Karlsruhe Institute of Technology

Analyzing Floating-Point Programs

Debasmita Lohar

Formal Systems II: Applications, Summer 2025

In this Part

you will learn the fundamentals of analyzing floating-point programs*:

27.06.

- Challenges
- Basics of Floating-Point Arithmetic
- Dataflow analysis
  - Interval Arithmetic (IA)
  - Floating-Point IA
  - Analysis of roundoff errors with IA
  - Demo: Daisy

01.07.

More on Dataflow Analysis

- Pro’s and con’s of Intervals
- Affine Arithmetic (AA)
- Analysis of roundoff errors with AA
- Interval Subdivision
- Other Approaches and Recent Directions
- Demo: Daisy

*Slides are based on Program Analysis Course (WS 20/21) at UdS + RPTU

Motivation

- Models of the physical world, control algorithms, etc:
    - Real-valued Arithmetic

- Computer implementations:
    - Finite Precision: e.g. Floating-Point Arithmetic

$\mathbb{R}$: `0.1 + 0.1 + 0.1 = 0.3` 2min

Try it in any programming language!

Motivation

- Models of the physical world, control algorithms, etc:
  - Real-valued Arithmetic
- Computer implementations:
  - Finite Precision: e.g. Floating-Point Arithmetic

$$\mathbb{R}\texttt{: 0.1 + 0.1 + 0.1 = 0.3}$$

$$\mathbb{F}\texttt{: 0.1 + 0.1 + 0.1 = 0.30000000000000004}$$

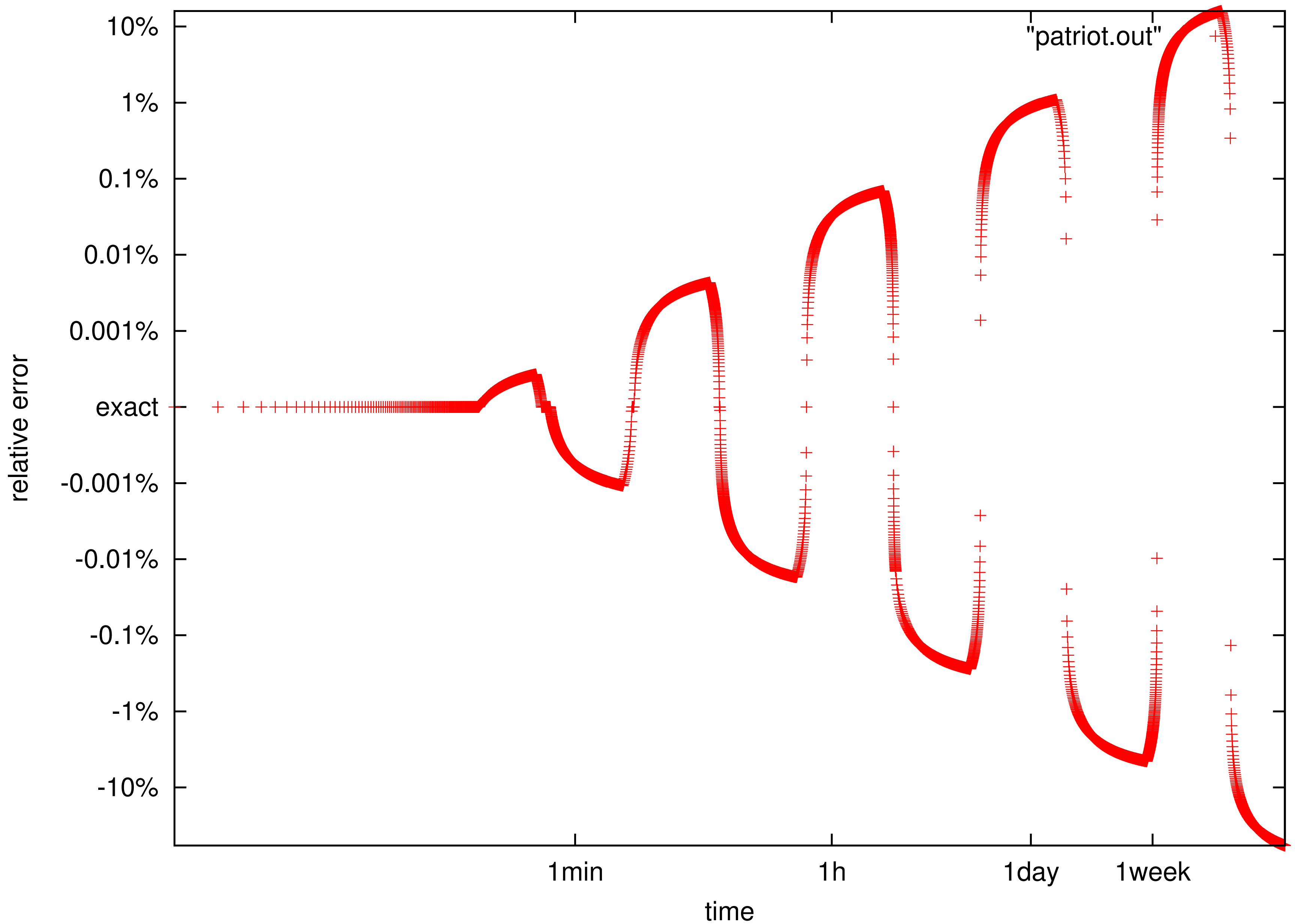
Xavier Leroy and many, many others:

