Expanding the Horizons of Finite-Precision Analysis

Debasmita Lohar

PhD Defense Talk

27th March, 2024

MAX PLANCK INSTITUTE
FOR SOFTWARE SYSTEMS

UNIVERSITÄT
DES
SAARLANDES

Programming with Finite-Precision

```
def controller(x:Real, y:Real, z:Real): Real = {
  val res = -x*y - 2*y*z - x - z
  return res
}
```

Programming with Finite-Precision

```
               (x:Float32, y:Float32, z:Float32): Float32
def controller(~~x:Real, y:Real, z:Real): Real~~ = {
  val res = -x*y - 2*y*z - x - z
  return res
}
```

- Reals are implemented in Floating-point / Fixed-point data type

Errors in Finite-Precision

```
def controller(x:Float32, y:Float32, z:Float32): Float32 = {
  val res = -x*y - 2*y*z - x - z
  return res
}        +/- error
```

- Reals are implemented in Floating-point / Fixed-point data type
- Introduces roundoff errors at potentially every operation

Errors in Finite-Precision

```
def controller(x:Float32, y:Float32, z:Float32): Float32 = {
  val res = -x*y - 2*y*z - x - z
  return res
}       +/- error
```

```
0.1 + 0.2 = 0.3
```

real arithmetic

```
>>> 0.1 + 0.2
0.30000000000000004
```

32-bit floating-point arithmetic

Errors in Finite-Precision

```
def controller(x:Float32, y:Float32, z:Float32): Float32 = {
  val res = -x*y - 2*y*z - x - z
  return res
}
```

`+/- error`

```
0.1 + 0.2 = 0.3
```

real arithmetic

```
>>> 0.1 + 0.2
0.30000000000000004
```

32-bit floating-point arithmetic

Does it even affect real-world systems?

Finite-Precision Errors in Real World

February 1991, Dhahran, Saudi Arabia

Gulf War: Loss of accuracy led to failure in US defense system, 28 killed!

April 1992, Schleswig-Holstein, Germany

Rounding error changed Parliament makeup!

June 1996

Overflow led to explosion of Ariane 5, 39s after lift-off, $370 million lost!

…

Finite-Precision Errors in Real World

February 1991, Dhahran, Saudi Arabia

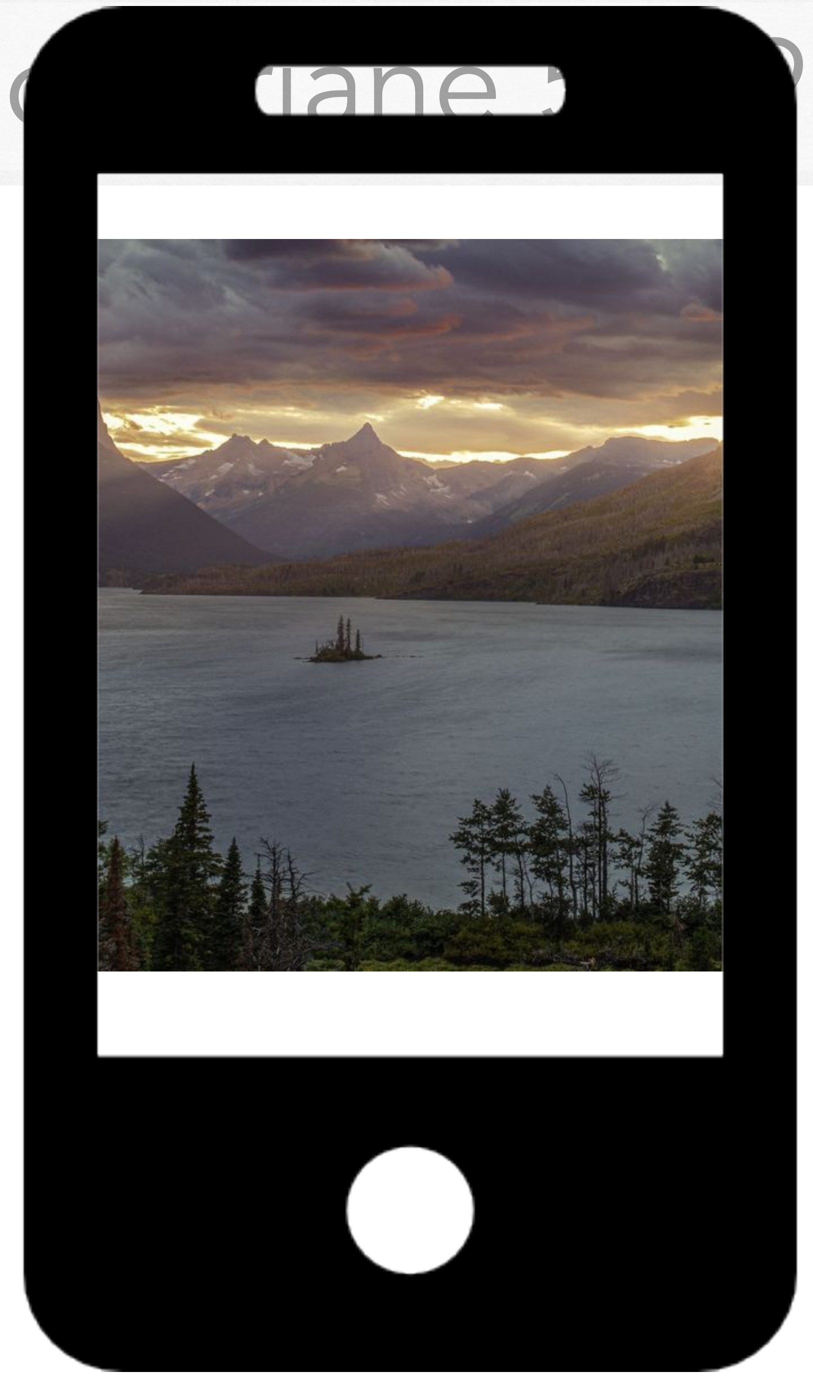
Gulf War: Loss of accuracy led to failure in US defense system, 28 killed!

April 1992, Schleswig-Holstein, Germany

Rounding error changed Parliament makeup!

June 1996

Overflow led to explosion o[illegible]iane [illegible]s after lift-off, $370 million lost!



May, 2020

Rounding error in luminance computation crashed Andriod phones

Finite-Precision Errors in Real World

February 1991, Dhahran, Saudi Arabia

Gulf War: Loss of accuracy led to failure in US defense system, 28 killed!

April 1992, Schleswig-Holstein, Germany

Rounding error changed Parliament makeup!

June 1996

Overflow led to explosion of Ariane 5, 39s after lift-off, $370 million lost!

…

May 2020

Rounding error in luminance computation crashed Andriod phones

How do we compute the errors?

Finite-Precision Accuracy Analysis

```
(x:Float32, y:Float32, z:Float32): Float32
def controller(x, y, z): = {
  val res = -x*y - 2*y*z - x - z
  return res
} ensuring (res +/- ?)
```

Finite-Precision Accuracy Analysis

```
(x:Float32, y:Float32, z:Float32): Float32
def controller(x, y, z): = {
  val res = -x*y - 2*y*z - x - z
  return res
} ensuring (res +/- ?)
```

compute a bound on the error

Finite-Precision Accuracy Analysis

```
(x:Float32, y:Float32, z:Float32): Float32
def controller(x, y, z): = {
  val res = -x*y - 2*y*z - x - z
  return res
} ensuring (res +/- ?)
```

absolute error:

$$\max_{x,y,z\in I} \mid f(x,y,z) - \tilde{f}(\tilde{x},\tilde{y},\tilde{z}) \mid$$

Finite-Precision Accuracy Analysis

$$\max_{x,y,z \in I} \mid f(x,y,z) - \tilde{f}(\tilde{x},\tilde{y},\tilde{z}) \mid$$

worst-case error analysis for small programs

Daisy FLUCTUAT Rosa

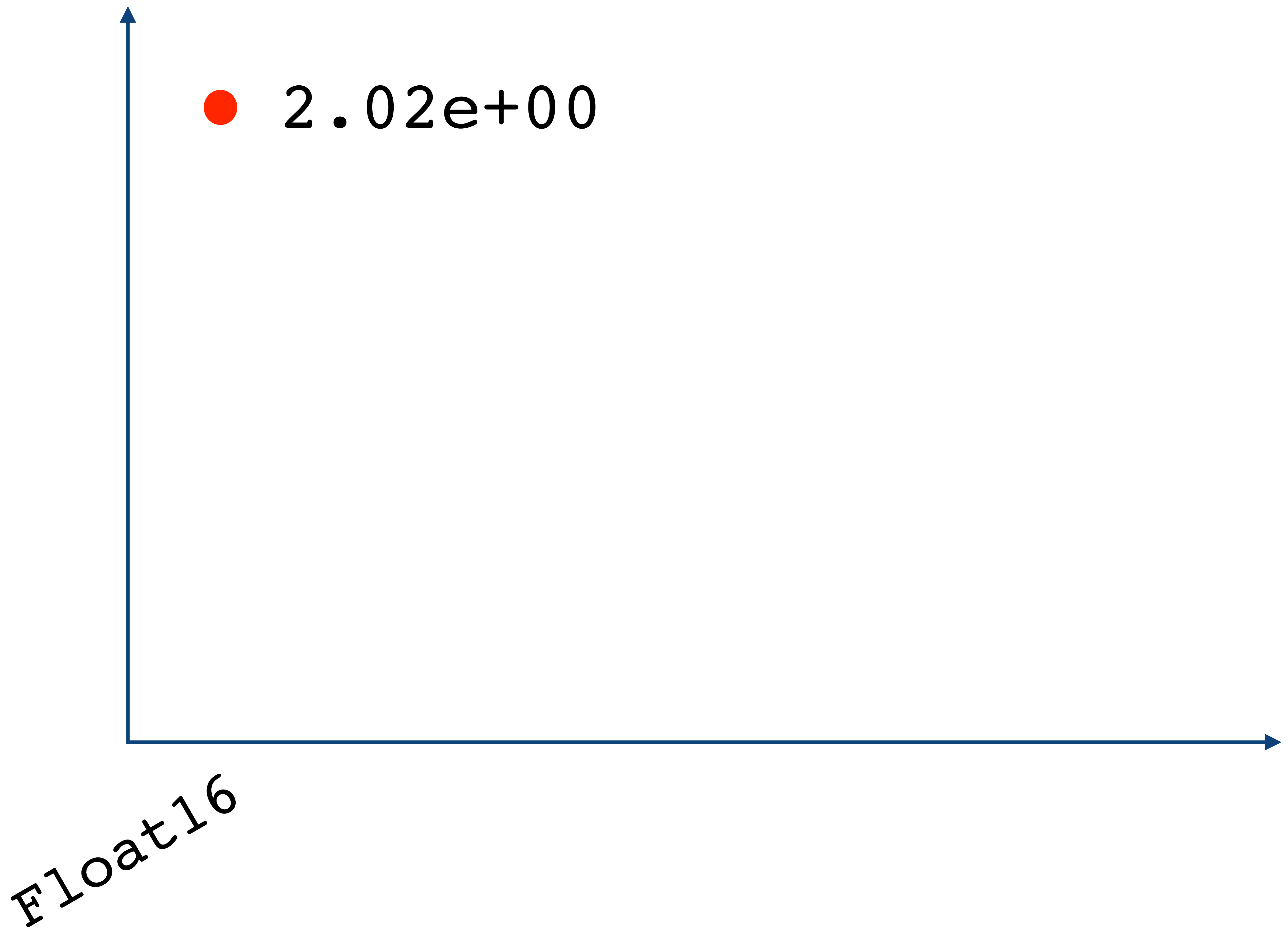
FPTaylor PRECiSA ⋯

Errors depend on Precision used

```
(x:Float16, y:Float16, z:Float16)
def controller(x, y, z): ___ = {
  val res = -x*y - 2*y*z - x - z
  return res
} ensuring (res +/- ?)
```

