On Rigorous Numerical Computation as a Scala Library

EPFL-REPORT-158754

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 programming languages have adopted the floating point type as a way to describe computations with real numbers. Thanks to the 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 library seamlessly integrates into the Scala programming language, thanks to its extensibility mechanisms such as implicit conversions and the treatment of arithmetic operations as method calls. Our numerical data type library tracks a (double) floating point value, but also an upper bound on the error between this value and the ideal value that would be computed in the real-value semantics. The library supports 1) an interval-based representation of the error, and 2) an affine arithmetic representation, which is generally more precise and keeps track of the correlation between different numerical values in the program. The library tracks errors arising from the rounding in arithmetic operations and constants, as well as user-provided errors that can model method errors of numerical algorithms or measurement errors arising in cyber-physical system applications. Our library provides approximations for most of the standard mathematical operations, including trigonometric functions. The library supports automated demand-driven refinement of computed errors by lazily increasing the precision of iteratively computed values to meet the desired precision of the final expression. Furthermore, the library supports dynamic transformation of the evaluation order following a set of algebraic rules to reduce the estimated error in the computed value. The transformed expressions can be used to suggest static rewrites of the source code to the developer. 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, as well as systems that need to reason about the parameters of the physical world. The IEEE standard [37] 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 [17, 29, 35]. 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 [18], in which the largest discrepancies among different software versions were found in numerical computation code.

One of the main difficulties 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. The floating point arithmetic is often, but not always sufficient to obtain a desired result, yet the developer currently receives little feedback on how accurate the computation is.

The goal of our work is to provide the developer with the information on how the inaccuracies propagate, and to suggest changes to the program to obtain sufficient accuracy and performance. The sources of inaccuracies that our system analyzes arise from roundoff in floating point computation, but also from the approximate nature of the measured quantities that the variables in a program represent. We aim to provide programmers with an easy-to-use system, which they can use in a programming language much like the standard floating point data types (such as double). For these reasons, we deploy our solution as a library to track numerical computation, estimate the errors, suggest alternative forms of expressions, and suggest a sufficient number of iterations for numerical algorithms.

Contributions. This paper makes the following contributions.

1. We develop a data type that provides rigorous upper bounds on the errors arising from the floating point computation, as well as user-specified errors on input variables (arising from, e.g. physical measurements or iterative numerical methods). Our data type computes practically useful error bounds through a combination of affine arithmetic and intervals.
2. Our affine arithmetic implementation goes beyond the existing ones because it 1) supports a large number of non-linear and transcendental functions, and 2) contains a mechanism to soundly bound the number of error terms, ensuring predictable performance.
3. We describe the deployment of our data type in the Scala programming language [31]. Thanks to Scala's treatment of operators as methods and the presence of the implicit conversions, the developers can use the resulting data type almost identically as the built-in primitive Double type.
4. Our library supports syntactic expression interpretation of arithmetic operations: it dynamically builds the syntax trees of numerical expressions and evaluates them on demand to control the precision and performance.
5. As one application of the syntactic interpretation, we describe rule-based expression rewriting. Expression rewriting automatically discovers alternative expressions that yield the same value

in real-number semantics, yet produce a more accurate result in the affine arithmetic.

6. As another application, we present the mechanism to perform demand-driven iterative numerical computations. The expression trees at runtime contain leaves with values initially computed with few iterations. Our system refines the values on demand to resolve e.g. the truth values of comparisons.

2. Examples

We illustrate the capabilities of our system through a number of examples.

2.1 Keeping track of time

In simulations it is common to use a variable to keep track of time. Developers may forget however, that not all numbers can be represented in binary and hence a small roundoff error is introduced at each step. Running the code in Figure 1 with our SmartFloat library type shows that indeed the first version is not accurate, but that using 0.125 as an increment, which can be represented in binary, produces an exact result.

2.2 Quadratic formula

Another common example to show the potential pitfalls of floating-point numbers is the quadratic formula in Figure 2, because it produces less accurate results (two orders of magnitude in this example), when one root is much smaller. Our library makes this effect visible through the computed errors. Figure 3 shows the result of rewriting this code following the method in [13]. Our library confirms that both roots are now computed with approximately the same accuracy.

2.3 Triangle

The textbook formula for computing the area of a triangle exhibits inaccuracy when the triangle is flat. A formula proposed by W. Kahan [23] rearranges the calculation such that more significant digits are correct. In Figure 4 we apply our automated equation rewriting technique (subsection 6.3) to one such flat triangle. The library suggests the following rewritten formula

$$\sqrt{\frac{(a+(b+c))*(-a+b+c)*(a-b+c)*(a+b-c)}{16}}$$

which, when evaluated, gives the same result in double precision as the approach by Kahan.

2.4 Error tolerance

It is often convenient to ignore very small differences between the numerical quantities. To achieve this, our library supports a user-defined error tolerance. An illustrative example code is in Figure 5. Note that the error on z is too small to influence the control flow. For

```
var time = SmartFloat(0.0)
for (i ← 0.until(864000)) time = time + 0.1
println("time 1: " + time.toStringAffine)

var time2 = SmartFloat(0.0)
for (i ← 0.until(691200)) time2 = time2 + 0.125
println("time 2: " + time2.toStringAffine)

> time 1: 86400.00000054126 (4.796186625010571E-11)
  time 2: 86400.0 (0.0)
```

Figure 1. Keeping track of time with time step 0.1 and 0.125. The numbers in parentheses denote the bounds on the relative errors.

```
var a = SmartFloat(3.0)
var b = SmartFloat(56.0)
var c = SmartFloat(1.0)
val discr = b*b − a * c * 4.0
var r2 = (−b + sqrt(discr))/(a * 2.0)
var r1 = (−b − sqrt(discr))/(a * 2.0)
println("r1="+r1.toStringInterval +" ,r2="+r2. toStringInterval)

> r1 = -18.648792408321412 (1.90506366364785E-16) ,
  r2 = -0.017874258345252986 (6.63832218741081E-14)
```

