On the Automatic Computation of Error Bounds for Solutions of Nonlinear Equations

Eva Darulova and Viktor Kuncak *

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

Abstract. A large portion of software is used for numerical calculations in mathematics, physics and engineering applications. Among the things that make verification in this domain difficult is the quantification of numerical errors, such as roundoff errors and errors due to the approximate numerical method. 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: nonlinear equation, solution verification, affine arithmetic, floating-points

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 like Matlab [3] and Mathematica [4] 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 [16].

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

Existing validation of numerical computations supports estimation of roundoff errors [10,5]; we have previously incorporated computation of roundoff errors in Scala using affine arithmetic [8]. In this paper we go a step further and estimate not only roundoff errors, but additionally also method errors, which arise, for example, when using numerical methods to iteratively solve equations. Iterative methods are often 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.

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 δ such that $|x - x_k| < \delta$, where x is the actual solution in real numbers. Fortunately, we can estimate δ 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 [14] or affine arithmetic. However, this solution would be inefficient, and would give too pessimistic error bounds. Instead, our solution uses a runtime checking approach.

Contributions. We allow any standard non-validated floating point code to compute the approximation x_k. We perform only the final validation of a 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.

To perform such a computation, our approach uses static information about the function and computes derivatives at compile time, using the macro facility of Scala, an 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 several examples that model physical processes, taken from [20,7,17]. 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 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. Method names printed in bold are part of our library.

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 [20].

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

Given a library function computeRoot and our library for certifying solutions, the engineer can write the following code:

```
1 val f1 = (v: D,w: D,r: D) ⇒ 3 + 2/(r∗r) − 0.125∗(3−2∗v)∗(w∗w∗r∗r)/(1−v)−4.5
2 val f2 = (v: D,w: D,r: D) ⇒ 6∗v − 0.5 ∗ v ∗ (w∗w∗r∗r) / (1−v)−2.5
3 val f3 = (v: D,w: D,r: D) ⇒ 3 − 2/(r∗r) − 0.125∗(1+2∗v)∗(w∗w∗r∗r) / (1−v)−0.5
4 val x0 = Array(0.75, 0.5, 0.5)
5 val roots: Array[D] = computeRoot(Array(f1,f2,f3), jacobian(f1,f2,f3), x0, 1e−8)
6 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 and 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, and 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 that were computed are, in fact, smaller than the tolerance given to the numerical method used to compute the root.

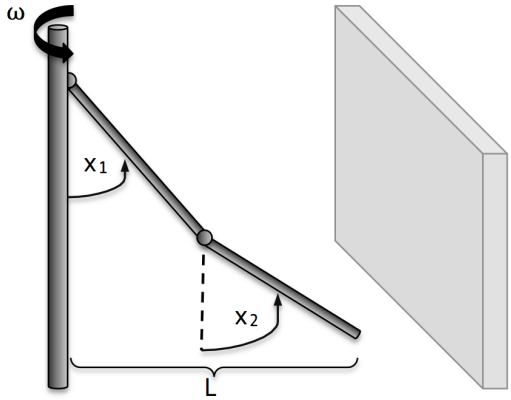

Fig. 1. A double pendulum standing close to an obstacle.

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)[7]. 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

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

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.

```
1  val distancePendulumWall : SmartFloat = ...
2  val length = ... //length of bars
3  val tolerance = 1e−13; val x0 = Array(0.18, 0.25)
4  val f1 = (x1: D, x2: D) ⇒ tan(x1) − k ∗ (2∗sin(x1) + sin(x2))
5  val f2 = (x1: D, x2: D) ⇒ tan(x2) − 2∗k ∗ (sin(x1) + sin(x2))
6  val r: Array[D] = computeRoot(Array(f1,f2), jacobian(f1,f2), x0, tolerance)
7  val roots: Array[SmartFloat] = certify(r, errorBound(f1, f2, r(0), r(1), tolerance))
8
9  val L: SmartFloat = _sin(roots(0)) ∗ length + _sin(roots(1)) ∗ length
10 if (certainly(L <= distancePendulumWall)) {
11    // continue computation
12 } else {
13    // reduce speed of the pendulum and repeat
14 }
```

To account for all sources of uncertainty, we use the SmartFloat data type developed previously [8]. 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 [17].

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

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:

```
1 val T = 300; val a = 0.401; val b = 42.7e−6;
2 val p = 3.5e7; val k = 1.3806503e−23; val x0 = 0.1
3 val N: Interval = 1000 +/− 5
4 val f = (V: D) ⇒ (p + a ∗ (N.mid / V) ∗ (N.mid / V)) ∗ (V − N.mid ∗ b)
5         − k ∗ N.mid ∗ T
6 val V: D = computeRoot(f, derivative(f), x0, 1e−9)
7 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 and 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 (previously denoted by δ). We will use the term range arithmetic to mean either interval [14] or affine arithmetic [9]. The material presented in this section is valid for any 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 the δ from Section 1, since here we consider the sign of the difference. For expository purposes, consider first the unary case $f : \mathbb{R} \to \mathbb{R}$ and suppose that we wish to solve the equation $f(x) = 0$. We assume without loss of generality the right-hand side of the equation to be zero (because any equation can be written in that form). Then, by the Mean Value Theorem

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

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

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

Note the inclusion instead of equality since the right hand side is now a range-valued expression. The following theorem (stated in the formulation from [18]) 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*

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

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 we compute an upper bound on the error as given in Equation 5 and it holds that $\Delta x \subseteq [-\tau, \tau]$, then the result $\tilde{x}$ that was computed is indeed within the specified precision τ.

