Trustworthy Numerical Computation in Scala

EPFL

Eva Darulová, Viktor Kuncak

http://lara.epfl.ch/~edarulov/TrustworthyComputation.pdf

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

"… ***any*** operation involving numerical realization of a geophysical algorithm led to significant disagreement.", L. Hatton and A. Roberts. How Accurate is Scientific Software? In *IEEE Trans. Softw. Eng.,* 20, 1994.

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

Equivalently in real numbers:

```
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)
}
```

```
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 IEEE 754

- 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, …* : 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

$$b \in [1.01, 1.02]$$

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

```
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

$$
\begin{array}{lll}
 & x \in [0, 1] & \\
u = x + 3 & u \in [3, 4] & \\
z = u - x & z \in [2, 4] & \text{but } z = x + 3 - x = 3!
\end{array}
$$

- 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

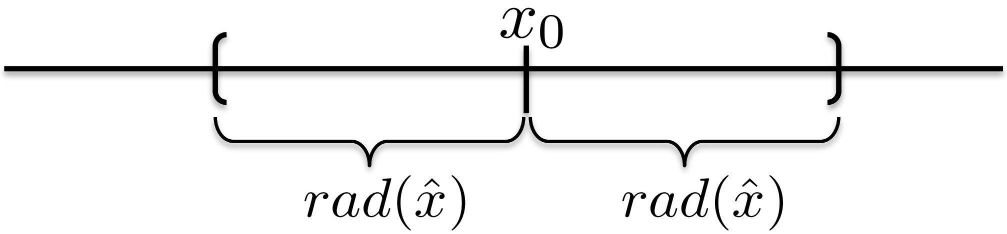
noise symbol

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

central value

max. magnitude of noise term

- represents the interval



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

- affine operations (+, -)

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

- non-linear operations need a linear approximation

Affine arithmetic

noise symbol

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

central value

max. magnitude of noise term

- avoids dependency problem for linear operations

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

$$\begin{aligned} u &= x + 3 = 3.5 + 0.5\epsilon_1 \\ z &= u - x = 3.5 + 0.5\epsilon_1 - 0.5 - 0.5\epsilon_1 \\ &= 3.0 \end{aligned}$$

`AffineFloat` data type

$$x = x_0 + \sum_{i=1}^{n} r_i \epsilon_i$$

computed Double value

roundoff errors

- each operation adds a new noise term
- each operation propagates existing noise terms
- roundoff = $\sum_{i=1}^{n} |r_i|$

`SmartFloat` data type

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

uncertainty on variable

maximum roundoff errors

- 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

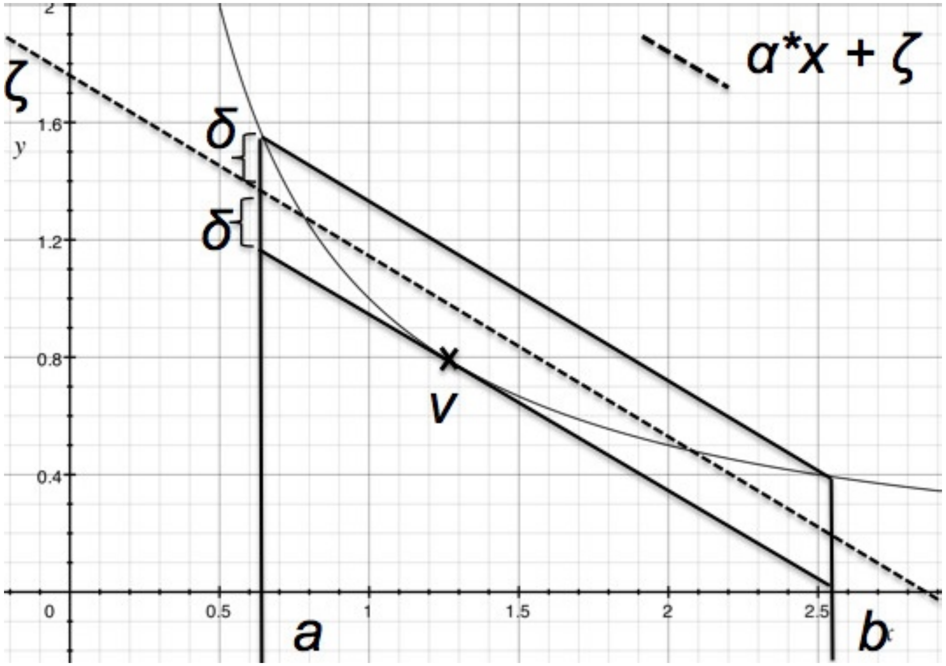
In our implementation, we face the same roundoff errors that we aim to quantify!