Figure 2. Classic quadratic formula.

```
val discr = b*b − a * c * 4.0
val (rk1: SmartFloat, rk2: SmartFloat) =
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("r1="+rk1.toStringInterval+",r2="+rk2.toStringInterval)

> r1 = -18.648792408321412 (1.90506366364785E-16),
  r2 = -0.01787425834525319 (3.882059758719987E-16)
```

Figure 3. Rewritten quadratic formula.

```
val a = SmartFloat(9.01, "a")
val b = SmartFloat(4.503, "b")
val c = SmartFloat(4.5092, "c")
val s = (a + b + c)/2.0
val area = sqrt(s * (s − a) * (s − b) * (s − c))
analyze(area)
```

Figure 4. Rewriting of the textbook formula for the area of a triangle.

```
errorTolerance = 1e−10
var x = SmartFloat(11.1)
var y = x + 0.00001
var z = x + 1.0e−11
if(x == z) println("x and z are equal")
else println("x and z are not equal")

if(x == y) println("x and y are equal")
else println("x and y are not equal")

errorTolerance = 1.0e−14
var f = (x−x) + (y−y) + (z−z)

if (f == 0.0) println("f is indeed 0")
println(f.toStringInterval)
println(f.toStringAffine)

> x and z are equal
  x and y are not equal
  f is indeed 0
  0.0 (1.4210854715202004E-14)
  0.0 (0.0)
```

Figure 5. Use of user-defined error tolerance.

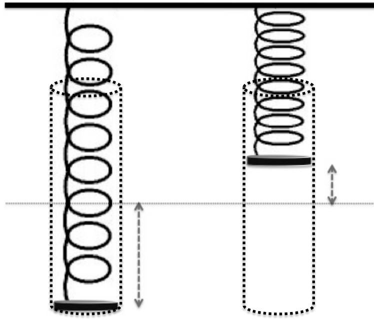


Figure 8. Two springs with two different amplitudes. We measure their collective extension from the middle the grey line.

the last test `f == 0`, the expression is evaluated first with interval arithmetic. This test fails, which we can see on the second last line in the output, as the error computed is larger than the error tolerance. Then, on the second evaluation with affine arithmetic, the correct truth value **true** can be determined. This illustrates a simple case in which our library gradually estimates the error using more and more precise methods. The next subsection shows a more radical approach to automatically refining errors due to the computation method itself.

2.5 Springs

Our system also supports quantities defined in an iterative fashion. As an example, consider a spring whose behavior is defined by a differential equation (Figure 6). The spring's position is calculated by simulating its behavior. The accuracy of the simulation is determined by the parameter i that controls the time step of the simulation. Because the method error outweighs the roundoff errors in this case, time steps of $0.1, 0.01, ...$ are acceptable. For an example application, consider the setup in Figure 8, where two springs with different parameters oscillate inside two tanks. After a certain time, say $1s$, a controller takes an action based on the amount of the total volume of the displaced fluid. Figure 7 shows the corresponding simplified piece of code. In this case, it is not necessary to know the exact values; a coarse estimate is sufficient. In the first case, the position of both springs is determined with a time step of $dt = 0.0001$. In the second case, only a very coarse estimate is computed initially, with $dt = 0.1$. The estimate is then automatically refined by our library on demand to decide the condition of the if-statement. The result is a difference in computation times: the second version takes only half the time to compute, because it turns out that the first spring should be computed with $dt = 0.0001$, whereas $dt = 0.01$ suffices for the second spring. The SmartFloat prints the number of iterations for each iterative variable that were needed to achieve the desired accuracy. The developer can then use this information to improve the final code, even if the final code ends up using standard Doubles and a pre-determined precision of iterative algorithms.

3. Interval Arithmetic

In the most basic version of our library, we wish to quantify the roundoff errors committed at each step of a floating-point calculation and their propagation. The natural way to do this is to keep for each value an interval that is guaranteed to contain the value that would have been computed in the real-number semantics. Using the standard interval arithmetic [30] we can then follow the computation trace and at each step to calculate the floating-point value and the associated interval error bound.

The propagation of errors is then carried out in interval arithmetic. The roundoff errors at each step can be determined from the specification of the IEEE 754 floating-point standard [37], which all recent hardware conforms to and which is also 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 [25]. 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 floating-point result $fl()$ is given by

$$fl(x \circ y) = (x \circ y)(1 + \delta) \qquad |\delta| \leq \epsilon_m, \quad \circ \in \{+, -, *, /\} \quad (1)$$

ϵ_m is the machine epsilon and determines the upper bound on the relative error of a floating-point computation. Then the rigorous bounding interval for each basic operation is computed as

$$fl(x \circ y) = [(x \circ y) - \epsilon_m(x \circ y), (x \circ y) + \epsilon_m(x \circ y)]$$

where outwards rounding retains rigorousness. The error for square root follows similarly.

The definition of constants requires the following consideration. A single value, say 0.1, is represented in a real valued interval semantics as the point interval $[0.1, 0.1]$. This no longer holds for floating-point values that cannot be represented exactly in the underlying binary representation. The library tests each value for whether it can be represented or not. If not, it computes the interval bound using outwards rounding. In this way the over-approximation is limited by only increasing the interval bounds where necessary. The library has the exact values at runtime, so that the error bounds computed will be as tight as the IEEE standard allows it. Our library can thus generally compute tighter bounds compared to a static analysis-based approach.

Thanks to dedicated floating-point units in most hardware, floating-point computations are fast, so that our library uses double precision floating-point values only (i.e. $\epsilon_m = 2^{-53}$). This is also the precision of choice for most numerical algorithms, but it is straight-forward to adapt the error computations for single precision, or any other precision with a corresponding semantics (including, for example, custom representations using a small number of bytes, as used in certain embedded applications). The outwards rounding on the interval bounds can be achieved easily by using the Java library function nextUp (added in Java 1.6 and accessible from Scala). The symmetric equivalent nextDown follows as $-$nextUp$(-x)$ since negation does not incur any rounding errors.