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

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

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

Our assertion library uses Algorithm 1. Note that we do not only check that errors are within a certain error tolerance, but we also return the computed error bounds. As we show in Section 5.1, 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.

Algorithm 1

```
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
```

Our error estimates for the unary case follow from the Mean Value Theorem which can be extended for n dimensions. Theorem 2 follows the interval formulation of [18] where J_f denotes the Jacobian matrix of f.

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

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

for $\mathbf{X} := hull(x \cup \tilde{x})$, where $\cup$ denotes the convex union.

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

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

where δ is the vector of errors on our tentative solution. However, since we now have to consider the Jacobian of f instead of a single derivative function, we can no longer solve for the errors by a simple division. Since we want to find the maximum possible error, we need another means to compute an upper bound on the right-hand side of Equation 9. Note that computing the inverse in 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 [13], but we use the formulation by [18].

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

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

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

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

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

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 i^{th} 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 Algorithm 2 for computing error bounds for systems of equations. The variables X, A, b, z are all range valued.

Algorithm 2

```
def assertBound (functions, Jacobian, xn, τ (tolerance) )
  Xn = [xn ± τ]
  A = Jacobian(Xn)
  b = − functions(Xn)
  R = inverse(mid(A))
  X = [0 ± τ]
  errors = R∗b + (I − RA)X
  if errors ∩ [−τ, τ]^n = ∅^n throw SolutionNotIncludedException
  if ¬(errors ⊆ [−τ, τ]^n) throw SolutionCannotBeVerifiedException
  return errors
```

Our approach requires the derivatives to be non-zero in the neighborhood of the root, respectively the Jacobian to be non-singular. This means that we can only verify single roots at this point. 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

Now that we have the theoretical building blocks the question is how to integrate it into a general-purpose programming language like Scala such that the resulting assertion framework for real numbers is intuitive to use but at the same time

efficient. In particular, Algorithms 1 and 2 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, hence this computation should be performed automatically and symbolically at compile time. Fortunately, Scala facilitates this within the existing compiler framework using macros.

4.1 Scala Macros

Scala version 2.10 (release candidate) introduces a macro facility [2]. To the 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 the necessary transformations. The type checker runs after the macro expansion so that the resulting code retains all guarantees from Scala's strong static typing. Our library provides the following functions

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

and similarly for functions of 2, 3 and more variables. The function **assertBound** computes the guaranteed bounds on the errors using Algorithms 1 and 2. **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 in the SmartFloat datatype and hereby provides the link to our assertion checking framework. We also expose the automatic symbolic derivative computation facility:

```
1 def derivative(f: Double ⇒ Double): (Double ⇒ Double)
2 def jacobian(f1: (Double, Double) ⇒ Double, f2: (Double, Double) ⇒ Double):
3     (Array[Array[(Double, Double) ⇒ Double]])
4 ...
```

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

We now turn our attention to efficiency. Given the function ASTs, we compute the derivatives or Jacobian matrices already at compile time, and thus need to do this symbolically. The straightforward runtime option is to use automatic

differentiation [12]. We will show however that this incurs an unnecessary computation cost. It turns out that fairly simple optimizations on top of the usual derivative rules already provide the needed precision and efficiency:

- constants are pulled outside of multiplications (before derivation)
- multiplications of the same terms are compacted into a power function (before derivation)
- multiplication and addition of zeros or ones arising from the differentiation are simplified (after derivation)
- powers with integers are evaluated by repeated multiplication (at runtime)

Overall, the effect is that the resulting expressions of derivatives do not blow up. This is important for evaluation efficiency, since each operation carries a computation cost (see Table 3). On the other hand, precision may be affected as well, since the over-approximation committed by range arithmetic may depend on the formulation of the expression. We have compared the errors computed with our symbolic differentiation routine against the results obtained with manually provided derivatives. The latter have the format 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 manual derivatives actually compute an error that is worse, but the precision is still sufficient to prove solutions are correct to within the given tolerance.

4.3 Integration into Roundoff Error Assertion Framework

We combine the current work with our existing library for tracking roundoff errors [8] 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
- method errors due to the approximate numerical method

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

```
1 def certainly(b : ⇒ Boolean) : Boolean = {
2  try b catch SmartFloatComparisonUndetermined ⇒ false
3 }
4 def possibly(b : ⇒ Boolean) : Boolean = {
5  try b catch SmartFloatComparisonUndetermined ⇒ true
6 }
```

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:

if (**certainly**(P)) T **else** E ⇔ **if** (**possibly**(!P)) E **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) ⇒ (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

5.1 Precision

The theorems from Section 3 provide us with sound guarantees regarding upper bounds. In practice however, we also need our method to be precise. Since 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 [1], 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 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, on all examples that we tried, our library only failed 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.