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

Nonlinear approximations

Chebyshev approximation

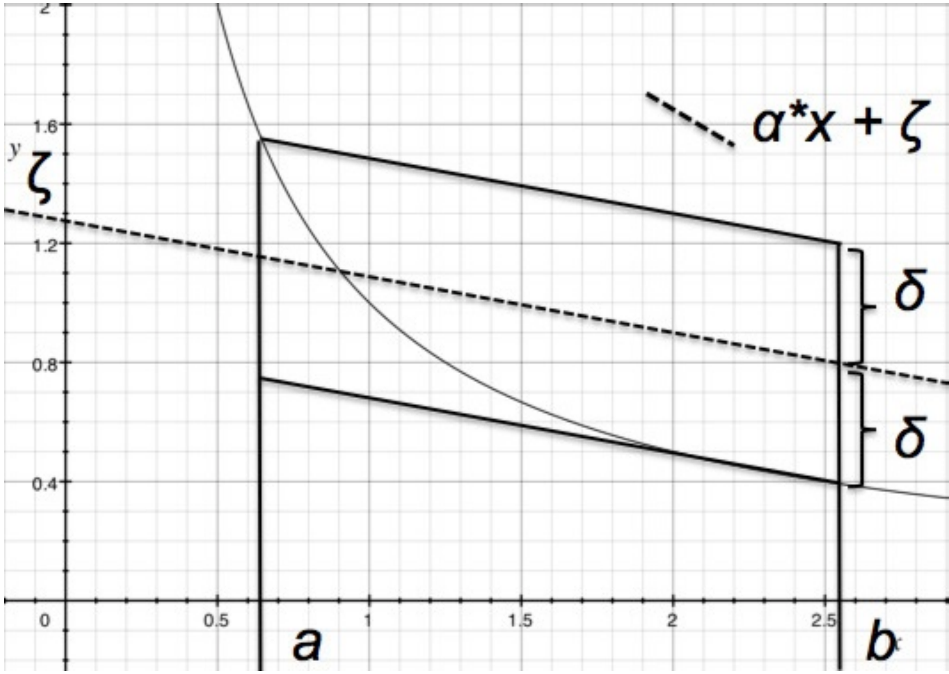
- needs a 3rd point, whose rounding direction is not clear
- can give **wrong** results for small input intervals



Minrange approximation

- rounding direction is clear

Less precise, but reliable!



Integration into Scala

```
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 implicits 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))
[-0.1310933634405553,0.09429437802880317] (7.5543e-16)
```

Precision: AffineFloat vs. Intervals

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

FFT: Fast Fourier Transform, followed by its inverse

	Intervals	**AffineFloat**
LU 5x5, with pivoting LU 10x10 LU 15x15	6.69e-13 2.13e-10 1.92e-8	1.04e-13 7.75e-12 6.10e-10
LU 5x5, no pivoting LU 10x10	1.24e-9 4.89e-6	2.50e-11 2.38e-10
FFT 512 FFT 256	6.43e-12 2.38e-12	9.73e-13 3.03e-13

Up to 4 decimal orders of magnitude improvement!

Generality: Doppler frequency shift

```
q1 = 331.4 + 0.6T
q2 = q1v
q3 = q1 + u
q4 = q3*q3
z = q2 / q4
```

$$-30°C \le T \le 50°C$$

$$20Hz \le v \le 20000Hz$$

$$-100\,\tfrac{m}{s} \le u \le 100\,\tfrac{m}{s}$$

	SMT[1]	**bits**	**SmartFloat**	**abs. roundoff**
q1	[313, 362]	6	[313.3999,361.40]	8.6908e-14
q2	[6267, 7228000]	23	[6267.9999,7228000.00]	3.3431e-09
q3	[213, 462]	8	[213.3999,461.40]	1.4924e-13
q4	[45539, 212890]	18	[44387.5599,212889.96]	1.6135e-10
z	[0, 138]	8	[-13.3398,162.7365]	6.8184e-13
	running time: order **100s**		**our running time: order 1s**	

[1] A.B. Kinsman, N. Nicolici. Finite Precision bit-width allocation using SAT-Modulo Theory. DATE, 2009.

Performance (ms)

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

- 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:

```
@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.FFFFFEp127 /\
      |rnd(sqrt(y))| <= 0x1.FFFFFEp127 }
```

That’s all.

http://lara.epfl.ch/w/smartfloat

Packing

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

Precision ⟷ *Performance*

1. Compact all other terms based on average errors and their deviation.
2. For pathological cases, compact all noise symbols into a single one.

Packing of noise terms