Certifying Solutions for Numerical Constraints

Eva Darulova and Viktor Kuncak [*]

EPFL
`{eva.darulova,viktor.kuncak}@epfl.ch`

Abstract. A large portion of software is used for numerical computation in mathematics, physics and engineering. Among the aspects that make verification in this domain difficult is the need to quantify numerical errors, such as roundoff errors and errors due to the use of approximate numerical methods. Much of numerical software uses self-stabilizing iterative algorithms, for example, to find solutions of nonlinear equations. To support such algorithms, we present a runtime verification technique that checks, given a nonlinear equation and a tentative solution, whether this value is indeed a solution to within a specified precision.
Our technique combines runtime verification approaches with information about the analytical equation being solved. It is independent of the algorithm used for finding the solution and is therefore applicable to a wide range of problems. We have implemented our technique for the Scala programming language using our affine arithmetic library and the macro facility of Scala 2.10.

Keywords: solution verification, numerical computation, error estimation, affine arithmetic

1 Introduction

Software manipulating numerical quantities has numerous applications in decision making, science, and technology. Such software is difficult to validate by any method—manual inspection, testing, or static analysis. One of the core challenges in each case is the gap between the approximate nature of numerical computations and the idealized mathematical models that form their foundation and specification. Specialized programming languages inside commercial computer algebra systems aim to simplify working with numerical computations. However, their precision and soundness guarantees compared to the mathematical meaning are not well documented, and many of the implementations are closed source. Much of the real-world computation is done in general-purpose languages, supported by many numerical software libraries written for them. The work on this paper builds on open-source general-purpose infrastructures, providing a next step in validated numerical computation for Scala [14].

Existing validation of numerical computations supports estimation of roundoff errors [7,1]; we have previously incorporated computation of roundoff errors

---

in Scala using affine arithmetic [5]. Going a step further, we present automated estimation of not only roundoff errors, but also *method errors*, which arise, for example, when using numerical methods to iteratively solve equations. Such methods are used to solve equations that have no symbolic closed-form solution, which is often the case in practice. Even if symbolic solutions exist, iterative approaches can be faster or better-behaved with respect to roundoff errors.

1.1 Contributions

To understand the notion of method errors we address, consider an iterative method that performs a search for the solution of $f(x) = 0$ by computing a sequence of approximations $x_0, x_1, x_2, \ldots$ One common stopping criterion for an iteration is finding x_k for which $|f(x_k)| < \varepsilon$, for a given error tolerance ε. From a validation point of view, however, we are ultimately interested not in ε but in τ such that $|x - x_k| < \tau$, where x is the actual solution in real numbers. Fortunately, we can estimate τ from ε using a bound on the derivative of f in an interval conservatively enclosing x and x_k.

A tempting approach is to perform the entire computation of x_k using interval [12] or affine arithmetic. However, this approach would be inefficient, and would give too pessimistic error bounds. Instead, our method uses a runtime checking approach. We allow any standard non-validated floating point code to compute the approximation x_k. We perform only the final validation of an individual candidate solution x_k using a range-based computation. In this way we achieve efficiency and reusability of existing numerical routines, while still providing rigorous bounds on the total error. The bounds certified by our system are always sound for the given execution.

Our system thus realizes a new kind of assertion, appropriate for numerical computation: an assertion that verifies "this was precise enough" in a way that takes into account both the numerical algorithm and floating point semantics.

To perform such sound computation, our approach uses static information about the function and computes derivatives at compile time. For this purpose it uses the macro facility of Scala, our implementation of symbolic differentiation, and a method to compute bounds of a function over an interval. A technical challenge that arises in rigorously estimating the error is that mean value theorems (the foundation for error estimation), refer to an arbitrary point between the approximate and the unknown exact solution. It is therefore not clear over which interval one needs to estimate the error. We solve this circularity through a simple design, which expects a bound on the argument error as the input, and verifies whether this bound indeed holds. This allows us to perform an estimation using very narrow intervals, contributing to the precision of our approach.

We integrated our method into the Scala programming language (Section 4). We demonstrate its applicability and usefulness on a number of examples (sections 2 and 5). Among the consequences of this development is a Scala framework that can check runtime assertions in a way consistent with mathematical reals, while executing on the standard virtual machine, soundly taking into account the concrete semantics of floating point operations and iterative numerical methods.

2 Examples

We motivate our contribution with examples that model physical processes, taken from [18,4,15]. These examples illustrate the applicability of our techniques and introduce the main features of our library. For space reasons we abbreviate the Scala Double type with D (the code snippets remain valid Scala code using the rename-on-import Scala feature). We include variable type declarations for expository purposes, even though the Scala compiler can infer all but the function parameter types. A function that maps x into $e(x)$ is denoted in Scala by $x \Rightarrow e(x)$. Method names printed in bold (e.g., **jacobian**, **assertBound**) are parts of the public interface of our library for certifying solutions of numerical computations.