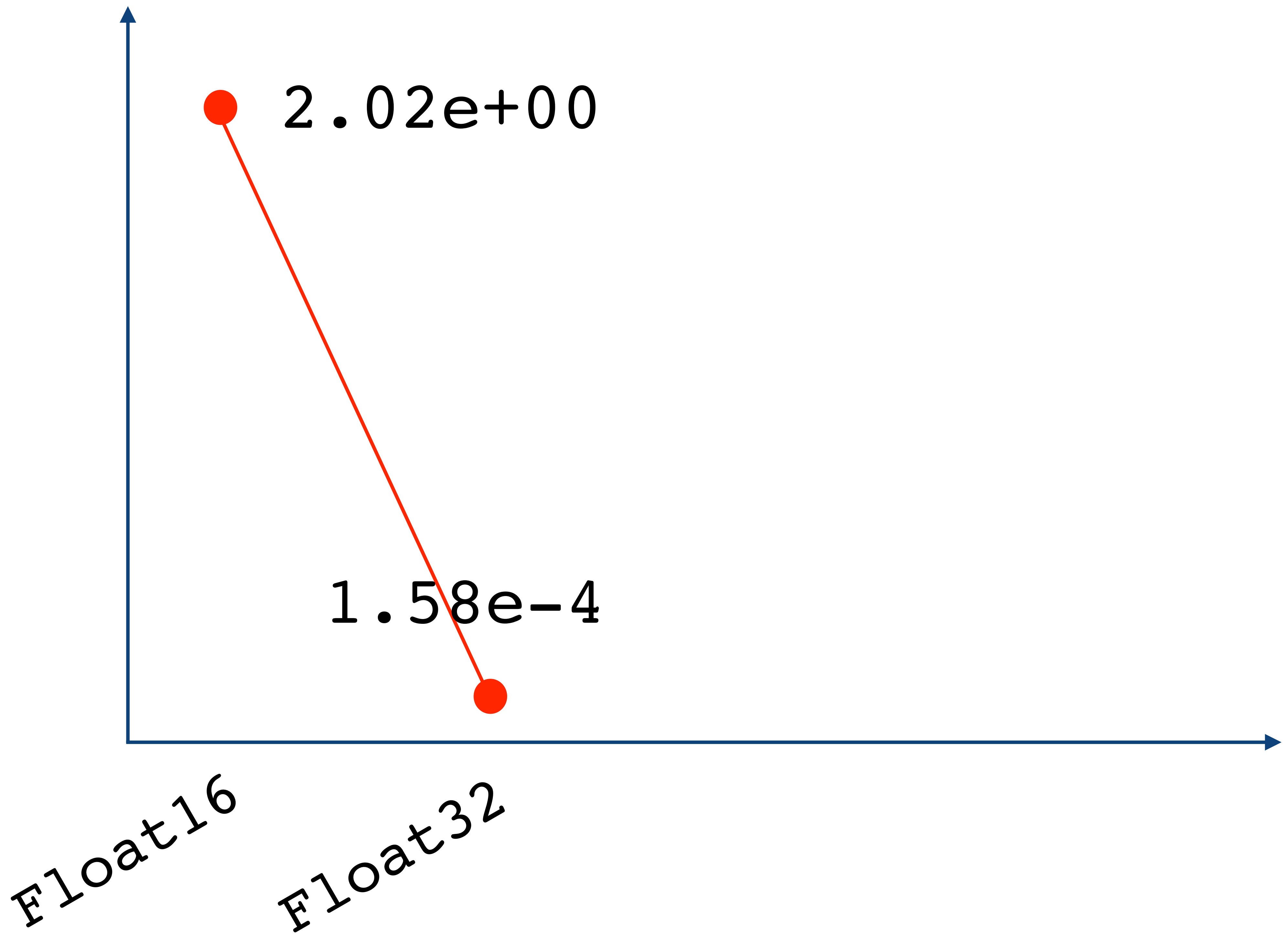
Errors depend on Precision used

```
(x:Float16, y:Float16, z:Float16)
def controller(x, y, z): ___ = {
  val res = -x*y - 2*y*z - x - z
  return res
} ensuring (res +/- ?)
```



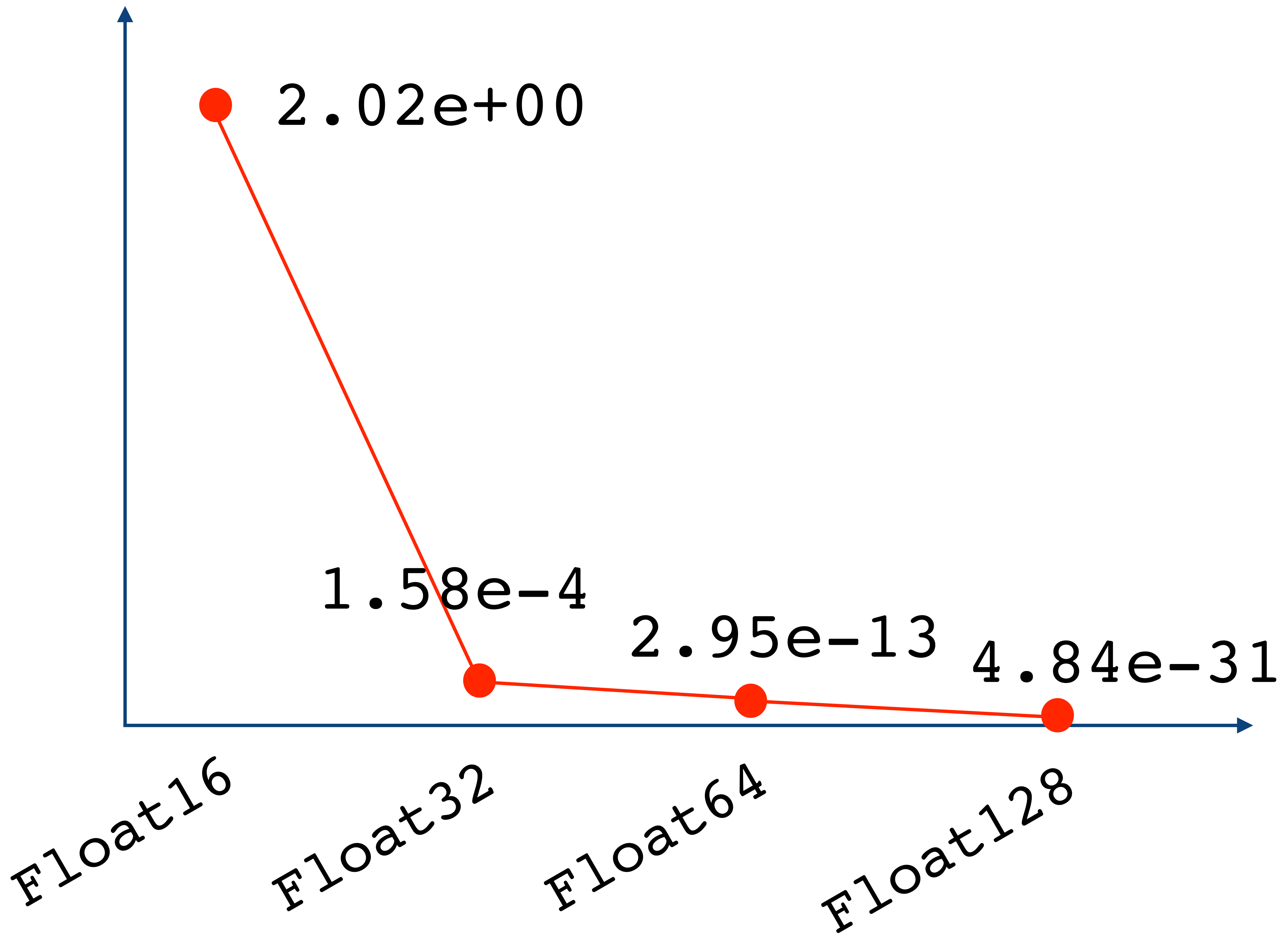
Errors depend on Precision used

```
(x:___, y:___, z:___)
def controller(x, y, z): ___ = {
  val res = -x*y - 2*y*z - x - z
  return res
} ensuring (res +/- ?)
```