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.

The library utilizes the 'soft' policy advocated in [10], whereby slight domain violations for some of the functions 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 the inherent over-approximation.

The calculation of nonlinear library functions log, exp, cos, ... requires specialized rounding, since these are correct to 1 ulp (unit in the last) only, and hence less accurate than the elementary arith-

```
def springFnc(k: SmartFloat, m: SmartFloat, xmax: SmartFloat) =
  (i: Int) => {
    var t = SmartFloat(0.0)
    val dt = 1.0/(math.pow(10, (i+1))) //0.1, 0.01, 0.001 etc.
    var x: SmartFloat = xmax
    var v = SmartFloat(0.0)
    var a = − (k/m) ∗ x
    val errX = 0.5∗dt∗dt∗(k/m)∗xmax
    val errV = 0.5∗dt∗dt∗(k/m)∗sqrt(k/m)∗xmax

    while ( t <= 1.0 ) {
      x = addNoise(x + dt∗v, errX)
      v = addNoise(v + dt∗a, errV)
      a = − (k/m) ∗ x
      t = t + dt
    }
    x
}
```

Figure 6. Function modeling the behavior of a spring.

```
{
  val spring1 = new SmartFloat(springFnc(6.0, 1.0, 12.0), 3)
  val spring1 = new SmartFloat(springFnc(1.0, 1.0, 1.0), 3)

  var volume = (2.3 ∗ spring1 + 1.3 ∗ spring2) + 5
  if(−15.60 < volume && volume < −15.50)
    println("take some action")
}

{
  val spring1 = new SmartFloat(springFnc(6.0, 1.0, 12.0), 0)
  val spring2 = new SmartFloat(springFnc(1.0, 1.0, 1.0), 0)

  var volume = (2.3 ∗ spring1 + 1.3 ∗ spring2) + 5
  if(−15.60 < volume && volume < −15.50)
    println("take some action")
}
```

Figure 7. Example usage of the two-spring example.

metic operations, which are correct to within 1/2 ulp. The directed rounding procedure is thus adapted in this case to produce larger interval bounds.

3.1 Why intervals can fail

This would be the end of the story, if intervals did not have the unfortunate disadvantage of ignoring correlations between variables and thus often over-approximating the errors far too much. For example, $x - x =$ returns 0 in interval arithmetic only if x is a point interval, otherwise the interval width is doubled. To illustrate this effect on a real-world example, consider the following piece of code that uses Halley's method [36] to compute the cube root of 10. After 5 iterations we notice that the result starts to oscillate in the last significant digit, so we stop.

```
val a: SmartFloat = 10
var xn = SmartFloat(1.6)

for(i ← 1 until 5) {
    xn = xn ∗ ((xn∗xn∗xn + 2.0∗a)/(2.0∗xn∗xn∗xn + a))
}
println(xn.toStringInterval + " " + xn.interval)
```

The result returned by interval arithmetic is the following

```
> 2.1544346900318834 (5.689131468487283E-14)
    [2.1544346900317617,2.154434690032006]
```

where the numbers in brackets show the computed interval. This result suggests that only 12 significant digits are correct. However, comparing the computed result to the result computed to an accuracy of 30 digits in Mathematica [38], we can see that only the last significant digit is wrong:

$$2.154434690031883721...$$

In the next section, we show that fortunately, there is a way to get around this problem and how to recover some of the accuracy. Indeed, if we use *affine arithmetic* on the same computation instead of intervals to track the errors, we obtain the following result:

```
> 2.1544346900318834 (8.245118070271899E-16)
    [2.1544346900318816, 2.154434690031885]
```

from which we can tell that the digits 2.15443469003188 are definitely correct. We gave gained 3 correct significant digits!

4. Affine Arithmetic

Affine arithmetic was introduced in [10] and addresses the problem of correlations between variables by representing 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 $\epsilon_i \in [-1, 1]$, called noise symbol, is a formal variable denoting a deviation from the center. The maximum magnitude of each deviation is given by the corresponding x_i. The sign of x_i does not matter in isolation, however it reflects the relative dependence between values. For example, take $x = x_0 + x_1\epsilon_1$, then

$$x - x = x_0 + x_1\epsilon_1 - (x_0 + x_1\epsilon_1) = x_0 - x_0 + x_1\epsilon_1 - x_1\epsilon_1 = 0$$

in real number semantics. In floating-point semantics the result is not entirely zero due to roundoff errors.

The interval represented by an affine form is computed as

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

Thus, by internally replacing intervals to track roundoff errors by affine forms and converting them in the end into intervals, the library keeps the resulting error bounds much tighter and is thus much more useful.