Stress on a Turbine Rotor. We illustrate the basic features of our library on the following system of three non-linear equations with three unknowns (v, ω, r). An engineer may need to solve such a system to compute the stress on a turbine rotor [18].

$$3 + \frac{2}{r^2} - \frac{1}{8}\frac{(3-2v)}{1-v}\omega^2 r^2 = 4.5 \qquad\qquad 6v - \frac{1}{2}\frac{v}{1-v}\omega^2 r^2 = 2.5$$
$$3 - \frac{2}{r^2} - \frac{1}{8}\frac{(1+2v)}{1-v}\omega^2 r^2 = 0.5 \tag{1}$$

Given a numerical routine computeRoot and our library for certifying solutions, the engineer can directly map the above equations into the following code:

val f1 = (v:D,w:D,r:D) $\Rightarrow$ 3 + 2/(r∗r) − 0.125∗(3−2∗v)∗(w∗w∗r∗r)/(1−v)−4.5
val f2 = (v:D,w:D,r:D) $\Rightarrow$ 6∗v − 0.5 ∗ v ∗ (w∗w∗r∗r) / (1−v)−2.5
val f3 = (v:D,w:D,r:D) $\Rightarrow$ 3 − 2/(r∗r) − 0.125∗(1+2∗v)∗(w∗w∗r∗r) / (1−v)−0.5

The engineer can then solve the problem numerically using an off-the-shelf numerical routine that accepts the function and its derivative as an argument:

val x0 = Array(0.75, 0.5, 0.5) *// initial value for iteration*
val roots: Array[D] = computeRoot(Array(f1,f2,f3), **jacobian**(f1,f2,f3), x0, 1e−8)

Finally, the engineer can *certify* the solution using our library:

val errors:Array[Interval] = **assertBound**(f1,f2,f3, roots(0), roots(1), roots(2), 1e−8)

The method **assertBound** takes as input the three functions of our system of equations, the previously computed roots and a tolerance. It returns sound bounds on the true errors on the roots. In the case where these errors are larger than the tolerance specified, the method throws an exception and thus acts like an assertion. Our library also includes the method **jacobian**, which computes the Jacobian matrix of the functions f_1, f_2 and f_3 symbolically at compile time (Section 4.2). The true roots for v, w and r are $0.5, 1.0$ and 1.0 respectively. The roots and maximum absolute errors computed by the above code are

```
0.5, 1.0000000000018743, 0.9999999999970013
2.3684981521893e-15, 1.8806808806556e-12, 3.0005349681420e-12
```

Note that the error bounds computed are, in fact, orders of magnitude smaller than the tolerance $1e-8$ given to the numerical routine and to **assertBound**.
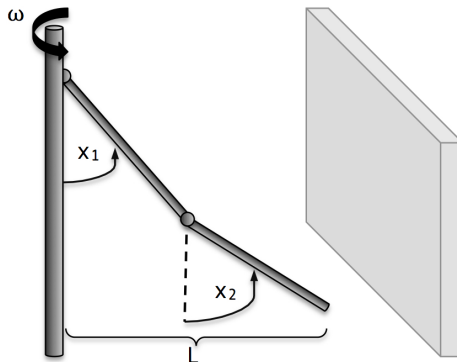
Fig. 1. A double pendulum standing close to an obstacle

Double Pendulum. The following example demonstrates how our library fits into a runtime assertion framework consistent with mathematical reals. A double pendulum rotates with angular velocity ω around a vertical axis, like a centrifugal regulator [4]. At equilibrium, the two pendulums make the angles x_1 and x_2 to the vertical axis. It can be shown that the angles are determined by the equations

$$\tan x_1 - k(2\sin x_1 + \sin x_2) = 0$$
$$\tan x_2 - 2k(\sin x_1 + \sin x_2) = 0 \tag{2}$$

where k depends on ω, the lengths of the rods and gravity. Suppose the pendulum is standing close to a wall (as in Figure 1) and we would like to verify that in the equilibrium position it cannot hit the wall. Also suppose that the distance to the center of the pendulum is given by a function distancePendulumWall. Then the following code fragment verifies that a collision is impossible in the real world, not just in a world with floating-points.

```
val distancePendulumWall : SmartFloat = ...
val length = ... //length of bars
val tolerance = 1e−13; val x0 = Array(0.18, 0.25)
val f1 = (x1: D, x2: D) ⇒ tan(x1) − k ∗ (2∗sin(x1) + sin(x2))
val f2 = (x1: D, x2: D) ⇒ tan(x2) − 2∗k ∗ (sin(x1) + sin(x2))
val r: Array[D] = computeRoot(Array(f1,f2), jacobian(f1,f2), x0, tolerance)
val roots: Array[SmartFloat] = certify(r, errorBound(f1, f2, r(0), r(1), tolerance))

val L: SmartFloat = _sin(roots(0)) ∗ length + _sin(roots(1)) ∗ length
if (certainly(L <= distancePendulumWall)) {
    // continue computation
} else {
    // reduce speed of the pendulum and repeat
}
```

To account for all sources of uncertainty, we use the SmartFloat data type developed previously [5]. SmartFloat performs a floating point computation while

additionally keeping track of different sources of errors, including floating point round-off errors, as well as errors arising from other sources, for example, due to the approximate nature of physical measurements.