"It makes us nervous to fly an airplane since we know they OPERATE using floating-point arithmetic."

Verified squared: does critical software deserve verified tools? Talk at POPL, 2011.

Accumulated Errors (a.k.a the Patriot bug)

```
float t = 0.0; while(1) { ... t = t * 0.1; ... }
```



Consequence: 28 casualties

Source: Verified squared: does critical software deserve verified tools? X. Leroy, talk at POPL, 2011.

Basics: Floating-Point Arithmetic

defined by IEEE 754 standard*

Representation:

$$(-1)^s \cdot m \cdot 2^e$$

- base `2` (base `10` also possible)
- `s ∈ {0,1}` : sign
- `m` : significand
- `e` : exponent

precision	sign	`m` bits	`e` bits
single (32)	1	23	8
double (64)	1	52	11
quad (128)	1	112	15

*IEEE 754-2008 standard, August 2008, available at https://ieeexplore.ieee.org/document/4610935.

Basics: Floating-Point Arithmetic

defined by IEEE 754 standard

Representation: $(-1)^s \cdot m \cdot 2^e$

precision	sign	`m` bits	e bits	e bias
single (32)	1	23	8	127
double (64)	1	52	11	1023
quad (128)	1	112	15	16383

floating-point formats in biased representation

What is the decimal number in single precision?

2min

`0`	`01111111`	`10000000000000000000000`
sign	exponent	significand

$$= (-1)^0 \cdot (1 + \frac{1}{2}) \cdot 2^{127-127}$$

$$= 1.5$$

Basics: Floating-Point Arithmetic

$$(-1)^s \cdot m \cdot 2^e$$

precision	sign	`m` bits	`e` bits	`e` bias
single (32)	1	23	8	127
double (64)	1	52	11	1023
quad (128)	1	112	15	16383

Limited precision → need to round every operation

Arithmetic operations (+, -, *, /, √) are accurately rounded, i.e.

- as if computed in real arithmetic and then rounded

Rounding modes:

- round to nearest
- round to 0
- round to +∞
- round to -∞

Abstraction for round to nearest:

$$x \circ_{fp} y = (x \circ y) \cdot (1 + \delta) \qquad \text{where} \, |\delta| \leq \epsilon_m$$

Special Values

$$(-1)^s \cdot m \cdot 2^e$$

precision	sign	`m` bits	`e` bits	e bias
single (32)	1	23	8	127
double (64)	1	52	11	1023
quad (128)	1	112	15	16383

Limited range → overflow, underflow

Special values: +∞, -∞, +`0.0`, `-0.0`, `NaN` (Not-a-Number)