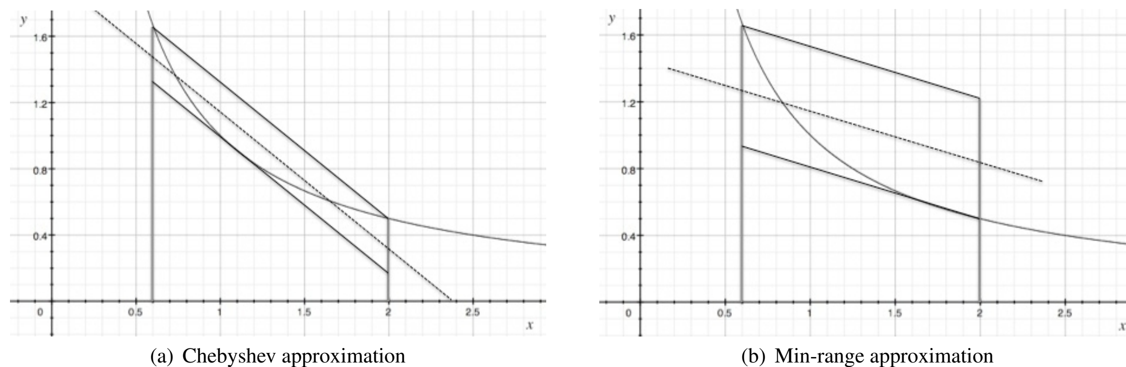
4.1 Sources of errors

Our system automatically introduces the roundoff errors arising from the declaration of constants and arithmetic operations. Additionally, the developer may specify further attached errors. As illustrated in the examples in Figure 5 and Figure 6, this facility is very useful, because it allows the library to propagate domain-specific errors. One source of these errors are measurement inaccuracies in systems that interact with the physical world through sensors. Another source are method errors, i.e., the differences between a (hypothetical) analytical solution and a result computed with an iterative method.

4.2 Nonlinear operations

The calculation of linear operations is straightforward:

$$\alpha\hat{x} + \beta\hat{y} + \zeta = (\alpha x_0 + \beta y_0 + \zeta) + \sum_{i=1}^{n} (\alpha x_i + \beta y_i)\epsilon_i \qquad (2)$$



(a) Chebyshev approximation

(b) Min-range approximation

Figure 9. Approximations of the inverse function.

and is apart from roundoff errors exact (see subsection 4.4), but nonlinear operations must be approximated. Multiplication is derived from multiplying the two affine forms:

$$\hat{x}\hat{y} = x_0y_0+\sum_{i=1}^{n}(x_0y_i+y_0x_i)\epsilon_i+(\sum_{i=1}^{n}|x_iy_i|+\sum_{i<j}|x_iy_jx_jy_i|)\epsilon_{n+1}$$

The first two terms are linear and thus exact. The last term over-approximates the nonlinear contribution and is added as the coefficient of a new 'fresh' (not yet used) noise symbol. In general, the nonlinear contribution is smaller than the linear part or the roundoff errors, so that the library does not use a more accurate approximation for performance reasons.

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

In [10] two approximations are suggested: a Chebyshev (min-max) or a min-range approximation. These two are illustrated in Figure 9. For both, the approximation is computed by converting the affine form into an interval and working with its endpoints. For the min-range approximation it suffices to compute the slope α at one of the endpoints and from this to compute the maximum deviation ζ from the (dashed) middle line. For the Chebyshev approximation a third point is needed and the slope is calculated at this point. The Chebyshev version indeed generally computes a tighter approximation (the area of the bounding parallelogram is smaller), on the other hand though, requires more and more complex calculations, especially for the trigonometric functions. Every computation step carries with it a roundoff error, which also has to be added to the resulting affine form, so that the more calculations are needed, the more the errors accumulate. Especially for small initial errors on the order of machine epsilon, the Chebyshev approximation sometimes computes the third (middle) point *outside* of the initial interval (this was not considered in [10]). For this reason, the library uses the min-range approximation, as it was found to work reliably. Division is decomposed into an inverse function application and a multiplication and the power function is computed as $e^{y \log x}$. Hence, by a careful choice of the approximation, we avoid using an expensive computation with, for example, an arbitrary precision library, while still computing accurate bounds.

4.3 Combining Intervals and Affine Forms

The mentioned computational inaccuracies for very small errors many times also cause the calculated δ to be (unnecessarily) too large. To limit this over-approximation, the library uses interval arithmetic to provide a maximum bound with which the error δ introduced in the affine approximation can be limited. Note that this works so neatly here because the geometric interpretation of the approximation of unary functions is an interval on the y-axis. While the library computes the same error bound in interval and affine arithmetic with this procedure, it also retains the correlation information, which helps to compute smaller bounds in the following calculations. For variables with bigger attached errors, the affine arithmetic's advantages prevail and no special treatment is needed.

4.4 Dealing with roundoffs

Each operation carries a roundoff error and all of them must be taken into account to achieve truly rigorous bounds. For each arithmetic operation or library function application the library collects the absolute values of all roundoff and approximation errors and adds them to the resulting affine form with a new 'fresh' noise symbol. The sign here is of no interest, since no correlations exist between this new error and any other variable.

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. $\alpha * x_0 + \zeta$), as the precise roundoff errors depend on the exact values of the variables. These errors can be determined by the following procedure [10]:

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

where $\downarrow$ denotes rounded towards negative infinity and $\uparrow$ denotes towards positive infinity. 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 symbol is added.

This particular preciseness makes the exact analysis in the example in Figure 1 feasible, i.e. the library can certify that the second calculation is indeed exact.

4.5 Bounding the number of noise symbols for performance

The performance of affine forms is approximately proportional to the number of noise symbols used. For short calculations, this is of no consequence, since every operation adds at most one noise symbol, however longer or iterative ones can accumulate so many noise terms that it makes a computation infeasible. Figure 10(a) shows this relationship for selected computations: $\sum \frac{1}{n^{-2}}$, an iterated computation using basic arithmetic, and the spring function from Figure 6 once with $dt = 0.1$ and once with $dt = 0.01$. The library limits the number of noise symbols an affine form can accumulate. Once this number is reached, all error terms with an absolute value smaller than the average are replaced by a new noise term, with magnitude equal to their sum (rounded towards infinity). The number of noise symbols also affects the accuracy, since correlations are lost in the process of replacement. This is illustrated by the graph in Figure 10(b) for the example of the spring with $dt = 0.1$, which shows the correlation between the number of noise symbols and the resulting relative error. We decided experimentally on a default limit of around 40 as a good compromise between performance and accuracy, but the developer may change this value for particular calculations.[1]

4.6 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.

