Certifying Solutions for Numerical Constraints

September 2012

Eva Darulová, Viktor Kuncak
Reasoning about the *real* world matters

“Numerical software is used to test scientific theories, design airplanes and bridges, operate manufacturing lines, control power plants and refineries, ...”

Accuracy and Reliability in Scientific Computing, 2005

Numerical software has special sources of errors:
- floating-point roundoff
- method errors (discretization, truncation, stopping criteria)
- model errors (assumptions about the physical model)
Example

The equilibrium position of the rotating double pendulum is given by:

\[ \tan x_1 - k(2\sin x_1 + \sin x_2) = 0 \]
\[ \tan x_2 - 2k(\sin x_1 + \sin x_2) = 0 \]

Suppose a black-box method solves the system for \( k = 0.3 \), with precision parameter \( 1e-8 \):

\[ x_1 = 0.179779407413519 \]
\[ x_2 = 0.250801371841681 \]

How close is \((x_1, x_2)\) to the true equilibrium?
Residual

As a possible check we can plug in the values:

\[
\begin{align*}
    f_1(x_1, x_2) &= 1.5208594605908843 \times 10^{-8} \\
    f_2(x_1, x_2) &= -1.0592200283543463 \times 10^{-8}
\end{align*}
\]

The true error is in fact one order of magnitude larger: \(1 \times 10^{-7}\).

\[\varepsilon \approx 0.025\]

\[\tau \approx 0.055\]
Contribution

Runtime assertions that verify error bounds on solutions of systems of equations, taking into account

• roundoff errors due to floating-point arithmetic
• ‘method’ errors due to the characteristics of the functions

We provide over-approximations sound with respect to reals.

✓“this was precise enough”
Certified results
sound error estimation for solutions of systems of equations

SmartFloat
Data type for floating-point roundoff error estimation [OOPSLA’11]

method errors

roundoff errors

Assertion language that can be assumed to work with real numbers.

• if no exceptions are thrown, the program would take the same path if real numbers were used instead of floating-points
• values computed are within the bounds computed by the SmartFloat datatype
val tolerance = 1e−8;
val x0 = Array(0.18, 0.25)

val f1 = (x1: Double, x2: Double) ⇒ tan(x1) − k * (2 * sin(x1) + sin(x2))
val f2 = (x1: Double, x2: Double) ⇒ tan(x2) − 2 * k * (sin(x1) + sin(x2))

val r: Array[Double] = computeRoot(Array(f1, f2), x0, tolerance)

val roots: Array[SmartFloat] = certify(f1, f2, r(0), r(1), tolerance)

val distancePendulumWall : SmartFloat = ...
val length = ... //length of bars

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
}

sound wrt. to floating-points
What we are after exactly…

Given a system of nonlinear equations (polynomials, trigonometric functions, \ldots)

\[ \bar{f}(\bar{x}) = \bar{0} \]

that has a solution \( x_* \), and given a tolerance \( \tau \), compute an approximate root \( x \) such that

\[ |\Delta x| = |x - x_*| \leq \tau \]

i.e. the computed solution is \textbf{guaranteed} to be off the true root by at most \( \tau \).

One approach: compute the root by iterating validated steps

Our alternative:

- reuse any existing (efficient) solution method
- verify the result a posteriori in a sound way
- verify results from unknown or closed sources
Unary case: \( f : \mathbb{R} \rightarrow \mathbb{R}, \quad f(x_*) = 0 \)

1. By the Mean Value Theorem:
\[
f(x) = f(x_* + \Delta x) = f(x_*) + f'(\zeta) \Delta x
\]  
\[\text{where } \zeta \in X \text{ and } X \text{ is sufficiently large to include } x, x_* .\]

2. \( \Delta x \in \frac{f(x)}{f'(X)} \) where the RHS is evaluated in range arithmetic.