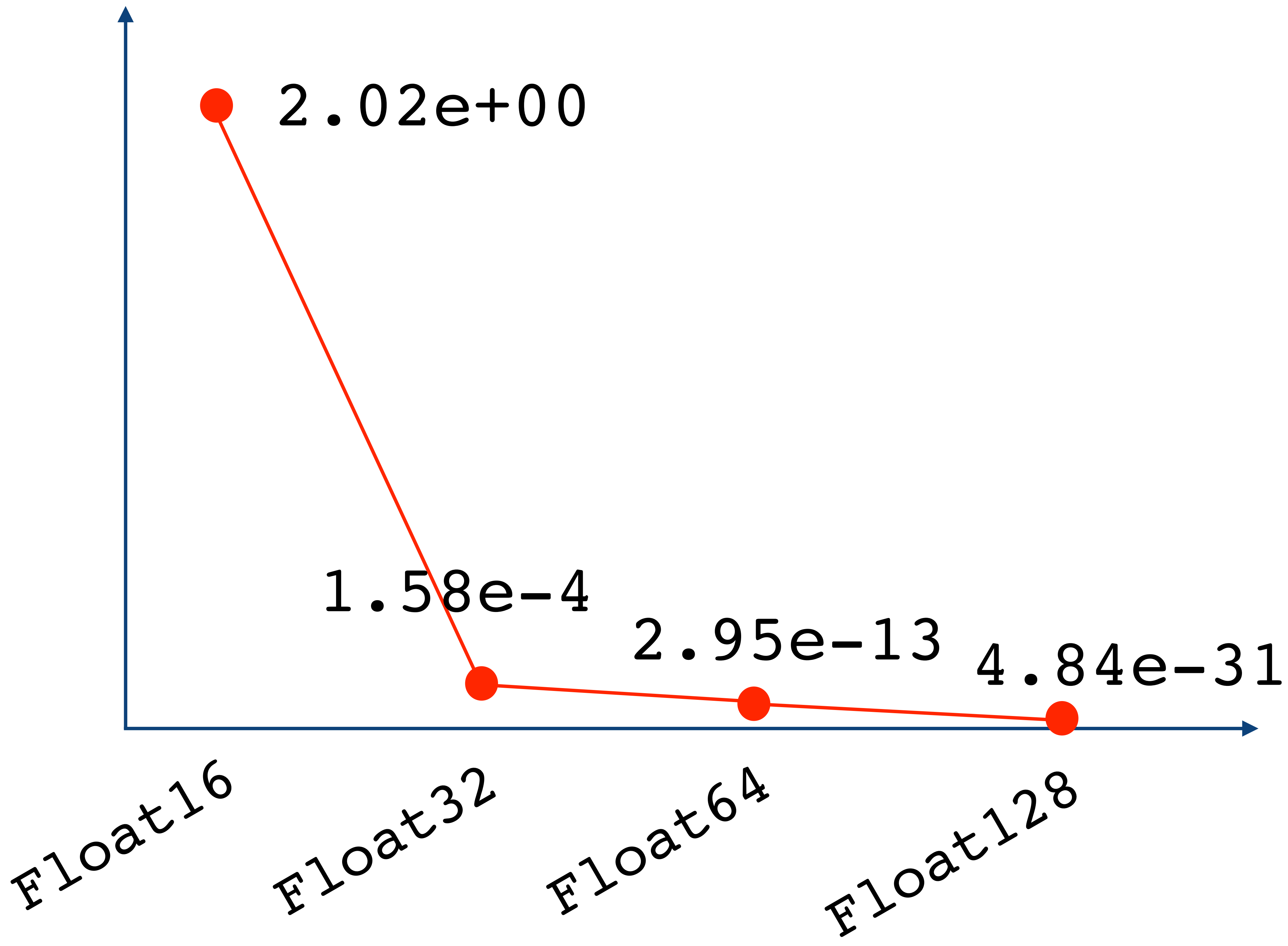
Errors depend on Precision used

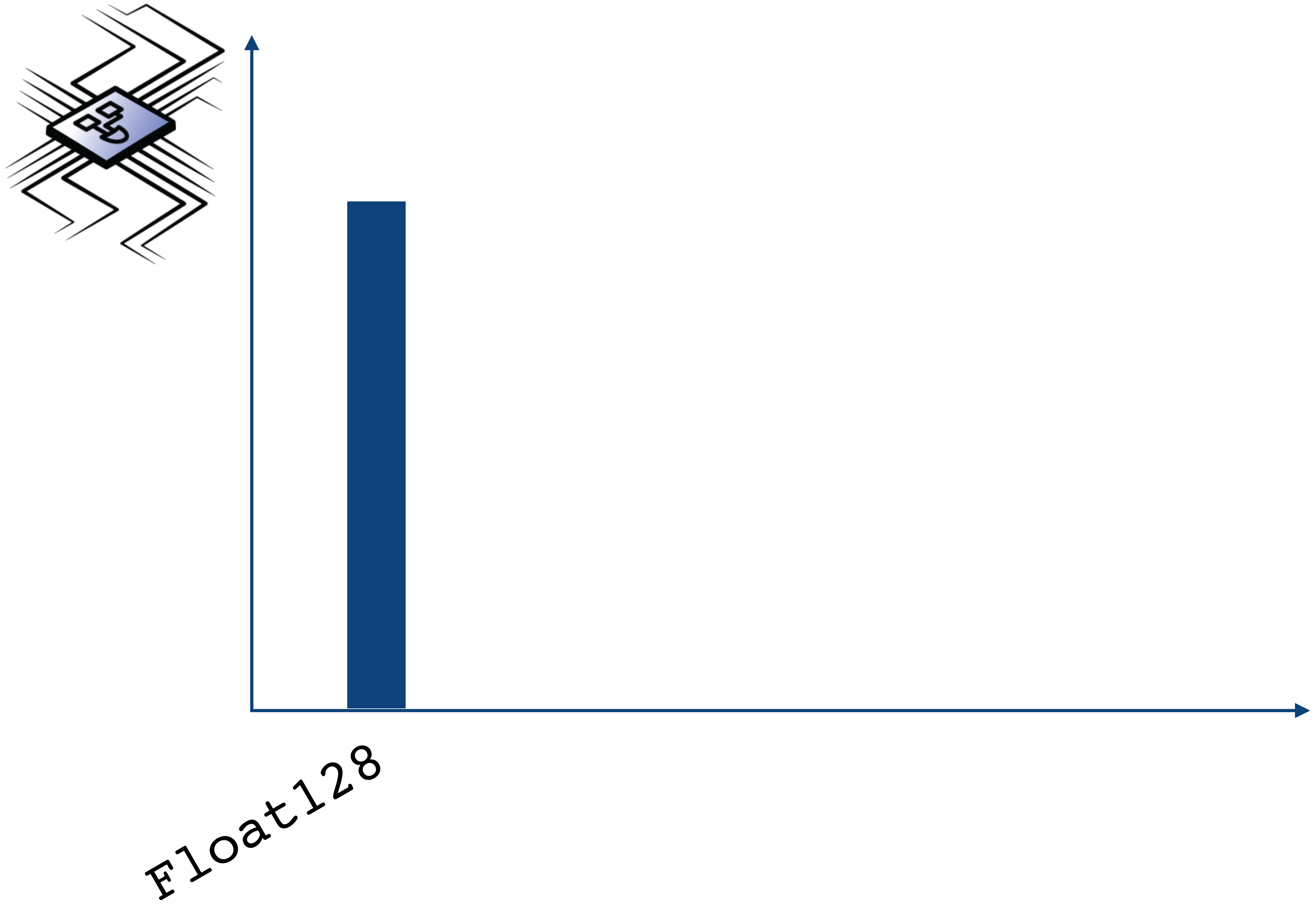
```
(x:___, y:___, z:___)
def controller(x, y, z): ___ = {
  val res = -x*y - 2*y*z - x - z
  return res
} ensuring (res +/- ?)
```



So are the Resource Costs!

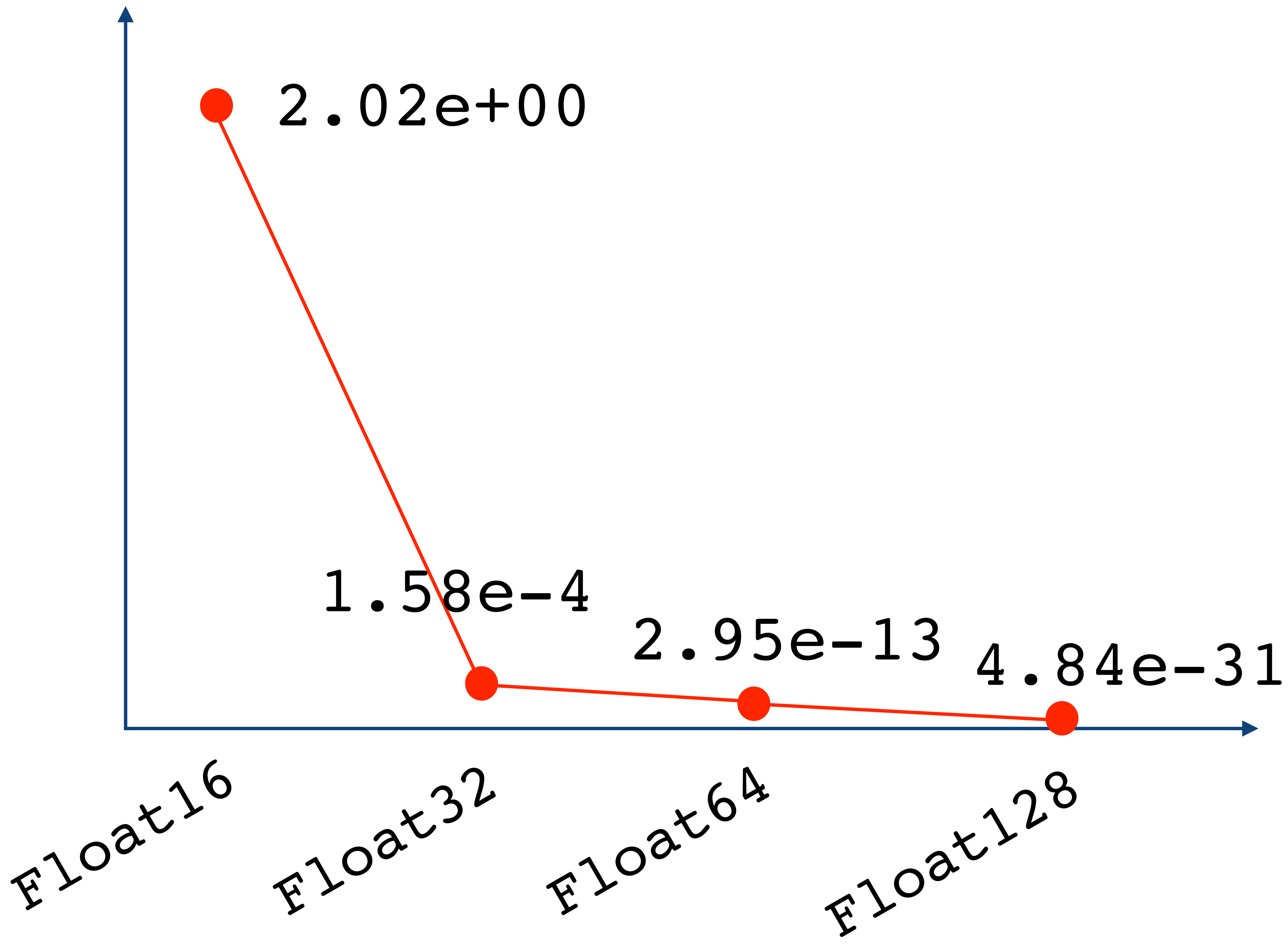
```
(x:___, y:___, z:___)
def controller(x, y, z): ___ = {
  val res = -x*y - 2*y*z - x - z
  return res
} ensuring (res +/- ?)
```

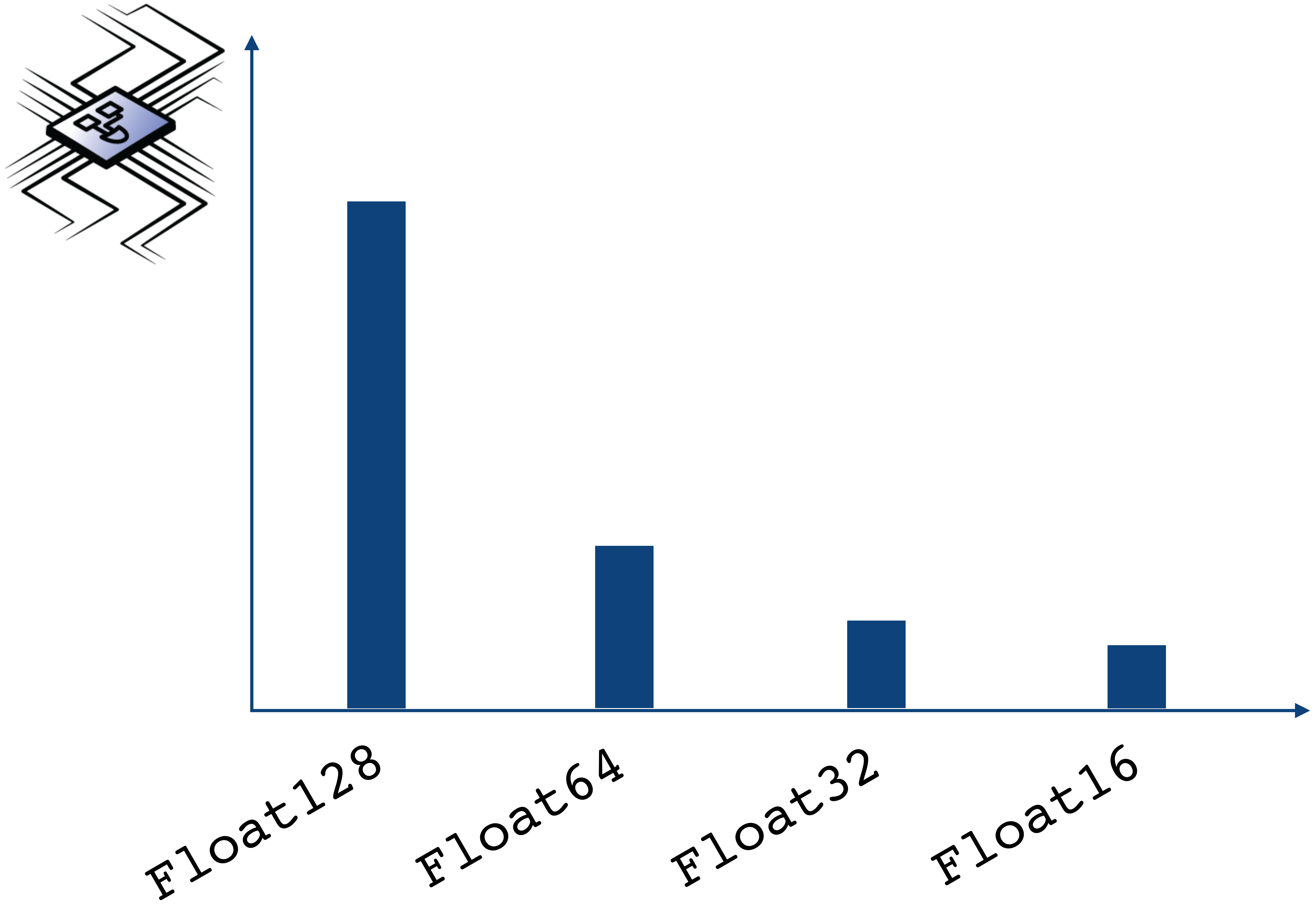




So are the Resource Costs!