This also implies that we can make statements about the robustness of a piece of code, i.e. we can show that the control flow is invariant to small perturbations in the input. Comparisons are performed with error bounds taken into account so that if the truth value of a condition cannot be unambiguously determined, a (sticky) global flag is set (subsection 6.2). Thus, if a computation can be performed for a value x with some given error bound without setting of this flag, it implies that the code is robust for this value within small perturbations (with the maximum size of the initial error).

5. Integration into a Programming Language

This section explains how the error tracking is integrated into Scala in a seamless way. Our library provides a wrapper type SmartFloat 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 typechecker. 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 [31] , 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 (see subsection 6.2 for details). Fortunately, Scala also provides the trait (similar to a Java [14] 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 [33].

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 [31]. 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

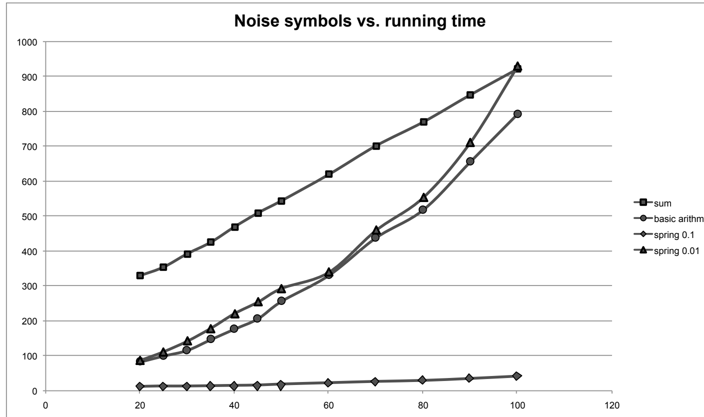
```
val x: SmartFloat = 0.5
val y = sin(x) * Pi
```

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.

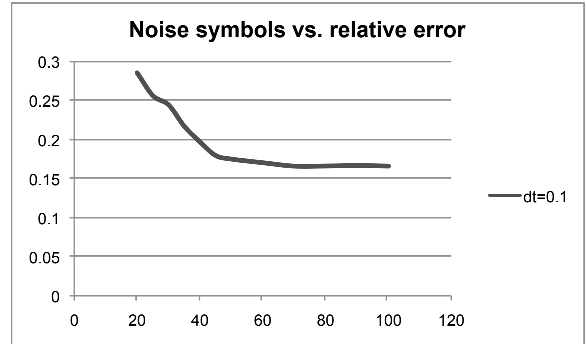
6. Tracking and Transforming Expressions

The library described so far is a fully functioning replacement for the standard Double data type. Developers can use this library to track rounding errors and to obtain rigorous answers. However, using affine arithmetic to track all the errors at each arithmetic operation can be time consuming and can slow down the calculation. Many times though, the added accuracy of affine arithmetic may not even be needed, for example when a comparison only needs a few correct significant digits. Therefore, we chose a lazy approach

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



(a) Effect on running time.



(b) Effect on accuracy for the spring example, taking into account the method error.

Figure 10. The effect of the number of noise symbols.

to implement the underlying operations of interval and affine arithmetic: initially, expressions are not evaluated, but the computation is stored in form of an expression abstract syntax tree (AST) and is evaluated on request. In our system, the evaluation is triggered only at the following points in program execution:

- call to the `toString` method
- at comparisons $<, ==, >$
- at conversions to other numeric types
- on explicit user request
- when expressions become too large

The expression trees are immutable, thus preserving the ability of viewing a SmartFloat as a numerical data type. This also allows the library to internally use sharing of expressions and reference equality.

6.1 Compacting large expressions

For performance and stack size reasons, our library compacts expressions as soon as they reach a certain size (each variable and each operation adds 1 to the overall expression size). Each expression stores its size. As soon as this size reaches a limit, our library replaces subtrees by leaves that store their values, which are computed using affine and interval arithmetic. Because this preserves the evaluation order, the computed results and error bounds remain the same. The maximum expression size is in the control of the user, but we have found experimentally that a maximum size on the order of 1000 is reasonable.

6.2 Comparisons

If a variable participates in a comparison (see Figure 5 for a concrete example), it necessarily needs to be evaluated. The evaluation is initially performed only in the faster interval arithmetic. Only if the result of the comparison still cannot be determined, will the expression be evaluated also in affine arithmetic. If this test fails as well, that is, given the attached errors, the library cannot conclusively determine whether the condition is true or false, the actual Double value of the expression is used to determine the truth value of the condition (the control flow is not altered) and a global (sticky) flag is set. Again, this feature integrates nicely into Scala as the lazy evaluation can be implemented by using the built-in `lazy` keyword .

```
def getRewrites(node: Expr): Set[Expr]
    if node is of form op(e1, e2)
        exprs = { op(r1, r2) | r1 ∈ getRewrites(e1),
                                r2 ∈ getRewrites(e2)}
    else if node is of form op(e1)
        exprs = { op(r1) | r1 ∈ getRewrites(e1)}

    return top(k, {applyRules(e) | e ∈ exprs})
```

Figure 11. Rewriting algorithm.

6.3 Rule-Based Rewriting

We have demonstrated in Figure 4, that two expressions that are mathematically equivalent can produce different floating-point results. Our library can use the collected computation trace and rewrite an expression to find a formulation that produces a more accurate result or a tighter error bound for the same input values.

A prerequisite for this is the assumption that the computation is meant in real semantics, not floating-point, i.e. all mathematically equivalent rewritings of the expression are permitted. The algorithm is given in Figure 11. The expressions are sorted first by accuracy, then by size, so that the head of the resulting set holds the most accurate expression that can be generated. Syntactically equivalent expressions are automatically discarded.

Depending on the number of rules applicable, the number of expressions can grow exponentially. To limit the number of expressions at each step and also the complexity of the overall algorithm, only the locally k best expressions are propagated. The value k is user-controlled, but $k = 25$ has been experimentally determined as a suitable limit.