`1.0 / 0.0` → Infinity

`1.0 / (-0.0)` → -Infinity

sqrt(`-42.0`) → NaN

NaN * `0.0` → NaN

NaN == NaN → false

More unintuitive behavior

Floating-point arithmetic is commutative, but not associative and distributive

$$x + (y + z) \mathrel{!=} (x + y) + z$$

$$x * (y * z) \mathrel{!=} (x * y) * z$$

$$x * (y + z) \mathrel{!=} (x * y) + (x * z)$$

Further,

$$x / 10 \mathrel{!=} x * 0.1$$

$$x == y \nRightarrow 1/x == 1/y$$

$$x \mathrel{!=} x$$

Overview

27.06.

- Challenges
- Basics of Floating-Point Arithmetic
- Dataflow analysis
  - Interval Arithmetic (IA)
  - Floating-Point IA
  - Analysis of roundoff errors with IA
  - Demo: Daisy

Intervals: Basic Terms and Concepts

```
# Each variable represents a closed interval
x = [1, 3], y = [-5, 4]
```

$$x \subseteq y$$

- x and y are **intervals over** $\mathbb{Z}$:

$$x := \{[l_1, h_1] \mid l_1 \leq h_1, l_1 \in \mathbb{Z} \cup \{-\infty\}, h_1 \in \mathbb{Z} \cup \{+\infty\}\}$$

$$y := \{[l_2, h_2] \mid l_2 \leq h_2, l_2 \in \mathbb{Z} \cup \{-\infty\}, h_2 \in \mathbb{Z} \cup \{+\infty\}\}$$

- with ordering: $[l_1, h_1] \subseteq [l_2, h_2] \quad \text{iff} \quad l_2 \leq l_1 \text{ and } h_1 \leq h_2$

Interval Arithmetic: From Principles to Implementation, T. Hickey, Q. Ju, M. H. Van Emden, JACM 2001

Intervals: Basic Terms and Concepts

```
# Each variable represents a closed interval
x = [1, 3], y = [-5, 4]
```

$x \neq y$

- x and y are **intervals over** $\mathbb{Z}$:

$$x := \{[l_1, h_1] \mid l_1 \leq h_1, l_1 \in \mathbb{Z} \cup \{-\infty\}, h_1 \in \mathbb{Z} \cup \{+\infty\}\}$$

$$y := \{[l_2, h_2] \mid l_2 \leq h_2, l_2 \in \mathbb{Z} \cup \{-\infty\}, h_2 \in \mathbb{Z} \cup \{+\infty\}\}$$

- with ordering: $[l_1, h_1] \subseteq [l_2, h_2] \quad \text{iff} \quad l_2 \leq l_1 \text{ and } h_1 \leq h_2$

- Two intervals x and y are equal if they are the same:

$$x = y \quad \text{if} \quad l_1 = l_2, h_1 = h_2$$

Intervals: Basic Terms and Concepts

```
# Each variable represents a closed interval
x = [1, 3], y = [-5, 4]
```

$$| x | = 3, | y | = 5$$

- x and y are **intervals over** $\mathbb{Z}$:

$$x := \{[l_1, h_1] \mid l_1 \leq h_1, l_1 \in \mathbb{Z} \cup \{-\infty\}, h_1 \in \mathbb{Z} \cup \{+\infty\}\}$$

$$y := \{[l_2, h_2] \mid l_2 \leq h_2, l_2 \in \mathbb{Z} \cup \{-\infty\}, h_2 \in \mathbb{Z} \cup \{+\infty\}\}$$

- with ordering: $[l_1, h_1] \subseteq [l_2, h_2] \quad \text{iff} \quad l_2 \leq l_1 \text{ and } h_1 \leq h_2$
- Two intervals x and y are equal if they are the same:

$$x = y \quad \text{if} \quad l_1 = l_2, h_1 = h_2$$

- Absolute values of intervals:

$$| x | = max\{ \, | l_1 | \, , | h_1 | \, \}$$

$$| y | = max\{ \, | l_2 | \, , | h_2 | \, \}$$

Interval Arithmetic (IA)

```
# Each variable represents a closed interval
x = [1, 3], y = [-5, 4]
add = x + y = [-4, 7]
```