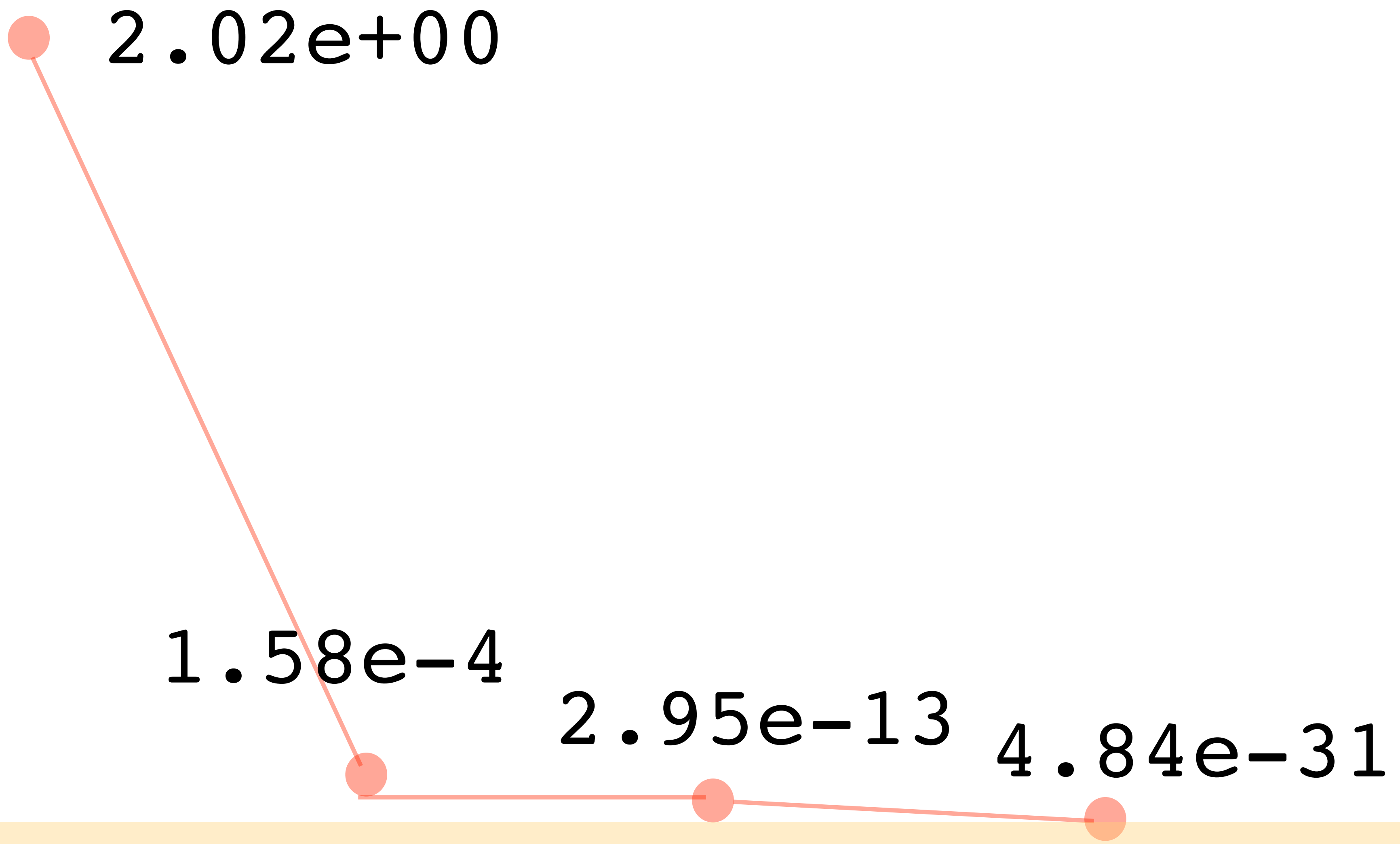
```
(x:↑___, y:↑___, z:↑___)
def controller(x, y, z): ___ = {
  val res = -x*y - 2*y*z - x - z
  return res
} ensuring (res +/- ?)
```

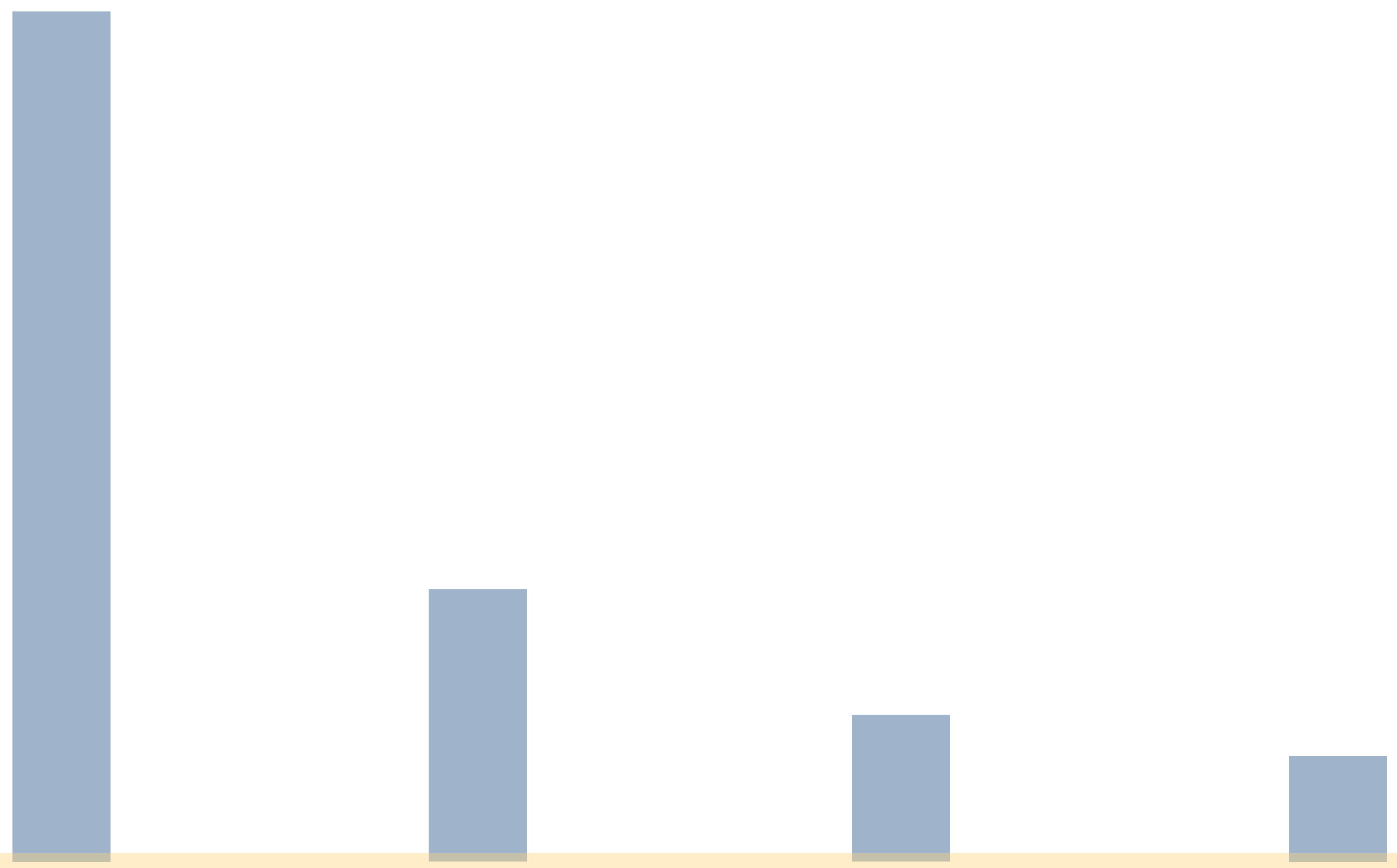




So are the Resource Costs!

```
(x:__, y:__, z:__)
def controller(x, y, z): ___ = {
  val res = -x*y - 2*y*z - x - z
  return res
} ensuring (res +/- ?)
```





We need to find a tradeoff between accuracy and resources!

Finite-Precision Optimization

```
(x:Float32, y:Float32, z:Float32)
 def controller(x, y, z): ___ = {
   val res = -x*y - 2*y*z - x - z
   return res
 } ensuring (res +/- ?)
```

```
def controller(x: ?, y: ?, z: ?): ? = {
  val res = -x*y - 2*y*z - x - z
  return res
} ensuring res +/- 0.00197
```

find the lowest precision satisfying error bound

Finite-Precision Optimization

```
(x:Float32, y:Float32, z:Float32)
 def controller(x, y, z): ___ = {
   val res = -x*y - 2*y*z - x - z
   return res
 } ensuring (res +/- ?)
```

```
def controller(x: ?, y: ?, z: ?): ? = {
  val res = -x*y - 2*y*z - x - z
  return res
} ensuring res +/- 0.00197
```

mixed-precision optimization

- minimize resource cost still satisfying the error
- assign different precisions to different variables

Finite-Precision Optimization

```
(x:Float32, y:Float32, z:Float32)
 def controller(x, y, z): ___ = {
   val res = -x*y - 2*y*z - x - z
   return res
 } ensuring (res +/- ?)
```

```
def controller(x: ?, y: ?, z: ?): ? = {
  val res = -x*y - 2*y*z - x - z
  return res
} ensuring res +/- 0.00197
```

- minimize resource cost still satisfying the error
- assign different precisions to different variables