In our example, distancePendulumWall and **certify** both return a SmartFloat; the first one captures the uncertainty on a physical quantity, and the second one the method error due to the approximate iterative method. If the comparison in line 9 succeeds, we can be sure the pendulum does not touch the wall. This guarantee takes into account roundoff errors committed during the calculation, as well as the error committed by the computeRoot method and their propagation throughout the computation.

State Equation of a Gas. Values of parameters may only be known within certain bounds but not exactly, for instance if we take inputs from measurements. Our library provides guarantees even in the presence of such uncertainties. Equation 3 below relates the volume V of a gas to the temperature T and the pressure p, given parameters a and b that depend on the specifics of the gas, N the number of molecules in the volume V and k the Boltzman constant [15].

$$[p + a(N/V)^2](V - Nb) = kNT \tag{3}$$

If T and p are given, one can solve the nonlinear Equation 3 to determine the volume occupied by the (very low-pressure) gas. Note however, that this is a cubic equation, for which closed-form solutions are non-trivial, and their approximate computation may incur substantial roundoff errors. Using an iterative method, whose result is verified by our library, is thus preferable:

val T = 300; **val** a = 0.401; **val** b = 42.7e−6;
val p = 3.5e7; **val** k = 1.3806503e−23; **val** x0 = 0.1
val N: Interval = 1000 +/− 5
val f = (V: D) ⇒ (p + a ∗ (N.mid/V) ∗ (N.mid/V)) ∗ (V − N.mid∗b) − k∗N.mid∗T
val V: D = computeRoot(f, **derivative**(f), x0, 1e−9)
val Vcert: SmartFloat = **certify**(V, **assertBound**(f, V, 0.0005))

We make the assumption that we cannot determine the number of molecules N exactly, but we are sure that our number is accurate at least to within ± 5 molecules (line 3). We compute the root as if we knew N exactly, using the middle value of the interval and the standard Newton's method. We only check *a posteriori* that the result is accurate up to $\pm 0.0005 m^3$, for all N in the interval $[995, 1005]$. Our library will confirm this providing us also with the (certified) bounds on V: `[0.0424713, 0.0429287].`

3   Computing the Error

Our verification technique is based on several theorems from the area of validated numerics. It can verify roots of a system of nonlinear equations computed by an arbitrary black-box solution or estimation method.

In the following, we denote computed approximate solutions by $\tilde{x}$ and true roots by x. $\mathbb{IR}$ denotes the domain of intervals over the real numbers $\mathbb{R}$ and variables written in bold type, e.g. $\mathbf{X}$, denote interval quantities. For a function f, we define $f(\mathbf{X}) = \{f(x) \mid x \in \mathbf{X}\}$. All errors are given in absolute terms. Error tolerance, that is, the maximum acceptable value for $|\tilde{x} - x|$, will be denoted by τ or tolerance. We will use the term *range arithmetic* to mean either interval arithmetic [12] or affine arithmetic [6]. The material presented in this section is valid for any such "arithmetic", as long as it computes guaranteed enclosures containing the result that would be computed in real numbers. We wish to compute a guaranteed bound on the error of a computed solution, that is, determine an upper bound on $\Delta x = \tilde{x} - x$. Note that Δx is different from τ, because Δ considers the sign of the difference.

Unary Case. For expository purposes, consider first the unary case $f : \mathbb{R} \to \mathbb{R}$, f differentiable, and suppose that we wish to solve the equation $f(x) = 0$. Then, by the Mean Value Theorem

$$f(\tilde{x}) = f(x + \Delta x) = f(x) + f'(\xi)\Delta x \qquad (4)$$

where $\xi \in \mathbf{X}$ and $\mathbf{X}$ is a range around $\tilde{x}$ sufficiently large to include the true root. Since $f(x) = 0$,

$$\Delta x \in \frac{f(\tilde{x})}{f'(\mathbf{X})} \qquad (5)$$

The set membership instead of equality is because the right-hand side is now a range-valued expression, which takes into account the fact that ξ in the Mean Value Theorem is not known exactly. The following theorem (stated in the formulation from [17]) formalizes this idea.

Theorem 1. *Let a differentiable function $f : \mathbb{R} \to \mathbb{R}$, $\mathbf{X} = [x_1, x_2] \in \mathbb{IR}$ and $\tilde{x} \in \mathbf{X}$ be given, and suppose $0 \notin f'(\mathbf{X})$. Define*

$$N(\tilde{x}, \mathbf{X}) := \tilde{x} - f(\tilde{x})/f'(\mathbf{X}). \qquad (6)$$

If $N(\tilde{x}, \mathbf{X}) \subseteq \mathbf{X}$, then $\mathbf{X}$ contains a unique root of f. If $N(\tilde{x}, \mathbf{X}) \cap \mathbf{X} = \varnothing$, then $f(x) \neq 0$ for all $x \in \mathbf{X}$.