- Addition: $x + y := [l_1 + l_2, h_1 + h_2]$

Interval Arithmetic (IA)

```
# Each variable represents a closed interval
x = [1, 3], y = [-5, 4]
add = x + y
sub = x - y = [3, 8]
```

- Addition: $x + y := [l_1 + l_2, h_1 + h_2]$
- Subtraction: $x - y := [l_1 - h_2, h_1 - l_2]$

Interval Arithmetic (IA)

```
# Each variable represents a closed interval
x = [1, 3], y = [-5, 4]
add = x + y
sub = x - y
mul = x * y = [-15, 12]
```

- Addition: $x + y := [l_1 + l_2, h_1 + h_2]$
- Subtraction: $x - y := [l_1 - h_2, h_1 - l_2]$
- Multiplication: $x \times y := [min((l_1 \times l_2), (l_1 \times h_2), (h_1 \times l_2), (h_1 \times h_2)),$
  $max((l_1 \times l_2), (l_1 \times h_2), (h_1 \times l_2), (h_1 \times h_2))]$

Interval Arithmetic (IA)

```
# Each variable represents a closed interval
x = [1, 3], y = [-5, 4]
add = x + y
sub = x - y
mul = x * y
div = x / y
```

$0 \in y$

1. ✔ Does the denominator contain 0? undefined!
2. In $\mathbb{Z}$, the results are computed by checking all combinations
3. Uses floor divisions for soundness!

- Addition: $x + y := [l_1 + l_2, h_1 + h_2]$
- Subtraction: $x - y := [l_1 - h_2, h_1 - l_2]$
- Multiplication: $x \times y := [min((l_1 \times l_2), (l_1 \times h_2), (h_1 \times l_2), (h_1 \times h_2)),$
  $max((l_1 \times l_2), (l_1 \times h_2), (h_1 \times l_2), (h_1 \times h_2))]$
- Division: $x/y :=$

Interval Arithmetic (IA)

```
# Each variable represents a closed interval
x = [1, 3], y = [1, 4]
add = x + y
sub = x - y
mul = x * y
div = x / y
```

1. Does the denominator contain 0?
2. In ℤ, the results are computed by checking all combinations
3. Uses floor divisions for soundness!

- Addition: $x + y := [l_1 + l_2, h_1 + h_2]$
- Subtraction: $x - y := [l_1 - h_2, h_1 - l_2]$
- Multiplication: $x \times y := [min((l_1 \times l_2), (l_1 \times h_2), (h_1 \times l_2), (h_1 \times h_2)),$
  $max((l_1 \times l_2), (l_1 \times h_2), (h_1 \times l_2), (h_1 \times h_2))]$
- Division: $x/y := [min(a/b), max(a/b)],$
  where $a \in \{l_1, h_1\}, b \in \{l_2, h_2\}$ and $0 \notin [l_2, h_2]$

Interval Arithmetic (IA)

```
# Each variable represents a closed interval
x = [1, 3], y = [1, 4]
add = x + y
sub = x - y
mul = x * y
div = x / y = [0, 3]
```

- Addition: $x + y := [l_1 + l_2, h_1 + h_2]$
- Subtraction: $x - y := [l_1 - h_2, h_1 - l_2]$
- Multiplication: $x \times y := [min((l_1 \times l_2), (l_1 \times h_2), (h_1 \times l_2), (h_1 \times h_2)), max((l_1 \times l_2), (l_1 \times h_2), (h_1 \times l_2), (h_1 \times h_2))]$
- Division: $x/y := [min(a/b), max(a/b)],$ where $a \in \{l_1, h_1\}, b \in \{l_2, h_2\}$ and $0 \notin [l_2, h_2]$

How is IA useful for floats?