worst-case tuning for small (floating-point) programs

Daisy FPTuner

The Horizons of Finite-Precision Analysis

Accuracy Analysis

worst-case error analysis for small programs

Daisy FLUCTUAT Rosa
FPTaylor PRECiSA …

Optimization

worst-case tuning for small (floating-point) programs

Daisy FPTuner

Our Work: Extending the Horizon of Finite-Precision Analysis

Accuracy Analysis

considering probability distribution of inputs

iFM '19 EMSOFT '18

Probabilistic Analysis

~~worst-case error~~ analysis for small programs

Daisy FLUCTUAT Rosa

FPTaylor PRECiSA ···

Optimization

worst-case tuning for small (floating-point) programs

Daisy FPTuner

Our Work: Extending the Horizon of Finite-Precision Analysis

Accuracy Analysis

handling larger programs

TACAS '21

Static + Dynamic Analysis

iFM '19 EMSOFT '18

Probabilistic Analysis

worst-case error analysis for ~~small programs~~

Daisy FLUCTUAT Rosa

FPTaylor PRECiSA …

Optimization

worst-case tuning for small (floating-point) programs

Daisy FPTuner

Our Work: Extending the Horizon of Finite-Precision Analysis

Accuracy Analysis

iFM '19 EMSOFT '18

Probabilistic Analysis

TACAS '21

Static + Dynamic Analysis

worst-case error analysis for small programs

Daisy FLUCTUAT Rosa

FPTaylor PRECiSA …

Optimization

specializing mixed fixed tuning for NNs

EMSOFT '23

NN Quantization

worst-case tuning for ~~small (floating-point) programs~~

Daisy FPTuner

Today's Talk: Probabilistic Error Analysis and NN Quantization

Accuracy Analysis

iFM '19 EMSOFT '18

Probabilistic Analysis

TACAS '21

Static + Dynamic Analysis

~~worst-case error~~ analysis for small programs

Daisy FLUCTUAT Rosa

FPTaylor PRECiSA …

Optimization

EMSOFT '23

NN Quantization

worst-case tuning for ~~small (floating-point) programs~~

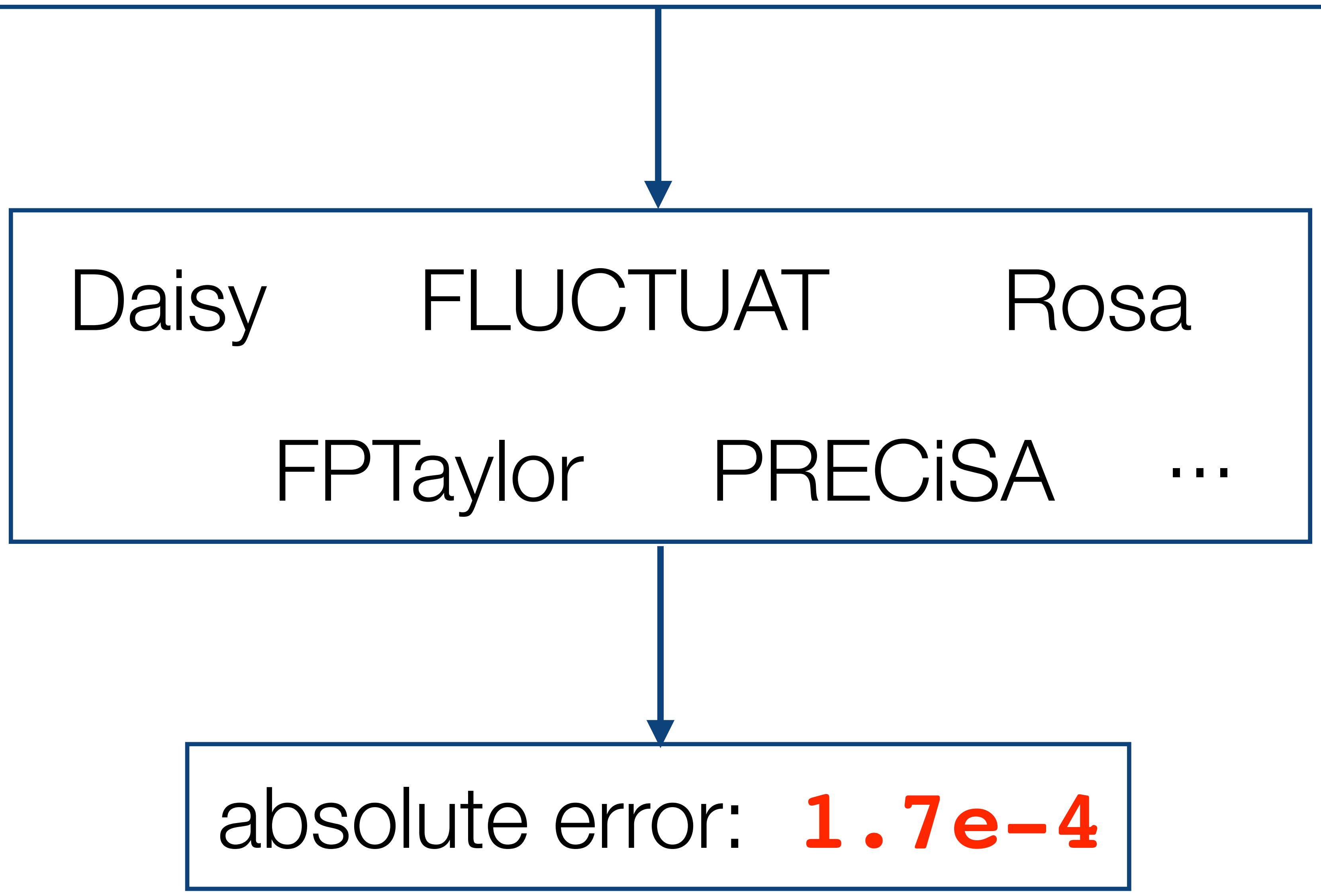
Daisy FPTuner

Probabilistic Roundoff Error Analysis

How do we take into account uncertainties in the inputs and compute the distribution of errors at the output?

State-of-the-Art: Worst-Case Error Analysis

```
def controller(x:Float32, y:Float32, z:Float32): Float32 = {
  require (-15.0 <= x, y, z <= 15.0)
  val res = -x*y - 2*y*z - x - z
    return res
} ensuring (res +/- ?)
```



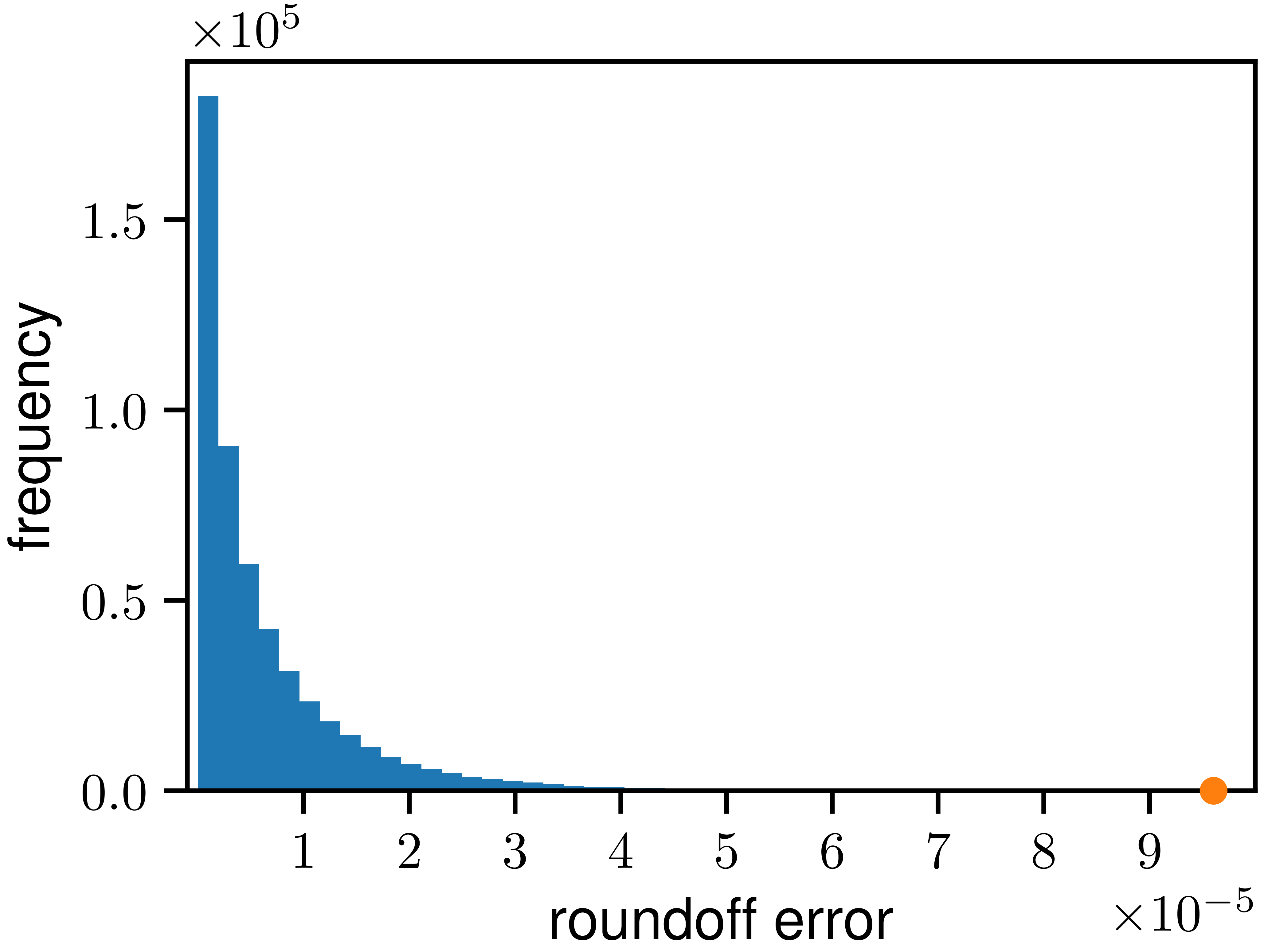
Daisy FLUCTUAT Rosa

FPTaylor PRECiSA …

absolute error: 1.7e-4

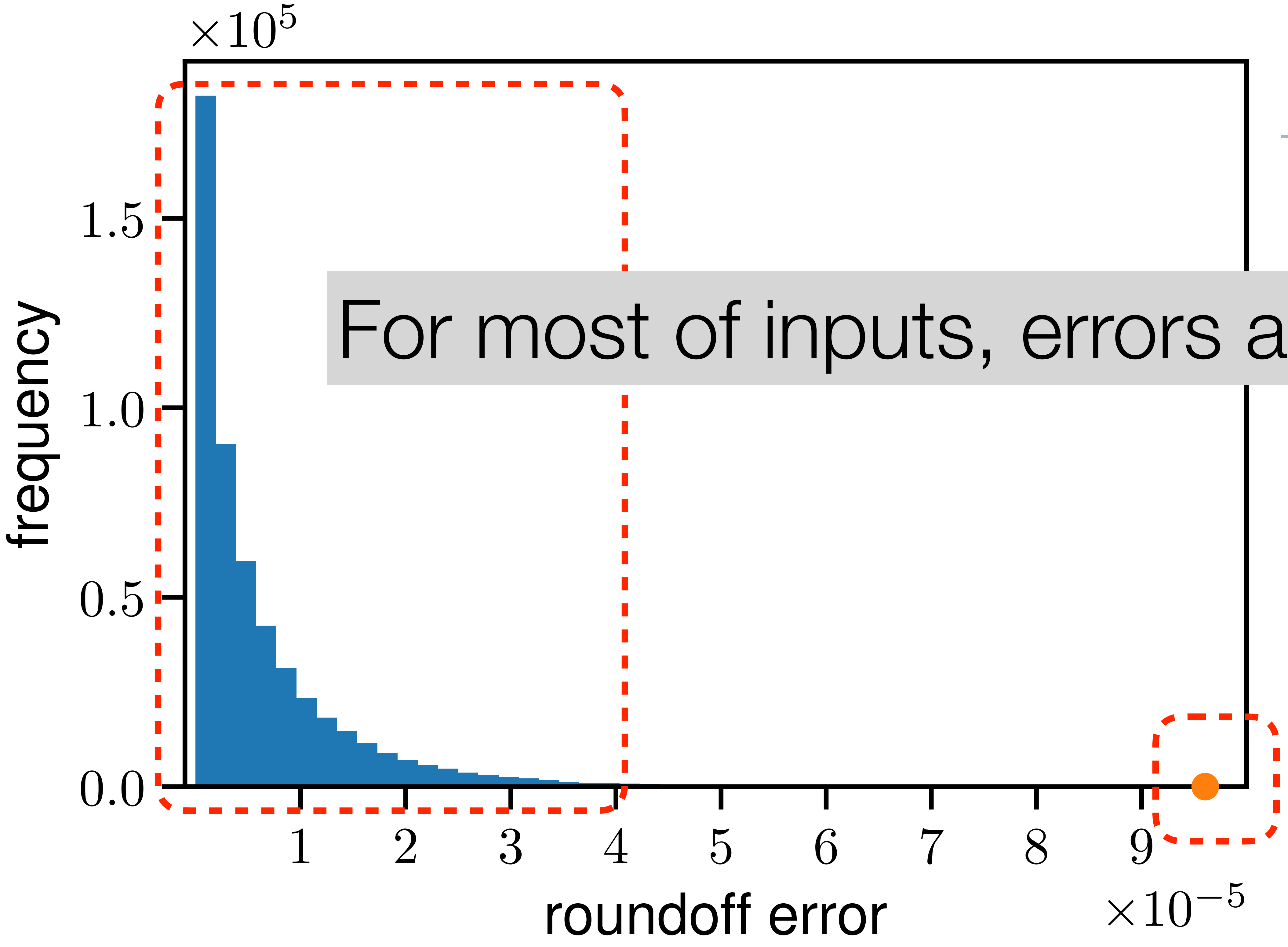
Worst-case can be pessimistic!

```
def controller(x:Float32, y:Float32, z:Float32): Float32 = {
 require (-15.0 <= x, y, z <= 15.0)
 val res = -x*y - 2*y*z - x - z
   return res
} ensuring (res +/- ?)
```



Worst-case can be pessimistic!

```
def controller(x:Float32, y:Float32, z:Float32): Float32 = {
  require (-15.0 <= x, y, z <= 15.0)
  val res = -x*y - 2*y*z - x - z
    return res
} ensuring (res +/- ?)
```



For most of inputs, errors are small!