Claim. If, following Equation 5, we compute an interval $\mathbf{\Delta x} = f(\tilde{x})/f'(\mathbf{X})$ enclosing the upper bound on the error Δx, and if $\mathbf{\Delta x} \subseteq [-\tau, \tau]$, then the approximately computed result $\tilde{x}$ is indeed within the specified precision τ.

Indeed, choose $\mathbf{X} = [\tilde{x} - \tau, \tilde{x} + \tau]$, i.e. the computed approximate solution plus or minus the tolerance we want to check, and compute $\mathbf{\Delta x} = \frac{f(\tilde{x})}{f'(\mathbf{X})}$. Then the condition $N(\tilde{x}, \mathbf{X}) \subseteq \mathbf{X}$ from Theorem 1 becomes

$$N(\tilde{x}, \mathbf{X}) = \tilde{x} - \mathbf{\Delta x} \subseteq \mathbf{X} = [\tilde{x} - \tau, \tilde{x} + \tau] \qquad (7)$$

If $\mathbf{\Delta x} \subseteq [-\tau, \tau]$, this condition holds, and thus the computed result is within the specified precision.

```
def assertBound (Function, Derivative, xn, τ)
  X = [xn ± τ]
  error = Function(xn) / Derivative(X)
  if error ∩ [−τ, τ] = ∅ throw SolutionNotIncludedException
  if ¬(error ⊂ [−τ, τ]) throw SolutionCannotBeVerifiedException
  return error
```

Fig. 2. Procedure for computing errors in the unary case.

Our assertion library uses the procedure in Figure 2 for unary problems. Note that we not only check that errors are within a certain error tolerance, but we also return the computed error bounds. As we show in Section 5, the computed error bounds tend to be much tighter than the user-required tolerance. As Section 4.3 illustrates, this error bound can be used in subsequent computations to track overall errors more precisely.

Multivariate Case. Our error estimates for the unary case follow from the Mean Value Theorem, which extends to n dimensions. Theorem 2 follows the interval formulation of [17] where J_f is the Jacobian matrix of f. If $\mathbf{D} = (\mathbf{x_1}, \ldots, \mathbf{x_n}) \in \mathbb{IR}^n$, let $\bar{\mathbf{D}}$ denote $\mathbf{x_1} \times \ldots \times \mathbf{x_n}$. For $a, b \in \bar{\mathbf{D}}$, define convex union as $a \cup b = \{a + \lambda b \mid \lambda \in [0,1]\}$. For $A \subseteq \bar{\mathbf{D}}$, define $hull(A) := \bigcap\{\mathbf{Z} \in \mathbb{IR}^n \mid A \subseteq \mathbf{Z}\}$.

Theorem 2. *Let there be given a continuously differentiable $f : \bar{\mathbf{D}} \to \mathbb{R}^n$ with $\mathbf{D} \in \mathbb{IR}^n$ and $x, \tilde{x} \in \bar{\mathbf{D}}$. Then for $\mathbf{X} := hull(x \cup \tilde{x})$*

$$f(x) \in f(\tilde{x}) + J_f(\mathbf{X})(x - \tilde{x}) \tag{8}$$

We extend our method for computing the error on each root in a similar manner:

$$\delta \in J_f^{-1}(\mathbf{X}) \cdot (-f(\tilde{x})) \tag{9}$$

where $\delta = x - \tilde{x}$ is the vector of errors on our tentative solution. Since we now must consider the Jacobian of f instead of a single derivative function, we can no longer solve for the errors by a simple scalar division. We wish to find the maximum possible error, so we need a way to compute an upper bound on the right-hand side of Equation 9. Computing the inverse of a Jacobian matrix in a range arithmetic typically does not yield a useful result, due to over-approximation. Instead, we use the following Theorem 3, which is originally due to [11], but we use the formulation by [17].

Theorem 3 ([17]). *Let $A, R \in \mathbb{R}^{n \times n}$, $b \in \mathbb{R}^n$ and $\mathbf{E} \in \mathbb{IR}^n$ be given, denote by I the identity matrix. Assume*

$$Rb + (I - RA)\mathbf{E} \subset int(\mathbf{E}). \tag{10}$$

where $int(\mathbf{E})$ denotes the interior of the set $\mathbf{E}$. Then the matrices A and R are non-singular and $A^{-1}b \in Rb + (I - RA)\mathbf{E}$.

We instantiate Theorem 3 with all possible matrices A such that $A \in J_f(\mathbf{X})$ and all possible vectors b such that $b \in -f(\tilde{x})$, where $J_f(\mathbf{X})$ and $-f(\tilde{x})$ are both evaluated in range arithmetic. Combining with Condition 9, we obtain

$$\delta \in J_f^{-1}(\mathbf{X}) * -f(\tilde{x}) \subseteq Rb + (I - RA)\mathbf{E}, \tag{11}$$

provided that Condition 10 is satisfied in range arithmetic.