3. Then one of the following holds:
   - \( \Delta x \subseteq [-\tau, \tau] \) unique root
   - \( \Delta x \cap [-\tau, \tau] = \emptyset \) root definitely not in interval
   - \( \neg (\Delta x \subset [-\tau, \tau]) \) don’t know

In general: \( \max \text{Abs} (\Delta x) \leq \tau \)
we can return a tighter error estimate for subsequent use.
def assertBound (Function, Derivative, xn, τ) {

    error = computeError(Function, Derivative, xn, τ)

    if error ⊆ [-τ, τ]
        return error

    if error ∩ [-τ, τ] == ∅
        throw SolutionNotIncludedException

    if ¬(error ⊂ [-τ, τ])
        throw SolutionCannotBeVerifiedException

}
Multivariate case

Mean Value Theorem generalizes to

\[ \Delta x \in \frac{f(x)}{f'(X)} \Rightarrow \Delta x \in J^{-1}(X) \cdot f(x) \] (2)

Evaluating the inverse in range arithmetic is unstable.

But, if it holds

\[ Rb + (I - RA)E \subset \text{interior}(E) \] (3)

then there is a theorem that tells us that it also holds:

\[ A^{-1}b \in Rb + (I - RA)E \] (4)

Then we can pick

\[ A = J(X), \quad b = f(x), \quad R \approx J^{-1}, \quad E = [-\tau, \tau]^n \]

so that we get:

\[ \Delta x \in J^{-1}(X) \cdot f(x) \subseteq Rf + (I - RJ)E \] (5)

Returns individual error bounds for each component of the root.
Chicken and egg

By the Mean Value Theorem:
\[ f(x) = f(x_\ast + \Delta x) = f(x_\ast) + f'(\zeta)\Delta x \] (1)

where \( \zeta \in X \) and \( X \) is sufficiently large to include \( x, x_\ast \).

Which \( X \) to choose?

For soundness: \( X \) needs to be large enough.

For precision: \( X \) needs to be as small as possible.

Our solution: use the tolerance specified by the user in the assertion.
\[ X = [x - \tau, x + \tau] \]
def assertBound (Function, Derivative, xn, τ) {
    error = computeError(Function, Derivative, xn, τ)
    if error ⊆ [−τ, τ]
        return error
    if error ∩ [−τ, τ] == ∅
        throw SolutionNotIncludedException
    if ¬(error ⊂ [−τ, τ])
        throw SolutionCannotBeVerifiedException
}
Derivatives Symbolically

1. pull constants outside of multiplications
   \[ x \cdot 6.0 \cdot y \rightarrow 6.0 \cdot (x \cdot y) \]
2. compact multiplications of the same terms
   \[ x \cdot x \rightarrow x^2 \]
3. apply rules for differentiation of
   \[ x \times y, \sin, x^n, \ldots \]
4. simplify
   \[ 1.0 \cdot x \rightarrow x \]
   \[ 0.0 + x \rightarrow x \]
5. evaluate powers with integers by repeated multiplication
   \[ x^5 \rightarrow x \cdot x \cdot x \cdot x \cdot x \]

- precision is not affected for our benchmarks
- the resulting expressions do not grow too large

<table>
<thead>
<tr>
<th>Problem set</th>
<th>with interval certification</th>
<th>interval w/o optimizations</th>
<th>improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>unary problems (set of 9)</td>
<td>0.459ms</td>
<td>0.733ms</td>
<td>37%</td>
</tr>
<tr>
<td>2D problems (set of 5)</td>
<td>0.984ms</td>
<td>1.240ms</td>
<td>21%</td>
</tr>
<tr>
<td>3D problems (set of 2)</td>
<td>1.063ms</td>
<td>1.515ms</td>
<td>30%</td>
</tr>
</tbody>
</table>
We choose a light-weight implementation using Scala macros (Scala 2.10).