Scenario 1: Applications may tolerate large infrequent errors

```
def controller(x:Float32, y:Float32, z:Float32): Float32 = {
 require (-15.0 <= x, y, z <= 15.0)
 val res = -x*y - 2*y*z - x - z
   return res
}ensuring (error <= 1.5e-4, 0.85)
```

tolerates big errors occurring with <= **0.15** probability

Scenario 1: Applications may tolerate large infrequent errors

```
(x:Float64, y:Float64, z:Float64): Float64
```

```
def controller(x:~~Float32, y:Float32, z:Float32): Float32~~ = {
 require (-15.0 <= x, y, z <= 15.0)
 val res = -x*y - 2*y*z - x - z
   return res
}ensuring (error <= 1.5e-4, 0.85)
```

worst-case error: 1.7e-4

tolerates big errors occurring with <= **0.15** probability

Scenario 1: Applications may tolerate large infrequent errors

```
(x:Float64, y:Float64, z:Float64): Float64
```

```
def controller(x:~~Float32, y:Float32, z:Float32): Float32~~ = {
 require (-15.0 <= x, y, z <= 15.0)
 val res = -x*y - 2*y*z - x - z
   return res
}ensuring (error <= 1.5e-4, 0.85)
```

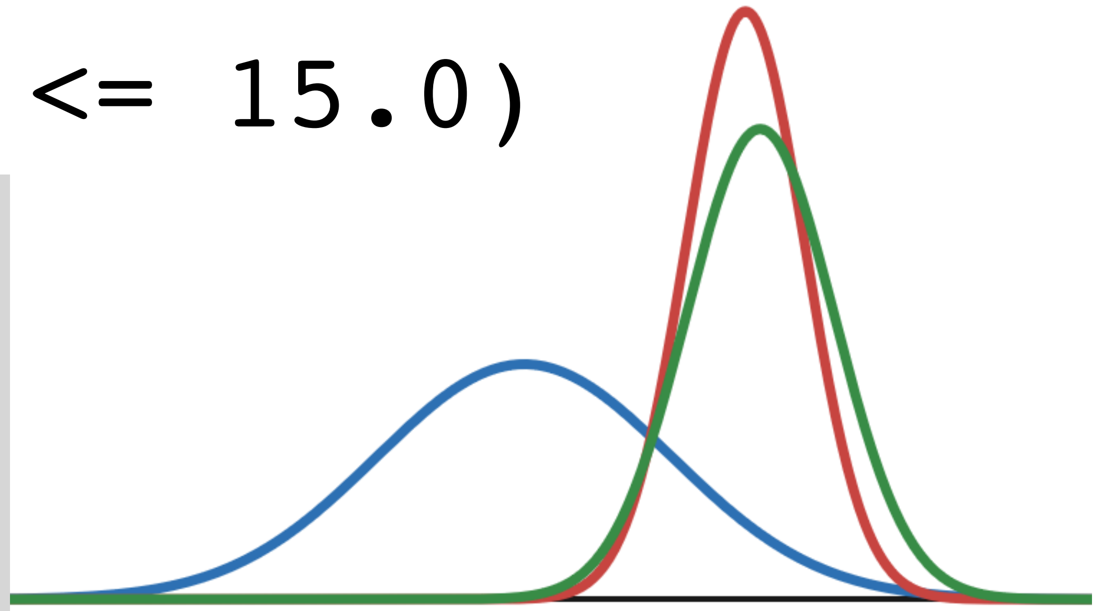
worst-case error: 1.7e-4

tolerates big errors occurring with <= **0.15** probability

We need to analyze roundoff errors probabilistically!

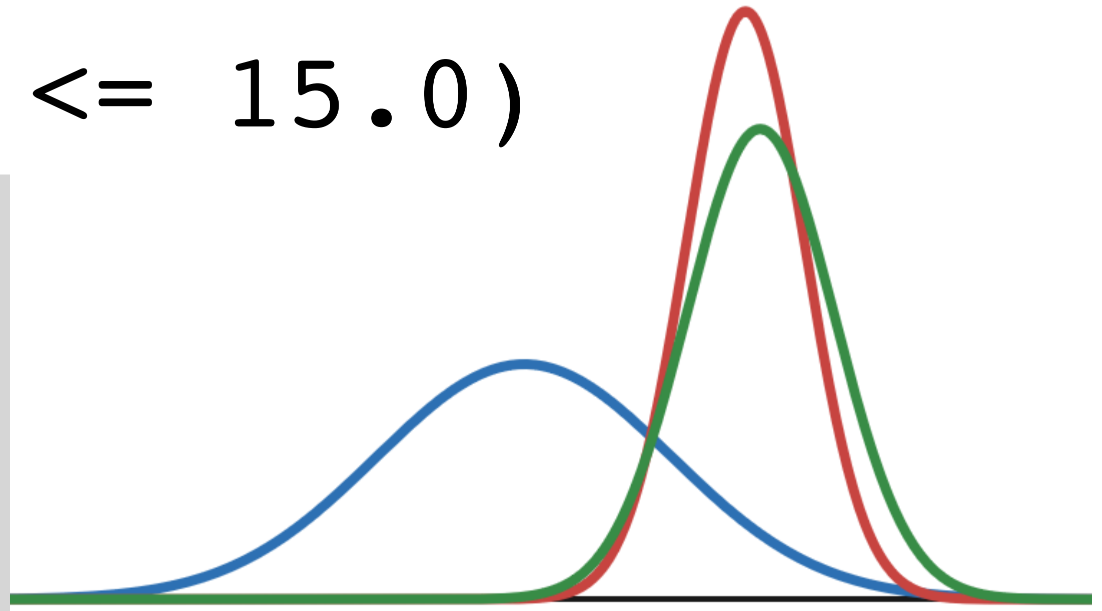
Our Contribution: Probabilistic Analysis for Roundoff Errors

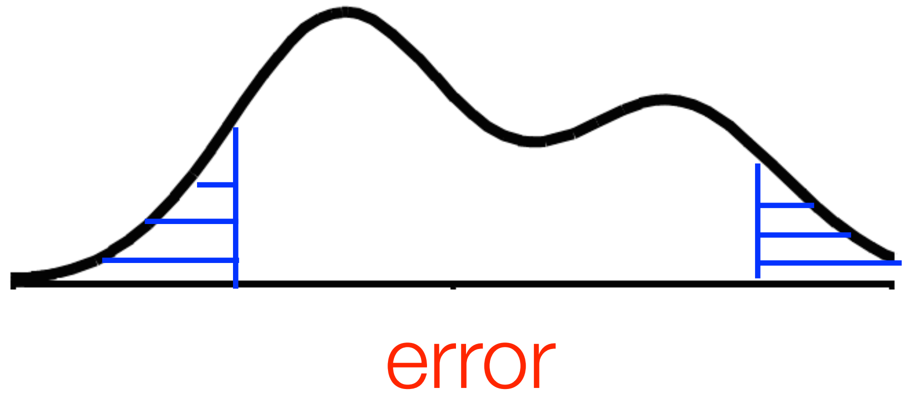
```
def controller(x:Float32, y:Float32, z:Float32): Float32 = {
require (-15.0 <= x, y, z <= 15.0)

 x:= gaussian(4.0, 0.5)
 y:= gaussian(4.75, 0.2)
 z:= gaussian(4.8, 0.25)

 val res = -x*y - 2*y*z - x - z
   return res
} ensuring (error <= 1.5e-4, 0.85)
```



Our Contribution: Probabilistic Analysis for Roundoff Errors

```
def controller(x:Float32, y:Float32, z:Float32): Float32 = {
require (-15.0 <= x, y, z <= 15.0)

 x:= gaussian(4.0, 0.5)
 y:= gaussian(4.75, 0.2)
 z:= gaussian(4.8, 0.25)

  val res = -x*y - 2*y*z - x - z
    return res
} ensuring (error <= 1.5e-4, 0.85)
```