Matrix R in Theorem 3 can be chosen arbitrarily as long as Condition 10 holds. A common choice is to use an approximate inverse of A. In our case, A is range-valued, so we first compute the matrix whose entries are the midpoints of the intervals of A, and use its inverse as R. It now remains to determine $\mathbf{X}$. We choose it to be the vector where the ith entry is the interval around $\tilde{x}_i$ and width τ. If we can then show that Condition 11 holds, we have proven that $\mathbf{X}$ indeed contains a solution. Moreover, we have computed a tighter upper bound on the error. We obtain the procedure in Figure 3 for computing error bounds for systems of equations. The variables Xn, A, b, E, errors are all range valued.

def assertBound (functions, Jacobian, xn, τ)
  Xn = [xn $\pm$ τ]
  A = Jacobian(Xn)
  b = $-$ functions(xn) *// goal is to certify that xn is a zero of 'functions' up to τ*
  R = inverse(mid(A)) *// calculated in ordinary floating points*
  E = [0 $\pm$ τ]
  errors = R$*$b + (I $-$ RA)E *// Theorem 3*
  if errors $\cap$ $[-\tau, \tau]^n = \varnothing^n$ **throw** SolutionNotIncludedException
  if $\neg$(errors $\subseteq$ $[-\tau, \tau]^n$) **throw** SolutionCannotBeVerifiedException
  return errors

Fig. 3. Procedure for computing errors in the multivariate case.

Our approach requires the derivatives to be non-zero, respectively the Jacobian to be non-singular, in the neighborhood of the root, . This means that, at present, we can only verify single roots. Verifying multiple roots is an ill-conditioned problem by itself, and thus requires further approximation techniques, as well as dealing with complex values. We leave this for future work. Our library does distinguish the cases when an error is provably too large from the case when our method is unable to ensure the result: we use two different exceptions for this purpose.

4 Implementation

Given the theoretical building blocks described above, the next question is how to integrate them into a general-purpose programming language. Our goal is to obtain an assertion framework for real numbers that is intuitive to use and

efficient. Figures 2 and 3 require the computation of derivatives and their evaluation in range arithmetic, but we do not want the user having to provide two differently typed functions, one in Doubles for the solver and one in Intervals for our verification method. Also, the solver may not actually require derivatives or the Jacobian, so this computation should be performed automatically and symbolically at compile time. Fortunately, Scala facilitates this within the existing compiler framework using a notion of macros.

4.1   Scala Macros

Scala version 2.10 (release candidate) introduces a macro facility [3]. To a user, macros look like regular methods, but in fact, their code is executed at compile time and performs a transformation on the Scala compiler abstract syntax tree (AST). Thus, by passing a regular function to a macro, we can access its AST and perform transformations, such as computing a derivative of an expression. The type checker runs after macro expansion, so the resulting code retains all guarantees from Scala's strong static typing. Our library provides the following functions:

def errorBound(f: (Double ⇒ Double), x: Double, tol: Double): Interval
def assertBound(f: (Double ⇒ Double), x: Double, tol: Double): Interval
def certify(root: Double, error: Interval): SmartFloat

and similarly for functions of two, three, and more variables. The function **assertBound** computes the guaranteed bounds on the errors using the algorithms in figures 2 and 3. **errorBound** removes the assertion check and only provides the computed error; the programmer is then free to define individual assertions. **certify** wraps the computed root(s) including their associated errors into a value of the SmartFloat datatype, thus providing a link to our assertion-checking framework. We also expose the automatic symbolic derivative computation facility:

def derivative(f: Double ⇒ Double): (Double ⇒ Double)
def jacobian(f1: (Double, Double) ⇒ Double, f2: (Double, Double) ⇒ Double):
   (Array[Array[(Double, Double) ⇒ Double]])

The functions passed to our macros have type (Double*) => Double and may be given as anonymous functions, or alternatively defined in the immediately enclosing method or class. The functions may use parameters, with the same restrictions on their original definitions. This is particularly attractive, as it allows us to write concise code as presented in the code snippets from Section 2. Source code including all examples can be downloaded from
`http://lara.epfl.ch/~darulova/cerres.zip`.

4.2   Computing Derivatives

In this section we explain how we compute derivatives and discuss the effects of our technique on efficiency and precision. Given the function ASTs, we compute

the derivatives or Jacobian matrices already at compile time, and thus need to do this symbolically. Our system computes derivatives with the standard derivation rules, such as the chain rule. Moreover, it performs the following expression simplifications:

- pull constants outside of multiplications (before differentiation);
- compact multiplications of the same terms into a power function (before differentiation);
- simplify multiplication and addition of zeros or ones arising from the differentiation (after differentiation);
- evaluate powers with integers by repeated multiplication (at runtime).