First of all we 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 multivariate case) than affine arithmetic, although the latter is usually presented as the superior approach. Indeed, for the tracking of round-off errors we have shown affine arithmetic to provide (sometimes much) better results that interval arithmetic [8]. 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 5.2), however, we leave it as a choice for the user which arithmetic to use and select interval arithmetic as a default.

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	**3.8960e-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 1. Comparison of errors for unary functions. All numbers are rounded.

5.2 Performance

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 should only be used if precision is of importance.

We have also included the runtimes of re-computing the root(s) in quadruple precision [1]. That is we have used approximately 64 decimal digits for all calcu-

Problem (tolerance specified)	certified (affine)	certified (interval)	true errors
stress distribution (1e-10)	3.584e-11 4.147e-11	3.584e-11 4.147e-11	3.584e-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)	**5.5510e-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

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

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

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

lations of the numerical method. The runtimes illustrate that this approach for computing trustworthy results is clearly unsuitable from the performance point of view, and would not actually provide any guarantees on errors either, merely more confidence.

Table 4 illustrates the dependence of runtimes on the complexity of the problems. The first three problems are those from our example section 2 and the second set is comprised of relatively short polynomial equations. Clearly, runtimes depend both on the type of equations, transcendental functions being more expensive, as well as on the size of the system of equations. It should be noted however, that the increases are clearly appropriate given the increase of complexity of the problems.

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

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

6 Related Work

Closest to our work are self-validated methods for solving systems of non-linear equations. [18] contains a fairly complete overview and an implementation exists in the INTLAB library [19] for MATLAB [3]. The main difference to our work is that these methods are solutions instead 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 [15]. The presented algorithm remains an iterative solver. [11] 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 [21,6] that uses program transformations to trade of accuracy of computations against performance. In these approaches, the error bounds are generally provided by the user in form of trusted specifications and/or are determined by simulations.

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.

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. *QD (C++/Fortran-90 double-double and quad-double package).* http://crd-legacy.lbl.gov/~dhbailey/mpdist/, 1012.
2. *Scala Macros.* http://scalamacros.org/, 1012.
3. *MATLAB.* The MathWorks Inc., 2012.
4. *Wolfram Mathematica 8.* Wolfram Research Inc., 2012.
5. Ali Ayad and Claude Marché. Multi-prover verification of floating-point programs. In *Fifth International Joint Conference on Automated Reasoning*, 2010.
6. Woongki Baek and Trishul M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *PLDI*, 2010.
7. Germund Dahlquist and Åke Björck. *Numerical Methods in Scientific Computing.* Society for Industrial and Applied Mathematics, 2008.
8. Eva Darulova and Viktor Kuncak. Trustworthy numerical computation in Scala. In *OOPSLA*, 2011.
9. L. H. de Figueiredo and J. Stolfi. *Self-Validated Numerical Methods and Applications.* IMPA/CNPq, Brazil, 1997.
10. David Delmas, Eric Goubault, Sylvie Putot, Jean Souyris, Karim Tekkal, and Franck Vedrine. Towards an industrial use of FLUCTUAT on safety-critical avionics software. In *FMICS*, 2009.
11. James W. Demmel, Yozo Hida, William Kahan, Xiaoye S. Li, Soni Mukherjee, and E. Jason Riedy. Error Bounds from Extra Precise Iterative Refinement. Technical report, EECS Department, University of California, Berkeley, 2005.
12. Andreas Griewank. A mathematical view of automatic differentiation. *Acta Numerica*, 12:321–398, 2003.
13. R. Krawczyk. Newton-Algorithmen zur Bestimmung von Nullstellen mit Fehlerschranken. *Computing*, 4:187–201, 1969.
14. R.E. Moore. *Interval Analysis.* Prentice-Hall, 1966.
15. Hong Diep Nguyen and Nathalie Revol. Solving and Certifying the Solution of a Linear System. *Reliable Computing*, 2011.
16. Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-step Guide.* Artima Incorporation, 2008.
17. Alfio Quarteroni, Fausto Saleri, and Paola Gervasio. *Scientific Computing with MATLAB and Octave.* Springer, 3rd edition, 2010.
18. Siegfried M. Rump. Verification methods: rigorous results using floating-point arithmetic. In *Proceedings of the 2010 International Symposium on Symbolic and Algebraic Computation*, pages 3–4, 2010.
19. S.M. Rump. INTLAB - INTerval LABoratory. In *Developments in Reliable Computing.* Kluwer Academic Publishers, 1999.
20. C. Woodford and C. Phillips. *Numerical Methods with Worked Examples*, volume 2nd. Springer, 2012.
21. Zeyuan Allen Zhu, Sasa Misailovic, Jonathan A. Kelner, and Martin Rinard. Randomized accuracy-aware program transformations for efficient approximate computations. In *POPL*, 2012.