This procedure does not generate all possible rewritings of equations, since it only moves up through the tree and does not backtrack. This is however, powerful enough to reformulate the expression in the triangle example Figure 4 or to rewrite the formula for the redshift parameter z

$$\sqrt{\frac{1+\frac{v}{c}}{1-\frac{v}{c}}} - 1 \quad \text{into} \quad \frac{\sqrt{v+c}-\sqrt{c-v}}{\sqrt{c-v}}$$

An important part of this rewriting is the simplification of the expressions prior to the application of any rules. For this to be possible, a distinction needs to be made between variables and constants, so that one can simplify e.g. `(3 * x *x * 4 *y) / (x *`

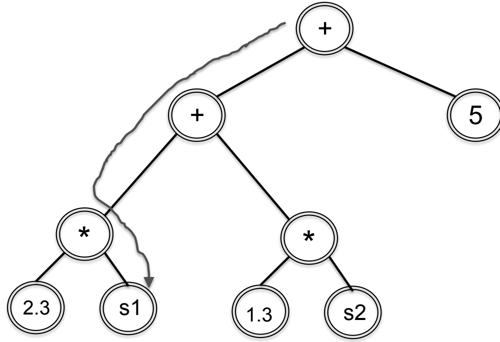


Figure 12. Demand-driven iteration: only those branches with the largest relative error are followed and iterated. The variables `s1` and `s2` depict the spring1 and spring2 variables from the example in Figure 7 respectively.

`2 * y * y)` into `(6 * x ) / y`. The following convention implements this difference

```
val x = SmartFloat(3.5, "x") // variable
val c: SmartFloat = 3.14 // constant
```

The first line utilizes the `apply` method and the second an implicit conversion so that the definitions of variables and constants can be distinguished. In addition, variables can be named for a more readable output. In many cases it is possible to simply copy-paste the suggested new expression into the source code and use in future executions.

We have implemented this functionality for the operations $+, -, *, /, \sqrt{}$ with a selected set of rules. Thanks to Scala's pattern matching capability it is easy to add new ones, since rules are of the forms such as

```
case Sqrt(Mult(x, y)) => Mult(Sqrt(x), Sqrt(y))
case Sqrt(Div(x, y)) => Div(Sqrt(x), Sqrt(y))
```

and thus closely resemble their mathematical formulation.

If the system finds a more accurate formulation of an expression, it only prints the suggestion, i.e., it does not reuse it directly. The reason for this is that this rewriting functionality is still too slow at this point in time. However, it can be used as a preprocessing step to improve selected expressions in a code in perhaps non-obvious ways and by doing so increase the understanding of the floating-point behavior.

7. Demand-Driven Computation with Method Error Guarantees

Method errors can perhaps be an even bigger concern in for example physics simulations in that they may outweigh the roundoff errors accumulated during a calculation. Method errors can be added manually by the user at specific steps and will be propagated together with all other errors to yield a total error estimate. It would be however even more useful, if certain computations could be made more accurate on demand also with respect to the method error. Consider the example in Figure 6. The method error in this simulation is determined by the length of the time step `dt`; the smaller this is, the more accurate will the resulting position of the end of the spring be. This suggests an iterative refinement of the accuracy, where at each iteration the library chooses a smaller time step `dt`, but at a higher cost, since each iteration takes considerably longer.

With our library, the user defines such a function, which takes as parameter an integer which determines the current accuracy and

```
def refine(node: Expr): Expr
    if node is of form Iterative
        return iterate(node)

    else if node is of form Op(e1, e2)
        err1 = e1.relativeError
        err2 = e2.relativeError
        if refine(e1).relativeError < err1
            return Op(refine(e1), e2)
        else
            return Op(e1, refine(e2))

    else if node is of form Op(e1)
        return Op(refine(e1))
```

Figure 13. Iteration algorithm.

returns the desired result as a SmartFloat. This function is passed to a special SmartFloat and can be part of any expressions as any other. Once a comparison is reached however, the value of this SmartFloat is iteratively refined until either a precision which makes it possible to decide the condition is reached, or until a maximum refinement can be detected.

The demand-driven refinement goes even further. One expression can potentially contain several iteratively defined variables, of which only some need refinement. This is the case in Figure 7 and is illustrated in Figure 12: the pictured tree is the expression tree from the variable volume. The library can again reuse the expression abstract tree that was collected and preferentially traverse the tree down branches with larger relative errors and refine those first. The algorithm is given in Figure 13. In this way, expressions involving iterative computations can be written in a very natural way, and computed only as accurately as needed. This potentially saves much time, in our spring example the second computation takes only half the time to compute.

7.1 Fixpoint computations

Our library supports demand-driven iterations also for fixed-point computations, where the next (more accurate) step is computed as

$$x_{n+1} = \phi(x_n)$$

In this case the user has to provide some more information, namely the iteration function ϕ, a function $err(e_n, x_n)$ which computes the method error e_{n+1} at each step, as well as the starting point and initial error. For an illustration of how this looks like, consider the function

$$f(x) = e^x - 4x^2$$

for which we want to compute the root between 4.1 and 5 (this initial interval can be determined by checking the sign of the function at both endpoints). A possible iteration function (illustrated in Figure 14) is $\phi(x) = 2\ln(2x)$ and a rigorous estimate of the method error is given by the maximum value of the derivative ϕ' inside of our search interval. Hence the code looks like

```
val fnc = (x: SmartFloat) => 2.0 * log(2.0 * x)
val err = (prevErr: SmartFloat, xn: SmartFloat)
    => 0.488*prevErr
val x = new SmartFloat(fnc, err, 4.1, 0.4, 1)
```

This iteration converges slowly, yielding only 7 significant digits after 20 iterations. Hence an iteration on demand can prove useful, especially if a high accuracy is not needed, or the needed accuracy is not known in advance.