Overall, the effect is that the resulting expressions of derivatives do not grow too large. The second and third column of Table 3 show the impact of our optimizations on execution times: the cumulative improvement is 28%.

On the other hand, the syntactic algebraic form of the expressions affects the precision of evaluating them in floating-point, interval or affine arithmetic. To estimate this impact, we have compared the overall behavior of our system with our symbolic differentiation routine against the results obtained with manually provided derivatives. Manually means that the derivatives have the syntax one would compute by hand on paper. We did the comparison on our unary benchmark problems (Table 1), and it turns out that except for two instances, the errors computed are exactly the same. For the two other functions, our manually computed derivatives actually compute an error that is worse, but the precision is still sufficient to prove solutions are correct to within the given tolerance.

A possible alternative to our compile time differentiation is runtime automatic differentiation [9]. Because it does not perform the optimizations listed above, we would expect it to have performance similar to our unoptimized version. Although we have opted for symbolic differentiation at compile time, in principle one can use any type of function and differentiation method, as long as the function is differentiable in a sufficiently large neighborhood of the root and the differentiation method keeps track of the roundoff and method errors it commits.

4.3   Integration into a Roundoff Error Assertion Framework

We combine the current work with our existing library for tracking roundoff errors [5] into an assertion language that can be assumed to work with real numbers. That is, if no exceptions are thrown, the program would take the same path if real numbers were used instead of floating-points and the values computed are within the bounds computed by the SmartFloat datatype. This assertion language thus tracks two sources of errors

- quantization errors due to the discrete floating-point number representation (this has been implemented in [5]);

– method errors due to the approximate numerical method (this is a contribution of the present paper).

The bounds on computed values are ensured by using SmartFloats throughout the straight-line computations. Note that the numerical method still uses only Doubles since we verify the result a posteriori. Path consistency is ensured by the compare method of the SmartFloat datatype, which takes uncertainties into account. That is, if a comparison $x < y$ cannot be decided for sure due to uncertainties on the arguments, an exception is thrown. This behavior can be adjusted to a particular application by the methods

def certainly(b : $\Rightarrow$ Boolean) : Boolean =
   try b **catch** { **case** e: SmartFloatComparisonUndetermined $\Rightarrow$ **false** }
def possibly(b : $\Rightarrow$ Boolean) : Boolean =
   try b **catch** { **case** e: SmartFloatComparisonUndetermined $\Rightarrow$ **true** }

If we cannot be sure a boolean expression involving SmartFloats is true, we assume it is **false** in the case of **certainly**, and that it is **true** in the case of **possibly**. Hence, the following identity holds:

$$\textbf{if } (\textbf{certainly}(P)) \text{ T } \textbf{else } E \quad \Leftrightarrow \quad \textbf{if } (\textbf{possibly}(!P)) \text{ E } \textbf{else } T$$

4.4 Uncertain Parameters

Theorem 3 also holds for range-valued A and b. It is thus natural to extend our macro functions to also accept range-valued parameters. The SmartFloat datatype already has the facility to keep track of manually user-added errors so that we can track external uncertainties as a third source of errors. Consider again the gas state equation example from Section 2, especially the following two lines:

val N = 1000 +/− 5
val f = (V:D) $\Rightarrow$ (p + a∗(N.mid/V)∗(N.mid/V))∗(V − N.mid∗b) − k∗N.mid∗T

The +/− method returns an Interval, which in turn defines the mid method. Thus, the function typechecks correctly and can be passed for example to a solver, but inside the macro we can use the interval version of the parameter.

5 Evaluation

Precision Evaluation. The theorems from Section 3 provide us with sound guarantees regarding upper bounds. In practice however, we also need our method to be precise. Because our library computes error bounds and not only binary answers for assertions, we are interested in obtaining as precise error estimates as possible. We have evaluated the precision of our approach in the following way. We compute a high-precision estimate of the root(s) using a quadruple precision library [10], which allows us to compute the true error on the computed solutions with high confidence. We compare this error to the one provided by our

Table 1. Comparison of errors for unary functions. All numbers are rounded.

Problem (tolerance specified)	certified (affine)	certified (interval)	true errors
system of rods (1e-10)	7.315e-13	**1.447e-13**	1.435e-13
Verhulst model (1-e9)	4.891e-10	**9.783e-11**	9.782e-11
predator-prey model (1e-10)	7.150e-11	**7.147e-11**	7.146e-11
carbon gas state equation (1e-12)	**1.422e-17**	2.082e-17	1.625e-26
Butler-Volmer equation (1e-10)	4.608e-15	**3.896e-15**	3.768e-17
$(x/2)^2 - sin(x)$ (1e-10)	7.4e-16	**5.879e-16**	1.297e-16
$e^x(x-1) - e^{-x}(x+1)$ (1e-8)	5.000e-10	5.000e-10	5.000e-10
degree 3 polynomial (1e-7)	7.204e-9	**1.441e-9**	1.441e-9
degree 6 polynomial (1e-5)	**2.741e-14**	3.538e-14	2.258e-14

Table 2. Comparison of errors for multivariate functions. All numbers are rounded.

