x0_up = $\pi/2$ $*^{\uparrow}_{DD}$ ((x0D $/^{\uparrow}_{DD}$ $\pi/2$) $-^{\uparrow}_{DD}$ k)

      var delta = max((x0_up $-^{\uparrow}_{DD}$ x0_new), (x0_new $-^{\uparrow}_{DD}$ x0_down))
      var (zeta, rdoff) = computeZeta(math.tan(x0), alpha, x0_new)
      delta = computeDelta(zeta, dmin, dmax) $+^{\uparrow}_{DD}$ rdoff

      return symmetricUnary(math.tan(x0), xnoise, alpha, zeta, delta)
    }
    else if(n−m == 2.0) {
      return interval2affine(interval.tangent, math.tan(x0))
    }
    else {
      FullForm
    }
  }
```

11.11 Arccosine

```
def arccosine: AffineForm = {
    var (a, b) = interval
    var (aExt$_{DD}$, bExt$_{DD}$) = interval$_{DD}$

    if(b $< -1.0$ || a $> 1.0$) return EmptyForm
    if(a == b) computeExactResult(x0, math.acos)
    if(a $< 0.0$ && b $> 0.0$) return interval2affine(interval.arccosine, math.acos(x0))

    if(b $> 1.0$) {b = 1.0; bExt = one}
    if(a $< -1.0$) {a = $-1.0$; aExt = _one}

    val alpha$_{DD}$=
      if(a $> 0.0$) ($-1$ $/_{DD}$ sqrt$_{DD}$(1 $-_{DD}$ aExt $*_{DD}$ aExt))
      else ($-1$ $/_{DD}$ sqrt$_{DD}$(1 $-_{DD}$ bExt $*_{DD}$ bExt))

    val dmin = $\mathbb{DD}\downdownarrows$(math.acos(b)) $-^{\downarrow}_{DD}$ (alpha $*^{\uparrow}_{DD}$ bExt)
    val dmax = $\mathbb{DD}\upuparrows$(math.acos(a)) $-^{\uparrow}_{DD}$ (alpha $*^{\downarrow}_{DD}$ aExt)
    var (zeta, rdoff) = computeZeta(math.acos(x0), alpha, x0)
    val delta = computeDelta(zeta, dmin, dmax) $+^{\uparrow}_{DD}$ rdoff

    return symmetricUnary(math.acos(x0), xnoise, alpha, zeta, delta)
}
```

11.12 Arcsine

```
def arcsine: AffineForm = {
    var (a, b) = interval
    var (aExt$_{DD}$, bExt$_{DD}$) = interval$_{DD}$

    if(b $< -1.0$ || a $> 1.0$) return EmptyForm
    if(a == b) computeExactResult(x0, math.asin)
    if(a $< 0.0$ && b $> 0.0$) return interval2affine(interval.arcsine, math.asin(x0))

    if(b $> 1.0$) {b = 1.0; bExt = one}
    if(a $< -1.0$) {a = $-1.0$; aExt = _one}

    val alpha$_{DD}$ =
      if(a $> 0.0$) (1 $/_{DD}$ sqrt$_{DD}$(1 $-_{DD}$ aExt $*_{DD}$ aExt))
      else (1 $/_{DD}$ sqrt$_{DD}$(1 $-_{DD}$ bExt $*_{DD}$ bExt))

    val dmin = $\mathbb{DD}\downdownarrows$(math.asin(a)) $-^{\downarrow}_{DD}$ (alpha $*^{\uparrow}_{DD}$ aExt)
    val dmax = $\mathbb{DD}\upuparrows$(math.asin(b)) $-^{\uparrow}_{DD}$ (alpha $*^{\downarrow}_{DD}$ bExt)
    var (zeta, rdoff) = computeZeta(math.asin(x0), alpha, x0)
    var delta = computeDelta(zeta, dmin, dmax) $+^{\uparrow}_{DD}$ rdoff

    return symmetricUnary(math.asin(x0), xnoise, alpha, zeta, delta)
}
```

11.13 Arctan

```
def arctangent: AffineForm = {
  var (a, b) = interval
  var (aExt$_{DD}$, bExt$_{DD}$) = interval$_{DD}$

  if(a == b) computeExactResult(x0, math.atan)
  if(a < 0.0 && b > 0.0) return interval2affine(interval.arctangent, math.atan(x0))

  val alpha$_{DD}$ =
    if(a > 0.0) (1 $/_{DD}$ (1 $+_{DD}$ (aExt $*_{DD}$ aExt)))
    else (1 $/_{DD}$ (1 $+_{DD}$ (bExt $*_{DD}$ bExt)))

  val dmin = ($\mathbb{DD}\downarrow\downarrow$(math.atan(b)) $-^{\downarrow}_{DD}$ (alpha $*^{\uparrow}_{DD}$ bExt))
  val dmax = ($\mathbb{DD}\uparrow\uparrow$(math.atan(a)) $-^{\uparrow}_{DD}$ (alpha $*^{\downarrow}_{DD}$ aExt))
  var (zeta, rdoff) = computeZeta(math.atan(x0), alpha, x0)
  var delta = computeDelta(zeta, dmin, dmax) $+^{\uparrow}_{DD}$ rdoff

  return symmetricUnary(math.atan(x0), xnoise, alpha, zeta, delta)

}
```