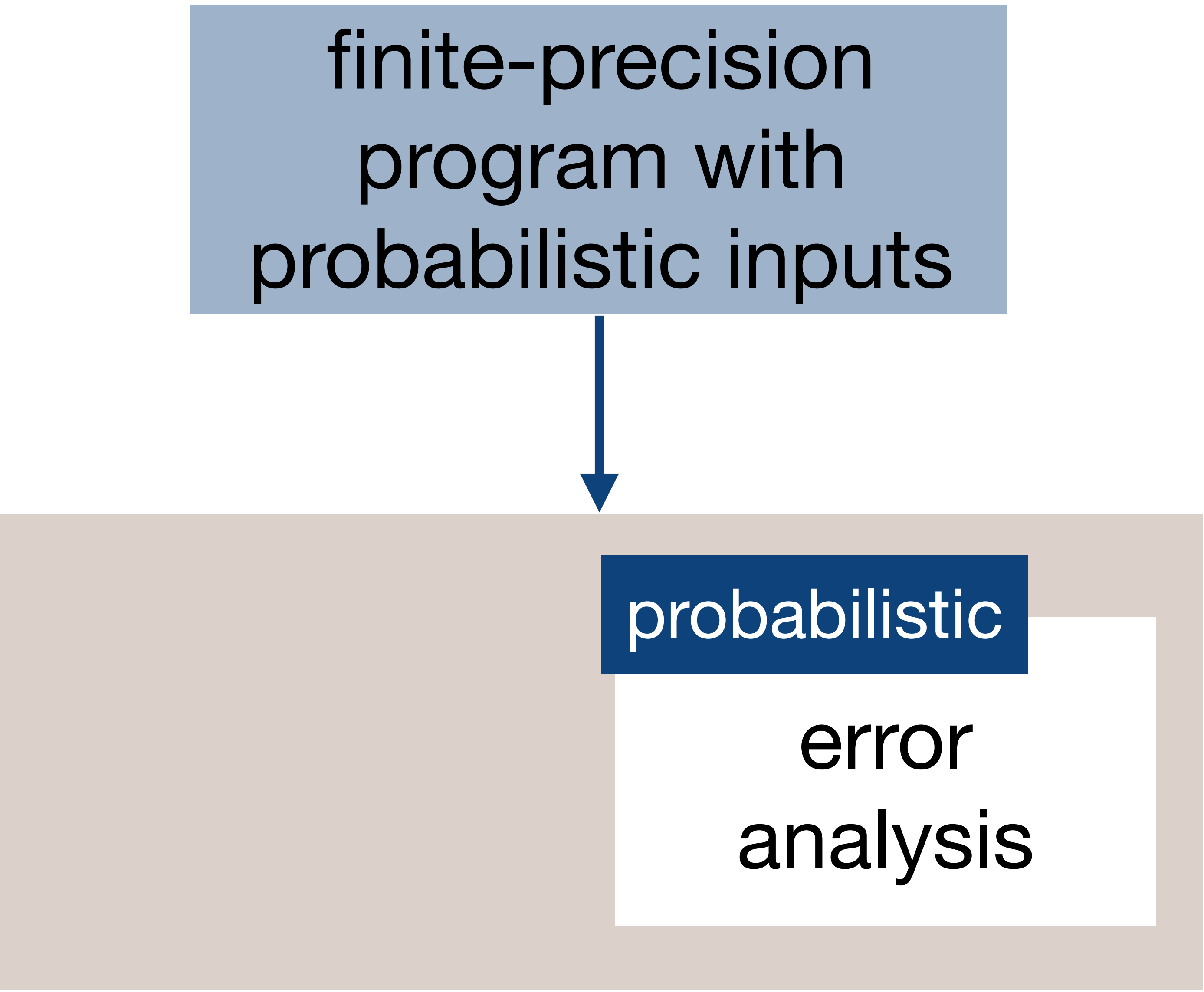




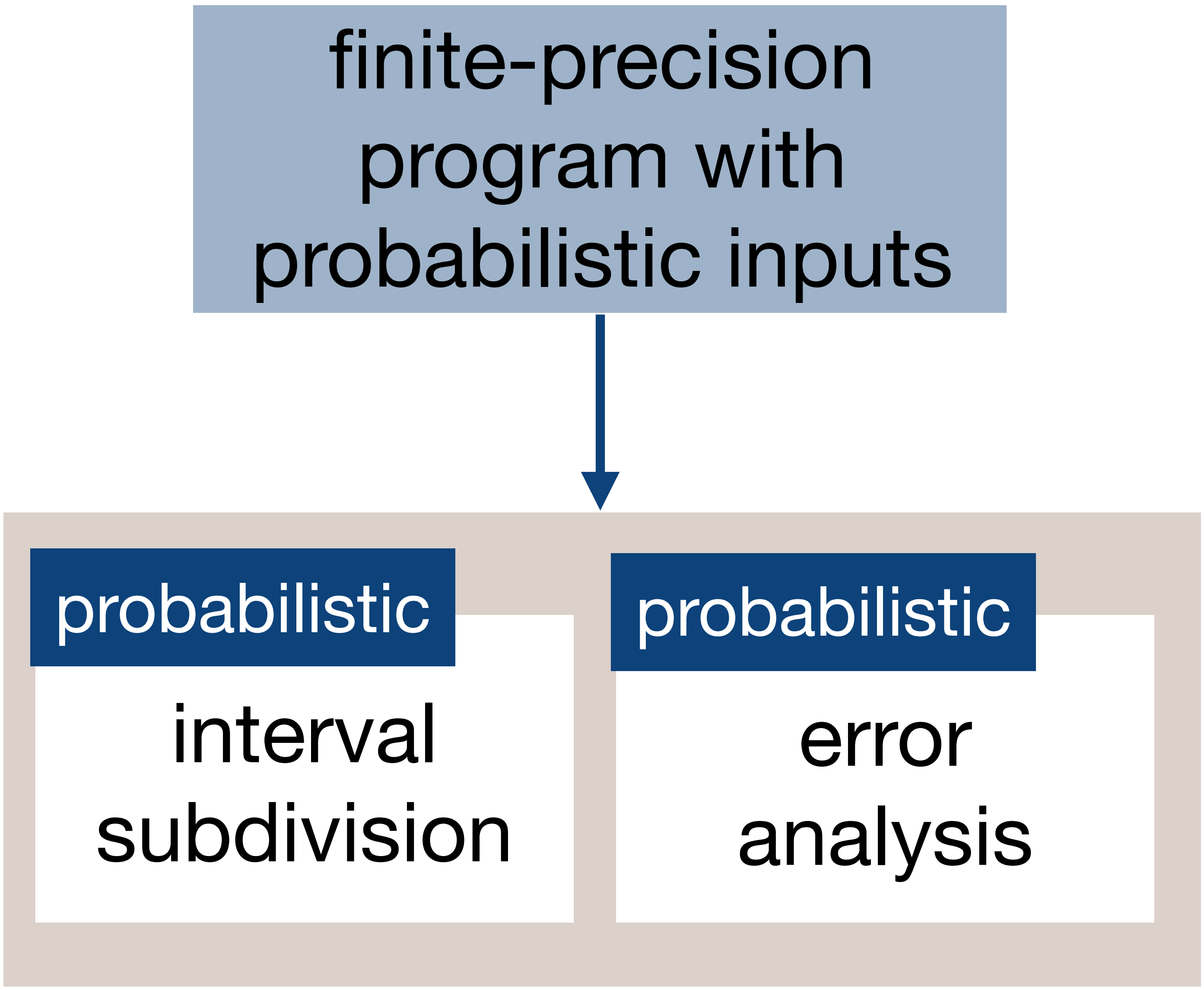
✓ probability distribution of errors

✓ a refined error that occurs with the threshold probability

Overview: Sound Probabilistic Roundoff Error Analysis



Overview: Sound Probabilistic Roundoff Error Analysis

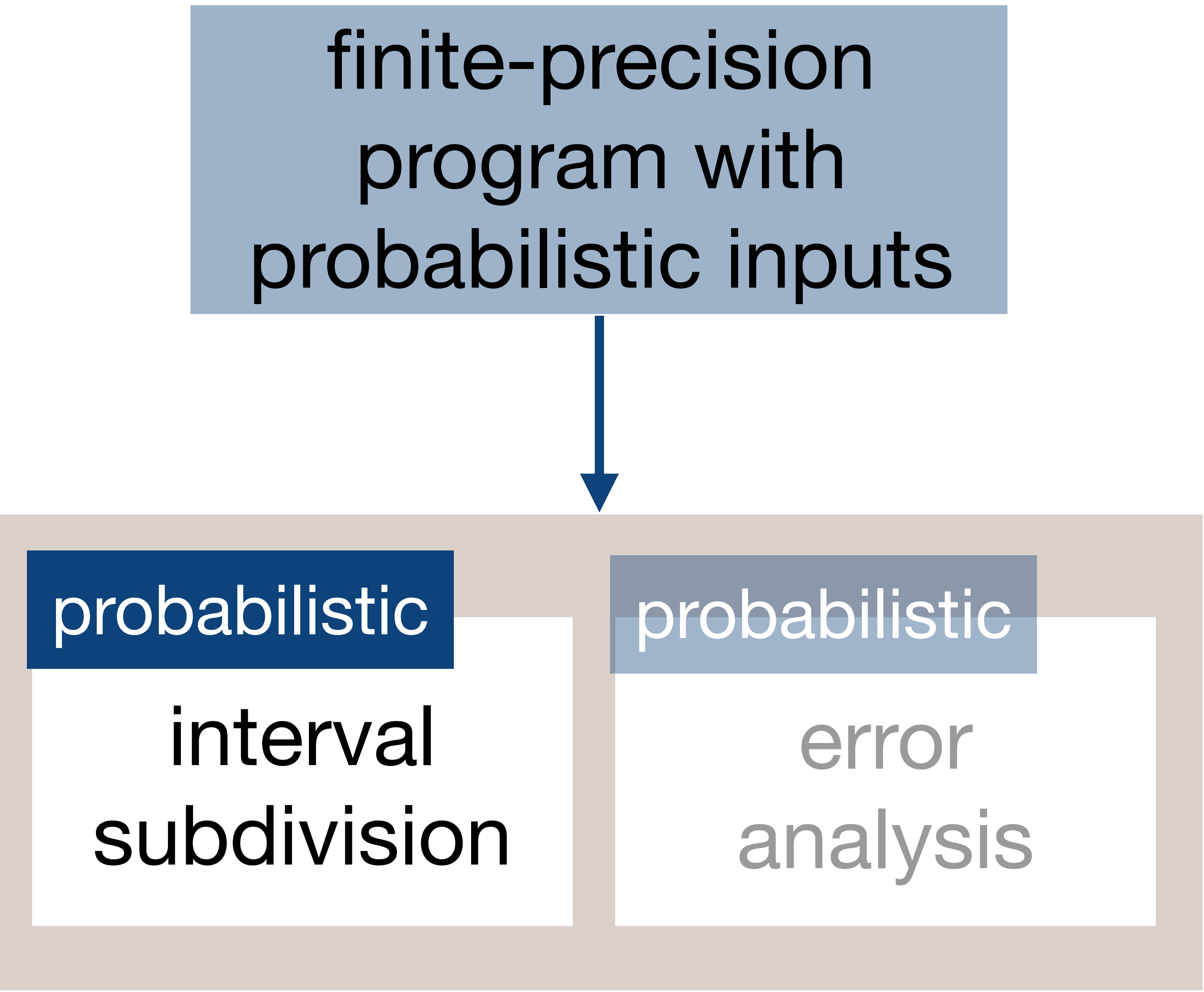


finite-precision program with probabilistic inputs