To avoid infinite iterations, our library defines the following termination criteria: either a user-specified maximum number of iterations is reached, or the iteration is stopped when the relative

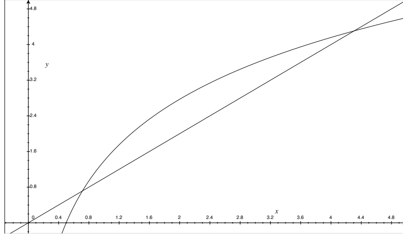


Figure 14. Fixed-points $x = 2\ln(2x)$.

	pure double	double	interval	affine
triangle	0	2	15	270
trigonometry	1	3	18	103
basic arithmetic	0	2	12	112
$\sum \frac{1}{n^{-2}}$	2	72	723	2850
physics simulation	$1 \cdot 10^4$	-	-	$7 \cdot 10^4$

Figure 15. Comparison of evaluation times in ms, for 1000 runs of each example. (This corresponds to 100s of simulated time in the physics simulation.)

error does not change or becomes larger from one iteration to the next.

8. Experience

All examples, as well as some 100 microbenchmarks were tried with our library to test its correctness and the accuracy of the approximations. We ran some of our results against results calculated by Mathematica [38] to a high precision and were satisfied with the rigorousness, but also the reasonably tight bounds. [2] We have not used the common floating-point benchmarks, because they are designed for performance and not accuracy in mind, so they do not fit our purpose.

8.1 Performance when Using the Library

Although the primary goal of our library is to increase the understanding of numerical calculations, the question of how fast it is compared to the original code is a valid one. Figure 15 gives an idea of the cost of accuracy. For the analysis, we chose the triangle example from Figure 4, some simple calculations involving trigonometric functions and basic arithmetic, the summing example we have used previously and one of our physics simulations (described in the next subsection). The first columns indicates the times for the original code, using only the standard Scala Double, columns one to three show the times when the expression trace was evaluated with Doubles, intervals or affine forms only.
It is apparent that affine arithmetic is expensive, so that our lazy approach saves considerable amounts of time, when applicable, but it is also clear, that meaningful answers about the accuracy of the developer's code are provided in reasonable time.

8.2 Usability and usage scenarios

In our envisioned usage scenario, the developer executes a numerical computation for a bounded amount of time, to obtain insight about any potential sources of inaccuracy. Using this information, along with suggested rewrites of parts of the code, the developer can improve the original code. It is thus not necessary to use our library in the final deployment of the program; what remains are the insights gained from library execution.

[2] The examples are available on request from the program chair.

We tested the usability of our library on a set of simple physics simulations, notably a real-scale simulation of the Earth circling the Sun (albeit with simplified physics and methods errors). We wrote the code first in standard Scala syntax with Double variables. We found that changing from the Scala's Double to our SmartFloat requires little work; only a few pieces of code required short manual review. When running the simulation we found that it runs around seven times slower than implementation using the Double floating points (see the last line in Figure 15). This was practical enough for our purpose; we were able to understand, for example, that the method error dominates over the roundoff errors, providing the insight of the kind that we hope to obtain when using our library.

9. Related Work

Estimating Roundoff Errors To our knowledge, this is the first system to keep track of roundoff errors for calculations involving operations other than $\{+, -, *, /, \sqrt{}\}$. The idea of expression rewriting has been explored by [28] in the context of abstract interpretation. However, the approach only works for $\{+, -, *\}$.

The Fluctuat tool [15] uses abstract interpretation to reason about numerical programs. The abstract domain developed tracks the roundoff errors committed at each program with the use of affine arithmetic, however the static approach limits the accuracy of the computed bounds. The domain works for the operations $\{+, -, *, /, \sqrt{}\}$, but the details of the last two operations are not provided. Other work in the abstract interpretation direction includes the Astrée analyzer [9] that also provides abstract domains which work correctly with floating-point numbers. Another abstract interpretation library is APRON [20], also with a floating-point abstract domain [8].

A recent approach [19] to statically detect loss of precision in floating-point computations uses bounded model checking based on SMT solvers. It uses interval arithmetic for scalability reasons. It is supposed to detect 'stable' computations, in the sense of small relative errors. [11] used affine arithmetic to track roundoff errors by means of a C library, however their work seems to be specific for the signal processing domain.

Other possible approaches to quantify roundoff errors in floating-point computations are summarized in [27]. This also includes stochastic estimations of the error, which has been implemented in the CADNA library [21]. However, this approach does not provide rigorous bounds. Similarly, specialized algorithms exist for dealing with floating-point inaccuracies in, for example, summation [34] or dot products [32].

Affine arithmetic is being used in some specific application domains to deal with uncertainties, as for example in signal processing [16], however our library is developed for general purpose calculations as well as user-defined additional errors.

[39] shows how to limit the over-approximations for non-linear operations in affine arithmetic even further. The resulting approach is more precise, so it remains to investigate whether it is beneficial and feasible to integrate it into a run-time library.

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 bit set upon the unresolved comparisons. Therefore, we believe that our library can be useful for understanding program robustness and continuity properties [7, 26].

Interactive approaches and decision procedures. Researchers have used theorem proving to verify floating-point programs [3, 4, 17, 29, 35]. 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. [24] extend the previous work by considering the problem of reducing pre-

cision for performance reasons, and uses affine arithmetic as well. [5] presents a decision procedure for checking satisfiability of a floating-point formula by encoding every operation by a propositional formula, however, they are forced to use approximations because of the complexity of the resulting formulas. There is a number of general-purpose approaches for reasoning about formulas in non-linear arithmetic, see e.g. [2]. 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.

Lazy computation. We are not aware of other approaches that apply demand-driven refinement of results in the presence of affine arithmetic. The general idea is inspired by the well known notion of lazy evaluation which has been used in non-strict functional languages [22] as well as non-deterministic languages [12]. The idea is also related to the concept of functional self-adjusting computation [1], but has so far not been applied to the domain of affine arithmetic with user-defined expression equivalence.

