Trustworthy Numerical Computation in Scala

Eva Darulová, Viktor Kuncak
Reasoning about the real world matters

The most widely used data type are floating-points

• IEEE 754-2008 standard gives precise behaviour
• efficient and, we hope, adequate is many cases


“It makes us nervous to fly an airplane since we know they operate using floating-point arithmetic.”, Xavier Leroy. Verified squared: does critical software deserve verified tools? In POPL, 2011.
How do you know you can trust your numerical computation?
def rootKahan(a: Double, b: Double, c: Double) {
    val discr = b * b - a * c * 4.0
    if (b*b - a*c > 10.0 && b > 0.0)
        return c * 2.0 /(-b - sqrt(discr))
    else
        return (-b + sqrt(discr))/(a * 2.0)
}

Equivalently in real numbers:

def root(a: Double, b: Double, c: Double) {
    val discr = b * b - a * c * 4.0
    return (-b + sqrt(discr))/(a * 2.0)
}

scala> root(2.999, 56.000003, 1.00076)
res0: Double = -0.017887849139318127

scala> rootKahan(2.999, 56.000003, 1.00076)
res0: Double = -0.017887849139317836
Floating-points

- Standard defines:
  - arithmetic formats (incl. NaN, infinities)
  - interchange formats
  - rounding rules
  - operations
  - exception handling

- (mostly full) hardware support
- varying software support
IEEE 754 software support

• JVM
  - \{+, -, *, /, \sqrt{\}} rounded to nearest
  - \sin, \cos, \ldots: API-specified roundoff errors

• C99
  - low-level control of hardware (all rounding modes)
  - beware of compiler optimizations

• CPython
  - math module wrapper around C library functions
  - “almost all platforms map Python floats to IEEE-754 “double precision”
How do you know you can trust your numerical computation?
Interval arithmetic

- interval width $\sim$ maximum roundoff error

$$b \in [1.01, 1.02]$$
$$b \times b \in [1.02, 1.05] \quad //1.01^2 = 1.0201, \quad 1.02^2 = 1.0404$$

scala> root(2.999, 56.000003, 1.00076)
[-0.017887849139321683, -0.017887849139313385] (2.6514e-13)

scala> rootKahan(2.999, 56.000003, 1.00076)
[-0.017887849139317846, -0.017887849139317825] (5.8187e-16)
Problems with interval arithmetic

• **Imprecision**: losing dependencies
  
  \[ x \in [0, 1] \]
  
  \[ u = x + 3 \quad u \in [3, 4] \]
  
  \[ z = u - x \quad z \in [2, 4] \quad \text{but } z = x + 3 - x = 3! \]

• **Lack of generality**: input range vs. roundoff

What is the maximum roundoff error over an entire input range?

```
scala> root(Interval(2.0, 4.0), Interval(50.0, 60.0), Interval(0.5, 1.5))
[−2.560144695375273,2.4916643505649514] (1.0073e+00)
```
Contributions

Rigorous numerical data types for precision and generality

Dependency-preserving estimation of roundoff errors of a concrete floating-point computation.

AffineFloat

Estimation of upper bounds on roundoff errors over an entire range of input values.

SmartFloat
Affine arithmetic

\[ x = x_0 + \sum_{i=1}^{n} x_i \varepsilon_i, \quad \varepsilon_i \in [-1, 1] \]

central value

max. magnitude of noise term

- represents the interval

\[ \text{rad}(\hat{x}) = \sum_{i=1}^{n} |x_i| \]

- affine operations (+, -)

\[ \alpha \hat{x} + \beta \hat{y} + \zeta = (\alpha x_0 + \beta y_0 + \zeta) + \sum_{i=1}^{n} (\alpha x_i + \beta y_i) \varepsilon_i + \nu \varepsilon_{n+1} \]

- non-linear operations need a linear approximation
Affine arithmetic

\[ x = x_0 + \sum_{i=1}^{n} x_i \epsilon_i, \quad \epsilon_i \in [-1, 1] \]

- central value
- max. magnitude of noise term
- noise symbol

• avoids dependency problem for linear operations

\[ x = 0.5 + 0.5\epsilon_1 \quad \in [0, 1] \]

\[ u = x + 3 = 3.5 + 0.5\epsilon_1 \]

\[ z = u - x = 3.5 + 0.5\epsilon_1 - 0.5 - 0.5\epsilon_1 = 3.0 \]
AffineFloat data type

\[ x = x_0 + \sum_{i=1}^{n} r_i \varepsilon_i \]