Problem (tolerance specified)	certified (affine)	certified (interval)	true errors
stress distribution (1e-10)	3.584e-11 3.584e-11	3.584e-11, 4.147e-11	4.147e-11 4.147e-11
sin-cosine system (1e-7)	6.689e-09 6.655e-09	6.689e-09 6.655e-09	6.689e-9 6.6545e-9
double pendulum (1e-13)	**4.661e-15** **6.409e-15**	5.454e-15 7.449e-15	5.617e-17 9.927e-17
circle-parabola intersection (1e-13)	**5.551e-17** 1.110e-16	1.110e-16 1.110e-16	8.0145e-51 5.373e-17
quadratic 2d system (1e-6)	**2.570e-12** 3.025e-09	3.326e-12 3.025e-09	2.192e-12 3.024e-9
turbine rotor (1e-12)	**1.517e-13** **1.707e-13** **1.908e-14**	1.523e-13 1.724e-13 1.955e-14	1.514e-13 1.703e-13 1.887e-14
quadratic 3d system (1e-10)	**4.314e-16** **5.997e-16** **4.349e-16**	6.795e-16 1.632e-15 5.127e-16	1.2134e-16 7.914e-17 7.441e-17

library. The results on a number of benchmark problems chosen from numerical analysis textbooks are presented in Tables 1 and 2. We are able to confirm the error bounds specified by the user in all cases. In fact, of all the examples we tried, our library failed only in the case of a multiple root for the reasons explained in Section 3 and never for precision reasons. We split the evaluation between the unary case and the multivariate case because of their different characteristics. All numbers are the maximum absolute errors computed. The numbers in parentheses are the tolerances given to the solvers and have been chosen randomly to simulate the different demands of the real world. We highlight the better error estimates in bold.

Note that the precision of the error estimates we obtain is remarkably good. Another perhaps surprising result of our experiments is that using interval arithmetic is generally more precise (in the unary case) or not much worse (in the mul-

Table 3. Average runtimes for the benchmark problems from Tables 1 and 2. Averages are taken over 1000 runs.

Problem set	solution time only	affine	interval	interval w/o optimizations	quadruple precision
unary problems	0.032ms	2.170ms	0.459ms	0.733ms	17.196ms
2D problems	0.044ms	2.779ms	0.984ms	1.240ms	4.446ms
3D problems	0.183ms	3.563ms	1.063ms	1.515ms	16.605ms

Table 4. Runtimes for individual problems. Averages are taken over 1000 runs.

Problem	affine	interval
carbon gas state equation	0.272ms	0.084ms
double pendulum problem	0.784ms	0.228ms
turbine problem	2.643ms	0.644ms
degree 3 polynomial	0.116ms	0.044ms
quadratic 2d system	0.425ms	0.200ms
quadratic 3d system	0.943ms	0.460ms

tivariate case) than affine arithmetic, although the latter is usually presented as the superior approach. Indeed, for the tracking of roundoff errors we have shown affine arithmetic to provide (sometimes much) better results than interval arithmetic [5]. The reason why intervals perform as well is that for transcendental functions they are able to compute a tighter range, since affine arithmetic has to compute a linear approximation of those functions. The exceptions in the unary case are the degree 6 polynomial and the carbon gas state equation example, which confirms our hypothesis, since in that case the dependency tracking of affine arithmetic can recover some of the imprecision in the long run.

For the multivariate case, affine arithmetic performs generally better because the computation consists to a large part of linear arithmetic. Due to the larger computation cost (see Section 5), however, we leave it as a choice for the user which arithmetic to use and select interval arithmetic as a default.

Performance Evaluation. Table 3 compares the performance of our implementation when using affine, interval arithmetic, or interval arithmetic without the differentiation optimizations listed in Section 4.2. Switching off the optimizations is similar to performing automatic differentiation. We can see that our optimizations actually make a big difference in the runtimes, improving by up to 37% for unary functions and 30% for our 3D problems over pure differentiation. On the other hand, the table clearly shows that affine arithmetic is much less efficient than interval arithmetic (factor 3-4.5 approx.), so it should only be used if precision is of importance. The first column shows the runtimes for computing the solutions with Newton's method without any kind of verification. We have also included the runtimes of re-computing the root(s) in quadruple precision [10]. That is we have used approximately 64 decimal digits for all calculations of the numerical method. The runtimes illustrate that this approach

for computing trustworthy results is unsuitable from the performance point of view, and would not actually provide any guarantees on errors.

Table 4 illustrates the dependence of runtimes on the complexity (operation count and dimension) of the problems. The first three problems are those from our example section 2 and the second set comprises relatively short polynomial equations. Runtimes depend both on the type of equations, as e.g. transcendental functions are more expensive, and on the size of the system of equations. We consider the increases appropriate given the increase of complexity of the problems.