Language virtualization. Language virtualization [6] has been proposed as a general method to embed domain specific languages into a flexible host programming language. Our approach uses some of the similar mechanisms to construct syntactic representations of the parts of the executing program, but performs the computations at run time (possibly after some delay).

10. Conclusions

We have presented an easy-to-use library solution for helping developers understand the accuracy of their numerical computation. Our system uses affine arithmetic to estimate errors and their correlations, it suggests alternative expressions with equivalent semantics with respect to given rewrite rules, and it supports demand-driven refinement of precision-adjustable computations. We have used our system in a number of benchmarks, including a simulation of gravitational interactions of physical bodies. We have found the slowdown of the libraries to be acceptable for understanding the sources and propagations of errors in numerical computations. We have also found our system to be useful in suggesting replacements of code fragments to reduce the roundoff errors. Finally, the system was helpful in identifying the amount of iterations needed in numerical computations, which allows us to produce faster code while providing sufficient accuracy for the application at hand.

References

[1] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. *ACM Trans. Program. Lang. Syst.*, 28(6), 2006.

[2] Behzad Akbarpour and Lawrence Charles Paulson. MetiTarski: An Automatic Theorem Prover for Real-Valued Special Functions. *J. Autom. Reason.*, 44(3):175–205, 2010.

[3] Ali Ayad and Claude Marché. Multi-Prover Verification of Floating-Point Programs. In *IJCAR*, 2010.

[4] Sylvie Boldo, Jean-Christophe Filliâtre, and Guillaume Melquiond. Combining Coq and Gappa for Certifying Floating-Point Programs. In *CICM '09*, pages 59–74, 2009.

[5] Angelo Brillout, Daniel Kroening, and Thomas Wahl. Mixed Abstractions for Floating-Point Arithmetic. In *FMCAD*, pages 69–76, 2009.

[6] Hassan Chafi, Zach DeVito, Adriaan Moors, Tiark Rompf, Arvind K. Sujeeth, Pat Hanrahan, Martin Odersky, and Kunle Olukotun. Language virtualization for heterogeneous parallel computing. In *OOPSLA*, 2010.

[7] Swarat Chaudhuri, Sumit Gulwani, and Roberto Lublinerman. Continuity analysis of programs. In *POPL*, 2010.

[8] L. Chen, A. Miné, J. Wang, and P. Cousot. A Sound Floating-Point Polyhedra Abstract Domain. In *APLAS*. LNCS 5356, Springer, 2008.

[9] Patrick Cousot, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTRÉE Analyser. In *ESOP'05*, Lecture Notes in Computer Science, pages 21–30. Springer, 2005.

[10] Luiz H. de Figueiredo and Jorge Stolfi. *Self-Validated Numerical Methods and Applications*. Brazilian Mathematics Colloquium monographs. IMPA/CNPq, Rio de Janeiro, Brazil, 1997.

[11] C.F. Fang, Tsuhan Chen, and R.A. Rutenbar. Floating-point error analysis based on affine arithmetic. In *ICASSP*, volume 2, pages II – 561–4 vol.2, 2003.

[12] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. Test Generation through Programming in UDITA. In *ICSE*, 2010.

[13] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, 1991.

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

[15] Eric Goubault and Sylvie Putot. Static Analysis of Numerical Algorithms. In *SAS*, pages 18–34, 2006.

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

[17] John Harrison. Formal verification at Intel. In *LICS*, 2003.

[18] Les Hatton and Andy Roberts. How Accurate is Scientific Software? *IEEE Trans. Softw. Eng.*, 20:785–797, 1994.

[19] Franjo Ivancic, Malay K. Ganai, Sriram Sankaranarayanan, and Aarti Gupta. Numerical stability analysis of floating-point computations using software model checking. In *MEMOCODE*, 2010.

[20] Bertrand Jeannet and Antoine Miné. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *CAV*, pages 661–667, 2009.

[21] Fabienne Jézéquel and Jean-Marie Chesneaux. CADNA: a library for estimating round-off error propagation. *Computer Physics Communications*, 178(12):933 – 955, 2008.

[22] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

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

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

[25] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

[26] Rupak Majumdar and Indranil Saha. Symbolic robustness analysis. In *IEEE Real-Time Systems Symposium*, 2009.

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

[28] Matthieu Martel. Program transformation for numerical precision. In *PEPM*, 2009.

[29] J. Strother Moore, Thomas W. Lynch, and Matt Kaufmann. A mechanically checked proof of the AMD5 K86 floating point division program. *IEEE Trans. Computers*, 47(9), 1998.

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

[31] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: a comprehensive step-by-step guide*. Artima Press, 2008.

[32] Takeshi Ogita, Siegfried M. Rump, and Shin'ichi Oishi. Accurate Sum and Dot Product. *SIAM J. Sci. Comput.*, 26(6):1955–1988, 2005.

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

[34] Siegfried M. Rump, Takeshi Ogita, and Shin'ichi Oishi. Accurate Floating-Point Summation Part i: Faithful Rounding. *SIAM J. Sci. Comput.*, 31(1):189–224, 2008.

[35] David M. Russinoff. A mechanically checked proof of correctness of the AMD K5 floating point square root microcode. *Formal Methods in System Design*, 14(1), 1999.

[36] T. R. Scavo and J. B. Thoo. On the Geometry of Halley's Method. *The American Mathematical Monthly*, 102(5):pp. 417–426, 1995.

[37] IEEE Computer Society. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1 –58, aug. 2008.

[38] Inc. Wolfram Research. *Mathematica Edition: Version 7*. Wolfram Reserach, Inc., 2008.

[39] Linsheng Zhang, Yan Zhang, and Wenbiao Zhou. Tradeoff between Approximation Accuracy and Complexity for Range Analysis using Affine Arithmetic. *Journal of Signal Processing Systems*, 61:279–291, 2010.