- computed Double value
- roundoff errors

- each operation adds a new noise term
- each operation propagates existing noise terms

\[ \text{roundoff} = \sum_{i=1}^{n} |r_i| \]
SmartFloat data type

\[ x = \left( x_0 + \sum_{i=1}^{n} x_i \epsilon_i , \sum_{i=1}^{m} r_i \rho_i \right) \]

- At each operation, adds the worst-case roundoff error for all possible values
- Propagation of errors is a little more involved
- maximum roundoff = \( \sum_{i=1}^{n} |r_i| \)
The quest for precision

In our implementation, we face the same roundoff errors that we aim to quantify!

- directed rounding in C++
- DoubleDouble precision
- precise handling of constants
- recognizing exact computations
- dependency problem with multiplication
- non-linear operations
Nonlinear approximations

Chebyshev approximation
- needs a 3rd point, whose rounding direction is not clear
- can give *wrong* results for small input intervals

Minrange approximation
- rounding direction is clear

Less precise, but reliable!
Integration into Scala

```scala
def rootKahan(a: SmartFloat, b: SmartFloat, c: SmartFloat) {
  val discr = b * b - a * c * 4.0
  if (b*b - a*c > 10.0 && b > 0.0)
    return c * 2.0 /(-b - sqrt(discr))
  else
    return (-b + sqrt(discr))/(a * 2.0)
}
```

- easy integration with implicit and strong type inference
- support for most common math. functions (`exp, sin, cos, log, Pi, ...`
- symmetric equals

scala> rootKahan(SmartFloat(3.0, 1.0),
       SmartFloat(55.0, 5.0), SmartFloat(0.5, 1.5))
[-0.13109336344405553, 0.09429437802880317] (7.5543e-16)
## Precision: AffineFloat vs. Intervals

LU: solution to $Ax = b$ by factorizing $A$

FFT: Fast Fourier Transform, followed by its inverse

<table>
<thead>
<tr>
<th></th>
<th>Intervals</th>
<th>AffineFloat</th>
</tr>
</thead>
<tbody>
<tr>
<td>LU 5x5, with pivoting</td>
<td>6.69e-13</td>
<td>1.04e-13</td>
</tr>
<tr>
<td>LU 10x10</td>
<td>2.13e-10</td>
<td>7.75e-12</td>
</tr>
<tr>
<td>LU 15x15</td>
<td>1.92e-8</td>
<td>6.10e-10</td>
</tr>
<tr>
<td>LU 5x5, no pivoting</td>
<td>1.24e-9</td>
<td>2.50e-11</td>
</tr>
<tr>
<td>LU 10x10</td>
<td>4.89e-6</td>
<td>2.38e-10</td>
</tr>
<tr>
<td>FFT 512</td>
<td>6.43e-12</td>
<td>9.73e-13</td>
</tr>
<tr>
<td>FFT 256</td>
<td>2.38e-12</td>
<td>3.03e-13</td>
</tr>
</tbody>
</table>

Up to 4 decimal orders of magnitude improvement!
Generality: Doppler frequency shift

\[ q_1 = 331.4 + 0.6T \]
\[ q_2 = q_1 v \]
\[ q_3 = q_1 + u \]
\[ q_4 = q_3 \times q_3 \]
\[ z = \frac{q_2}{q_4} \]

\(-30^\circ C \leq T \leq 50^\circ C\)
\[ 20Hz \leq v \leq 20000Hz \]
\[-100 \frac{m}{s} \leq u \leq 100 \frac{m}{s} \]