6 Related Work

We are not aware of any work for general-purpose programming languages that could verify solutions of nonlinear constraints or that provides runtime assertions that are consistent with mathematical reals. Closest to our work are self-validated methods for solving systems of non-linear equations. [17] contains a fairly complete overview and an implementation exists in the INTLAB library [16]. The main difference to our work is that these methods are solution methods that use interval arithmetic throughout the computation. In contrast, we use the theorems from Section 3 as a *verification* method that accepts solutions computed by an arbitrary method. This allows us to leverage the generally good results and efficiency of numerical methods with sound results. Moreover, our implementation performs part of the computation already at compile time, and is thus more efficient.

In the case of systems of linear equations, one can use the linearity for optimizations [13]. The presented algorithm remains an iterative solver. [8] gives an iterative refinement algorithm for linear systems that uses higher precision arithmetic to compute the residual. The techniques cannot however be translated to nonlinear systems. Since we do not compute residuals that suffer heavily from cancellation errors in our approach, we believe that the additional cost of higher precision arithmetic is not warranted in order to achieve a slightly better precision. Another related area is that of approximate computation [19,2], which uses program transformations to trade accuracy for efficiency. The error bounds are generally provided by the user in form of trusted specifications or are determined by simulations. The results show a great potential for improving computation efficiency while retaining precision sufficient for the application.

7 Conclusion

We have shown how to integrate the theory of error estimation from numerical analysis into a general-purpose programming language. This allows us to estimate how close computed numerical quantities are from the corresponding values that would be computed using idealized operations on real numbers. As a result, it is now possible to use the well-developed theory of reals to reason about the programs manipulating floating points. The expectations of the programmer

can already be validated using runtime assertions that are easy and intuitive to use for developers. Static analysis approaches can complement our solution and can be built to use the same specification language.

References

1. A. Ayad and C. Marché. Multi-prover verification of floating-point programs. In *Fifth International Joint Conference on Automated Reasoning*, 2010.
2. W. Baek and T. M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *Proc. 2010 ACM SIGPLAN conference on Programming language design and implementation*, 2010.
3. E. Burmako, M. Odersky, C. Vogt, S. Zeiger, and A. Moors. Sip 16: Self-cleaning macros. *http://scalamacros.org/documentation/specification.html*, 2012.
4. G. Dahlquist and r. Björck. *Numerical Methods in Scientific Computing*. Society for Industrial and Applied Mathematics, 2008.
5. E. Darulova and V. Kuncak. Trustworthy numerical computation in Scala. In *Proc. 2011 ACM international conference on Object oriented programming systems languages and applications*, 2011.
6. L. H. de Figueiredo and J. Stolfi. *Self-Validated Numerical Methods and Applications*. IMPA/CNPq, Brazil, 1997.
7. D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and F. Vedrine. Towards an industrial use of FLUCTUAT on safety-critical avionics software. In *Proc. 14th International Workshop on Formal Methods for Industrial Critical Systems*, 2009.
8. J. W. Demmel, Y. Hida, W. Kahan, X. S. Li, S. Mukherjee, and E. J. Riedy. Error Bounds from Extra Precise Iterative Refinement. Technical report, EECS Department, University of California, Berkeley, 2005.
9. A. Griewank. A mathematical view of automatic differentiation. *Acta Numerica*, 12:321–398, 2003.
10. Y. Hida, S. L. Xiaoye, D. H. Bailey, and A. Kaiser. Quad Double computation package. *http://crd-legacy.lbl.gov/~dhbailey/mpdist/*, 2012.
11. R. Krawczyk. Newton-Algorithmen zur Bestimmung von Nullstellen mit Fehlerschranken. *Computing*, 4:187–201, 1969.
12. R. Moore. *Interval Analysis*. Prentice-Hall, 1966.
13. H. D. Nguyen and N. Revol. Solving and Certifying the Solution of a Linear System. *Reliable Computing*, 2011.
14. M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, 2008.
15. A. Quarteroni, F. Saleri, and P. Gervasio. *Scientific Computing with MATLAB and Octave*. Springer, 3rd edition, 2010.
16. S. Rump. INTLAB - INTerval LABoratory. In *Developments in Reliable Computing*. Kluwer Academic Publishers, 1999.
17. S. M. Rump. Verification methods: rigorous results using floating-point arithmetic. In *Proc. 2010 International Symposium on Symbolic and Algebraic Computation*, pages 3–4, 2010.
18. C. Woodford and C. Phillips. *Numerical Methods with Worked Examples*, volume 2nd. Springer, 2012.
19. Z. A. Zhu, S. Misailovic, J. A. Kelner, and M. Rinard. Randomized accuracy-aware program transformations for efficient approximate computations. In *Proc. 39th ACM SIGPLAN-SIGACT symp. Principles of programming languages*, 2012.