- transformations on Scala compiler ASTs
- run at compile time
- type checker runs afterwards

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

Inputs may be:

- anonymous functions, or
- functions defined in the immediately enclosing method or class
- functions may use parameters
Uncertainties on parameters

Consider the state equation of a gas.
N is the number of molecules.

\[
\text{val } f = (V: \text{Double}) \Rightarrow \\
(p + a \times (N / V) \times (N / V)) \times (V - N \times b) - k \times N \times T
\]
Uncertainties on parameters

Consider the state equation of a gas. N is the number of molecules.

\[
\text{val } N: \text{ Interval } = 1000 +/− 5
\]

\[
\text{val } f = (V: \text{ Double}) \Rightarrow
(p + a \ast (N.\text{mid}/V) \ast (N.\text{mid}/V)) \ast (V − N.\text{mid} \ast b) − k \ast N.\text{mid} \ast T
\]

\[
\text{val } V: D = \text{computeRoot}(f, \text{derivative}(f), x0, 1e−9)
\]\n
assertBound(f, V, 0.0005)

Our library computes the certified bound on V:
[ 0.0424713, 0.0429287 ]

We can track **external uncertainties** as a third source of errors.
<table>
<thead>
<tr>
<th>Problem</th>
<th>certified affine</th>
<th>certified interval</th>
<th>true errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>system of rods (1e-10)</td>
<td>7.315e-13</td>
<td>1.447e-13</td>
<td>1.435e-13</td>
</tr>
<tr>
<td>Verhulst model (1-e9)</td>
<td>4.891e-10</td>
<td>9.783e-11</td>
<td>9.782e-11</td>
</tr>
<tr>
<td>carbon gas state equation (1e-12)</td>
<td>1.422e-17</td>
<td>2.082e-17</td>
<td>1.625e-26</td>
</tr>
<tr>
<td>degree 6 polynomial</td>
<td>2.741e-14</td>
<td>3.538e-14</td>
<td>2.258e-14</td>
</tr>
<tr>
<td>stress distribution (1e-10)</td>
<td>3.584e-11</td>
<td>3.584e-11</td>
<td>3.584e-11</td>
</tr>
<tr>
<td></td>
<td>4.147e-11</td>
<td>4.147e-11</td>
<td>4.147e-11</td>
</tr>
<tr>
<td>double pendulum (1e-13)</td>
<td>4.661e-15</td>
<td>5.454e-15</td>
<td>5.617e-17</td>
</tr>
<tr>
<td>turbine rotor (1e-12)</td>
<td>1.517e-13</td>
<td>1.523e-13</td>
<td>1.514e-13</td>
</tr>
<tr>
<td></td>
<td>1.707e-13</td>
<td>1.724e-13</td>
<td>1.703e-13</td>
</tr>
<tr>
<td></td>
<td>1.908e-14</td>
<td>1.955e-14</td>
<td>1.887e-14</td>
</tr>
<tr>
<td>quadratic 3d system (1e-10)</td>
<td>4.314e-16</td>
<td>6.795e-16</td>
<td>1.2134e-16</td>
</tr>
<tr>
<td></td>
<td>5.997e-16</td>
<td>1.632e-15</td>
<td>7.914e-17</td>
</tr>
<tr>
<td></td>
<td>4.349e-16</td>
<td>5.127e-16</td>
<td>7.441e-17</td>
</tr>
</tbody>
</table>
## Performance

<table>
<thead>
<tr>
<th>Problem set</th>
<th>solution time only</th>
<th>with affine certification</th>
<th>with interval certification</th>
<th>quadruple precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>unary problems</td>
<td>0.032ms</td>
<td>2.170ms</td>
<td>0.459ms</td>
<td>17.196ms</td>
</tr>
<tr>
<td>(set of 9)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2D problems</td>
<td>0.044ms</td>
<td>2.779ms</td>
<td>0.984ms</td>
<td>4.446ms</td>
</tr>
<tr>
<td>(set of 5)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3D problems</td>
<td>0.183ms</td>
<td>3.563ms</td>
<td>1.063ms</td>
<td>16.605ms</td>
</tr>
<tr>
<td>(set of 2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Solution time only for solution with Newton’s method with Doubles only. Quadruple precision uses approximately 64 decimal digits.
Conclusion

We provide runtime assertions that verify error bounds on solutions of systems of equations, taking into account

- **roundoff errors** due to floating-point arithmetic
- ‘**method’ errors** due to the characteristics of the functions

We provide over-approximations **sound with respect to reals**.

- **widely applicable**: solutions computed by any method
- **easy to use**: no modifications to existing code
- **precise**: tight error bounds
- **efficient**: much faster than high-precision
Thank you.