<table>
<thead>
<tr>
<th></th>
<th>SMT[1]</th>
<th>bits</th>
<th>SmartFloat</th>
<th>abs. roundoff</th>
</tr>
</thead>
<tbody>
<tr>
<td>q1</td>
<td>[313, 362]</td>
<td>6</td>
<td>[313.3999,361.40]</td>
<td>8.6908e-14</td>
</tr>
<tr>
<td>q2</td>
<td>[6267, 7228000]</td>
<td>23</td>
<td>[6267.9999,7228000.00]</td>
<td>3.3431e-09</td>
</tr>
<tr>
<td>q3</td>
<td>[213, 462]</td>
<td>8</td>
<td>[213.3999,461.40]</td>
<td>1.4924e-13</td>
</tr>
<tr>
<td>q4</td>
<td>[45539, 212890]</td>
<td>18</td>
<td>[44387.5599,212889.96]</td>
<td>1.6135e-10</td>
</tr>
<tr>
<td>z</td>
<td>[0, 138]</td>
<td>8</td>
<td>[-13.3398,162.7365]</td>
<td>6.8184e-13</td>
</tr>
</tbody>
</table>

running time: order **100s**

our running time: order **1s**

### Performance (ms)

<table>
<thead>
<tr>
<th></th>
<th>double</th>
<th>interval</th>
<th>AffineFloat</th>
<th>SmartFloat</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nbody (100 steps)</td>
<td>2.1</td>
<td>21</td>
<td>779</td>
<td>33756</td>
</tr>
<tr>
<td>Spectral norm (10 iter.)</td>
<td>0.6</td>
<td>31</td>
<td>198</td>
<td>778</td>
</tr>
<tr>
<td>Whetstone (10 repeats)</td>
<td>1.2</td>
<td>2</td>
<td>59</td>
<td>680</td>
</tr>
<tr>
<td>Fbench</td>
<td>0.2</td>
<td>1.3</td>
<td>10</td>
<td>1082</td>
</tr>
<tr>
<td>Scimark - FFT (512x512)</td>
<td>1.2</td>
<td>18</td>
<td>1220</td>
<td>39987</td>
</tr>
<tr>
<td>Scimark - SOR (100x100)</td>
<td>0.8</td>
<td>25</td>
<td>698</td>
<td>127168</td>
</tr>
<tr>
<td>Scimark - LU (50x50)</td>
<td>2.6</td>
<td>30</td>
<td>2419</td>
<td>4914</td>
</tr>
<tr>
<td>Spring sim. (10000 steps)</td>
<td>0.2</td>
<td>46</td>
<td>1283</td>
<td>4086</td>
</tr>
</tbody>
</table>

- acceptable for understanding floating-point computations
- slower than a hardware implementation, but faster than existing approaches that achieve similar precision
Semantics for floating-point programs

• interval arithmetic
• affine arithmetic

• stochastic arithmetic
  Run the program repeatedly with random rounding. Mainly useful for finding stability issues.

• automatic differentiation
  Computes the derivate of a program to expose sensitivities to input changes.
Floating-point verification

• **Abstract interpretation**
  Computes an overapproximation of variable values used to
  – guarantee no run-time errors can occur (Astree)
  – roundoff errors are within certain bounds (Fluctuat)

• **Model-checking**
  Models a floating-point computation as a finite-state system and
  performs a path sensitive analysis
  – precise but expensive

• **SAT**
  Encodes floating-point operations bit-precisely (basically encodes the
circuit) and checks the formula against user-provided assertions.
  – check for exceptions (e.g. underflow)
Floating-point verification

• Theorem proving

  Provide code contracts (specifications) about the precision of methods and check the properties with a theorem prover.
  – detailed specification necessary
  – interaction with the theorem prover

Example: check that a piece of code is overflow-safe:

```plaintext
@rnd = float<ieee_32,ne>;
z = rnd(rnd(x * x) + rnd(sqrt(y)));
{ |x| <= 2 /\ y in [1,9]
  -> z in [1,7] /\ |rnd(x * x)| <= 0x1.FFFFEp127 /\ |
           rnd(sqrt(y))| <= 0x1.FFFFEp127 }
```
That’s all.

http://lara.epfl.ch/w/smartfloat
1. Compact all other terms based on average errors and their deviation.
2. For pathological cases, compact all noise symbols into a single one.

\[
\hat{x} = x_0 + \sum_{i=1}^{n} x_i \epsilon_i \quad \Rightarrow \quad \hat{x} = x_0 + \sum_{i=1}^{m} x_i \epsilon_i , \quad m < n
\]

Precision \longleftrightarrow Performance
Packing of noise terms

![Graph showing the average running time in ms for different algorithms as the maximum number of noise symbols increases. The x-axis represents the maximum number of noise symbols, ranging from 20 to 100. The y-axis represents the average running time in ms, ranging from 0 to 2000. The graph includes lines for Nbody, Spectral, Fbench, SOR, LU, Spring, and FFT.]