Let's assume we want to develop an analysis which can prove that special values +∞, -∞, NaN do not appear in a program execution.

```
inputs: x, y

w = 1.0 / (x + y)

z = sqrt(w)
```

→

```
inputs: x, y     // x,y ∈ [a, b]

t = x + y        // show 0 ∉ δ[t]
w = 1.0 / t

z = sqrt(w)    // show δ[w] non-negative
```

Note: we could do integer-valued interval analysis, but it would be very imprecise

Floating-Point IA

Define the domain of **intervals over** $\mathbb{F}$ as:

$$I := \{[l, h] \mid l \leq h, l \in \mathbb{F}, h \in \mathbb{F}\}$$

with **ordering**: $[l_1, h_1] \subseteq [l_2, h_2]$ iff $l_2 \leq l_1$ and $h_1 \leq h_2$

For soundness, arithmetic operations need to be **rounded outwards**:

- $[l_1, h_1] \oplus [l_2, h_2] \coloneqq [l_1 +\!\downarrow l_2, h_1 +\!\uparrow h_2]$
- $[l_1, h_1] \ominus [l_2, h_2] \coloneqq [l_1 -\!\downarrow h_2, h_1 -\!\uparrow l_2]$
- $[l_1, h_1] \otimes [l_2, h_2] \coloneqq [\min\{l_1 l_2\downarrow, l_1 h_2\downarrow, h_1 l_2\downarrow, h_1 h_2\downarrow\}, \max\{l_1 l_2\uparrow, l_1 h_2\uparrow, h_1 l_2\uparrow, h_1 h_2\uparrow\}]$
- $[l_1, h_1] \oslash [l_2, h_2] = [\min\{l_1\downarrow \times (1/h_2)\downarrow, l_1\downarrow \times (1/l_2)\downarrow, h_1\downarrow \times (1/h_2)\downarrow, h_1\downarrow \times (1/l_2)\downarrow\},$
  $\max\{l_1\uparrow \times (1/h_2)\uparrow, l_1\uparrow \times (1/l_2)\uparrow, h_1\uparrow \times (1/h_2)\uparrow, h_1\uparrow \times (1/l_2)\uparrow\}]$

where $\downarrow$: rounded to $-\infty$ and $\uparrow$: rounded to $+\infty$

Roundoff Errors

Before: Analysis for showing absence of "**runtime errors**":

inputs: x, y

w = 1.0 / (x + y)   // no div-by-zero

z = sqrt(w)   // no sqrt of negative number

Next: Analysis for verifying the accuracy of a computation

- ✔ assume no infinities, NaNs can be checked with interval analysis!
- compute worst-case roundoff errors: i.e. difference to real-valued execution
  - ➡ needed e.g. to check validity of controller stability

Accuracy

- Absolute Errors: $$\max_{x \in I} |f(x) - \tilde{f}(\tilde{x})|$$

input spec — real-valued program — floating-point program

- Relative errors: $$\max_{x \in I} \frac{|f(x) - \tilde{f}(\tilde{x})|}{|f(x)|}$$

Problem: if 0 ∈ f(x), the expression is not well-defined
- focus on absolute errors

Intervals for Individual Executions

f(x) = x + 3.1*x*x

Idea:

Track individual executions with floating-point interval arithmetic

x = 3.14 // [3.14↓, 3.14↑]

in the end:

roundoff error ≤ width of interval

+ Often used in numerical analysis
+ Easy to implement
- Only provides information about a single execution

Interval Analysis for Roundoffs

Goal: compute worst-case roundoff error estimate for a range of executions

$$z = \underbrace{eval(x \odot y)}_{\text{arithmetic expression}} \quad // x \in [-10,10], y \in [-10,10]$$

Idea: track ranges of variables and errors separately

- we need the ranges, because roundoff errors depend on them

step 1. Compute real-valued range:

step 2. Compute the errors: $\underline{propagatedError} + newRoundoffError$

Error Propagation

For arithmetic operations: $eval(x \odot y)$

- real-valued range: $x_{range} \odot y_{range}$
- error: $propagatedError + newRoundoffError$

Error propagation depends on the arithmetic operation:

- Addition: $\tilde{x} + \tilde{y} = (x + err_x) + (y + err_y) = \underbrace{x + y}_{\text{real value}} + \underbrace{err_x + err_y}_{\text{propagated error}}$
- Subtraction: $\tilde{x} - \tilde{y} = (x + err_x) - (y + err_y) = \underbrace{x - y}_{\text{real value}} + \underbrace{err_x - err_y}_{\text{propagated error}}$

Error Propagation

For arithmetic operations: $eval(x \odot y)$

- real-valued range: $x_{range} \odot y_{range}$
- error: $propagatedError + newRoundoffError$

Error propagation depends on the arithmetic operation:

- Multiplication:

$$\tilde{x} \times \tilde{y} = (x + err_x) \times (y + err_y) = \underbrace{x \times y}_{\text{real value}} + \underbrace{x \times err_y + y \times err_x + err_x \times err_y}_{\text{propagated error}}$$

- Division: compute inverse and then multiplication

- Inverse and Square Root: compute linear approximation

New Roundoff Error

For arithmetic operations: $eval(x \odot y)$

- real-valued range: $x_{range} \odot y_{range}$
- error: $propagatedError + newRoundoffError$

Recall floating-point abstraction (round to nearest):

$$x \circ_{fp} y = (x \circ y) \cdot (1 + \delta) \qquad \text{where} \, |\delta| \leq \epsilon_m$$

$$x \circ_{fp} y = \underbrace{x \circ y}_{\text{real value}} + \underbrace{(x \circ y) \cdot \delta}_{\text{error}}$$

- New worst-case roundoff error: $\max |x \circ y| \cdot \epsilon_m$
- $x \circ y$ here includes propagated errors
- ϵ_m depends on the floating-point precision

Quiz

2min

What is the range for z after the operation?

$x \mapsto ([1, 2], [-0.1, 0.1]),\ y \mapsto ([3, 4], [-0.3, 0.3])$

z = x + y

a)

$([3.6, 6.4], [-0.4, 0.4])$

b)

$([4, 6], [-0.4, 0.4] + [-0.4, 0.4]\epsilon_M)$

c)

$([4, 6], [-0.4, 0.4] + [3.6, 6.4]\epsilon_M)$

Quiz

2min

What is the range for z after the operation?

$x \mapsto ([1, 2], [-0.1, 0.1]), y \mapsto ([3, 4], [-0.3, 0.3])$

z = x + y

a)

([3.6, 6.4], [-0.4, 0.4])

b)

$([4, 6], [-0.4, 0.4] + [-0.4, 0.4]\epsilon_M)$

c) ✓

$([4, 6], [-0.4, 0.4] + [3.6, 6.4]\epsilon_M)$

Demo ~40min

Prerequisite: Docker

- Ubuntu/Debian/Fedora/CentOS:
  https://docs.docker.com/engine/install/
- MacOS:
  https://docs.docker.com/desktop/setup/install/mac-install/
- Windows:
  https://docs.docker.com/desktop/setup/install/windows-install/

Daisy - Framework for Analysis and Optimization of Numerical Programs,
E. Darulova, A. Izycheva, F. Nasir, F. Ritter, H. Becker, R. Bastian, TACAS'18

Demo ~40min

Installation:

```
- sudo docker pull dlohar/daisy
- sudo docker run -it dlohar/daisy
```

```
float-analysis-class/Test.scala:
```

```scala
import daisy.lang._
import Real._
object Test {
  def test(x: Real, y: Real): Real = {
    require(1 <= x && x <= 3 && -5 <= y && y <= 4)

    val add = x + y
    add
  }
}
```

Daisy - Framework for Analysis and Optimization of Numerical Programs,
E. Darulova, A. Izycheva, F. Nasir, F. Ritter, H. Becker, R. Bastian, TACAS'18

Demo ~40min

Running Daisy:

```
- sbt script
- ./daisy float-analysis-class/Test.scala
    --rangeMethod=interval --errorMethod=interval
    --precision=Float32
```

Try it with other precision, input ranges,
arithmetic operations, noInitialErrors etc!