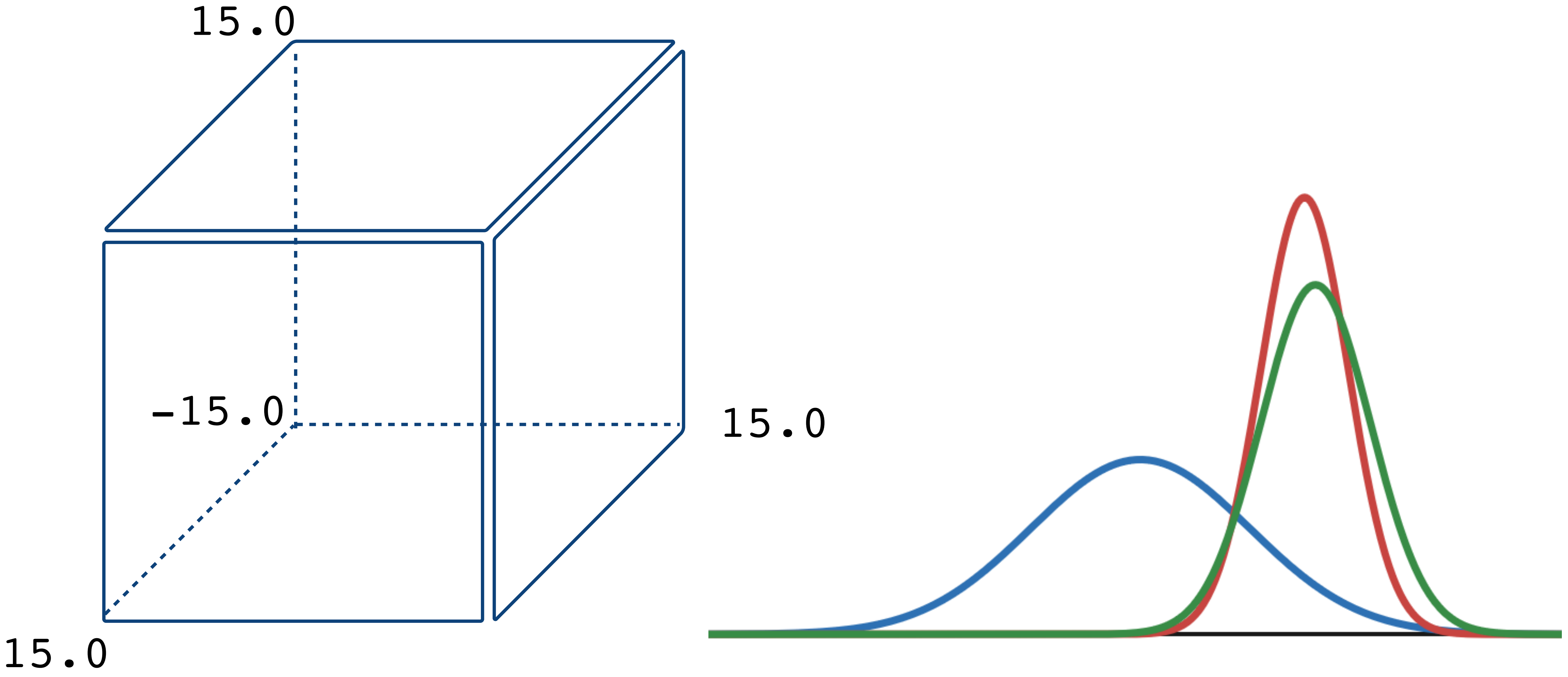
probabilistic interval subdivision

probabilistic error analysis

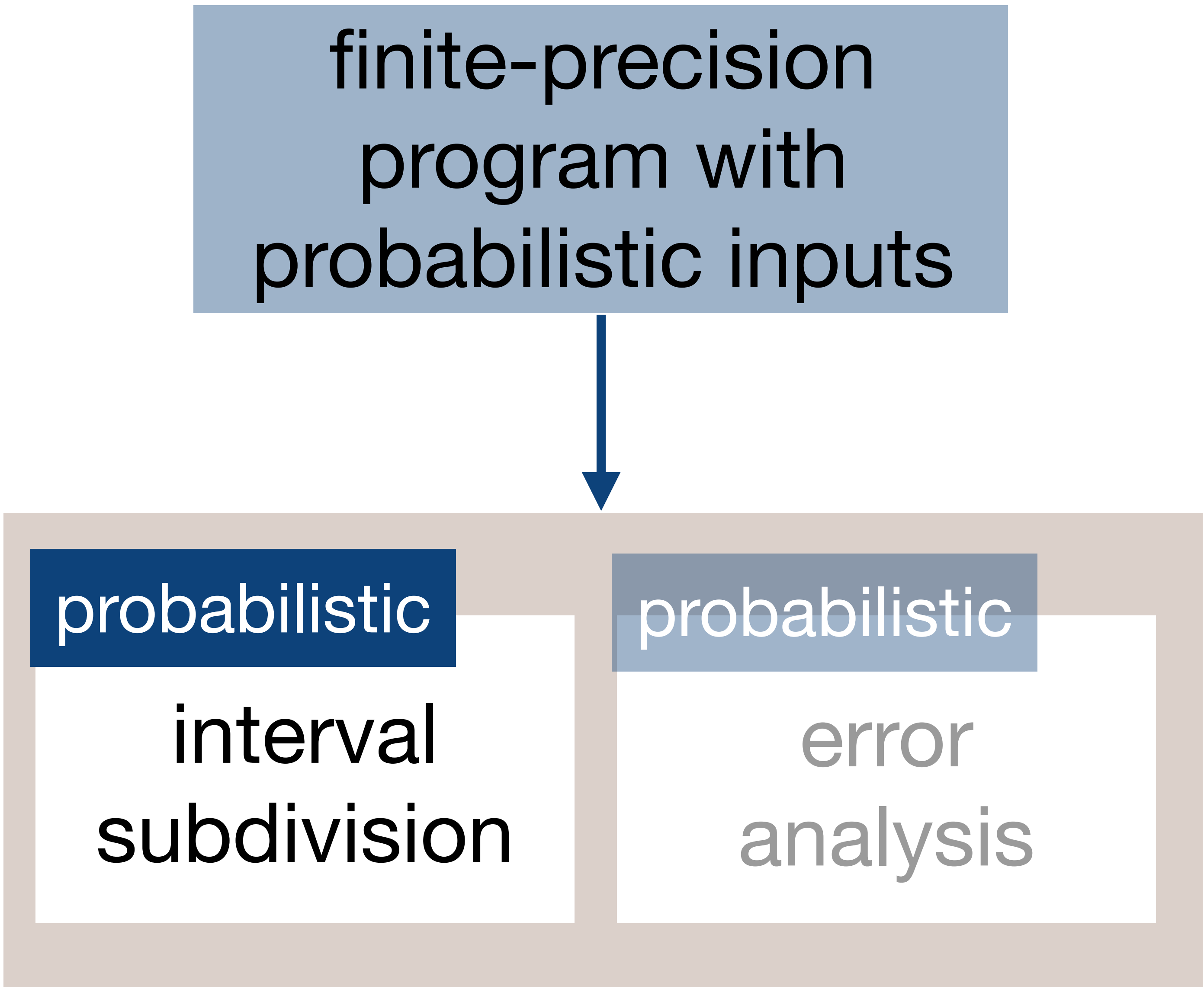
Probabilistic Interval Subdivision



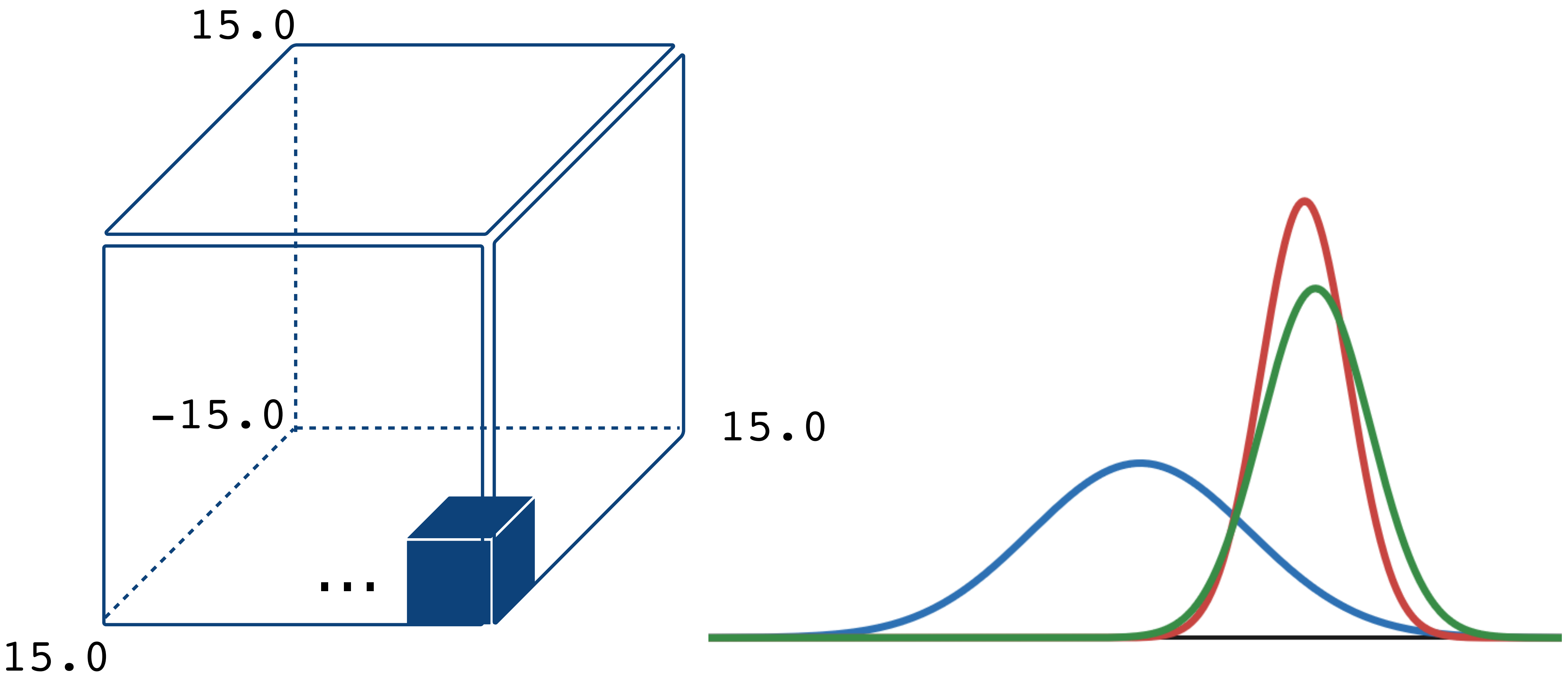
```
require (-15.0 <= x, y, z <= 15.0)
```



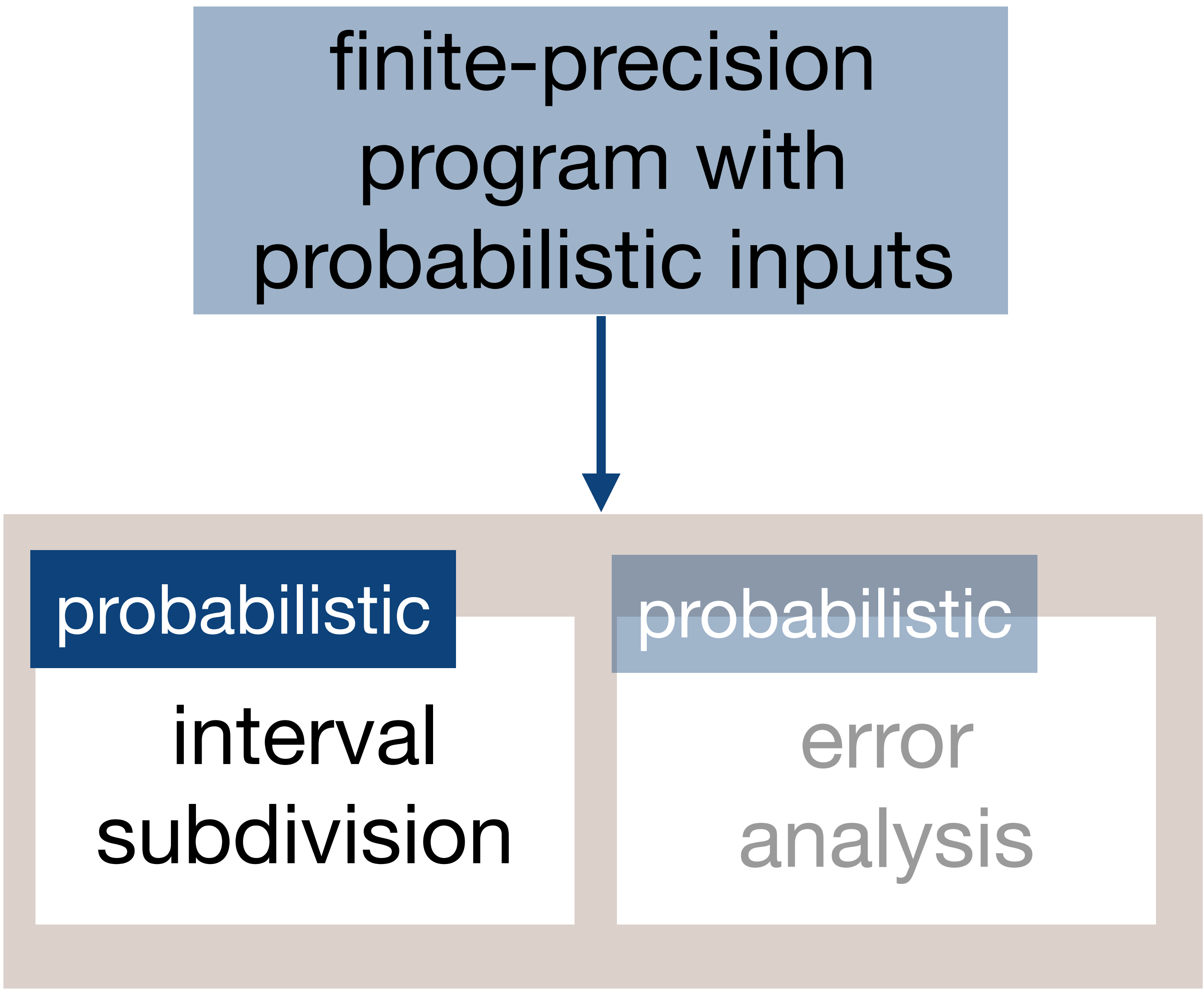
Probabilistic Interval Subdivision



```
require (-15.0 <= x, y, z <= 15.0)
```



Probabilistic Interval Subdivision

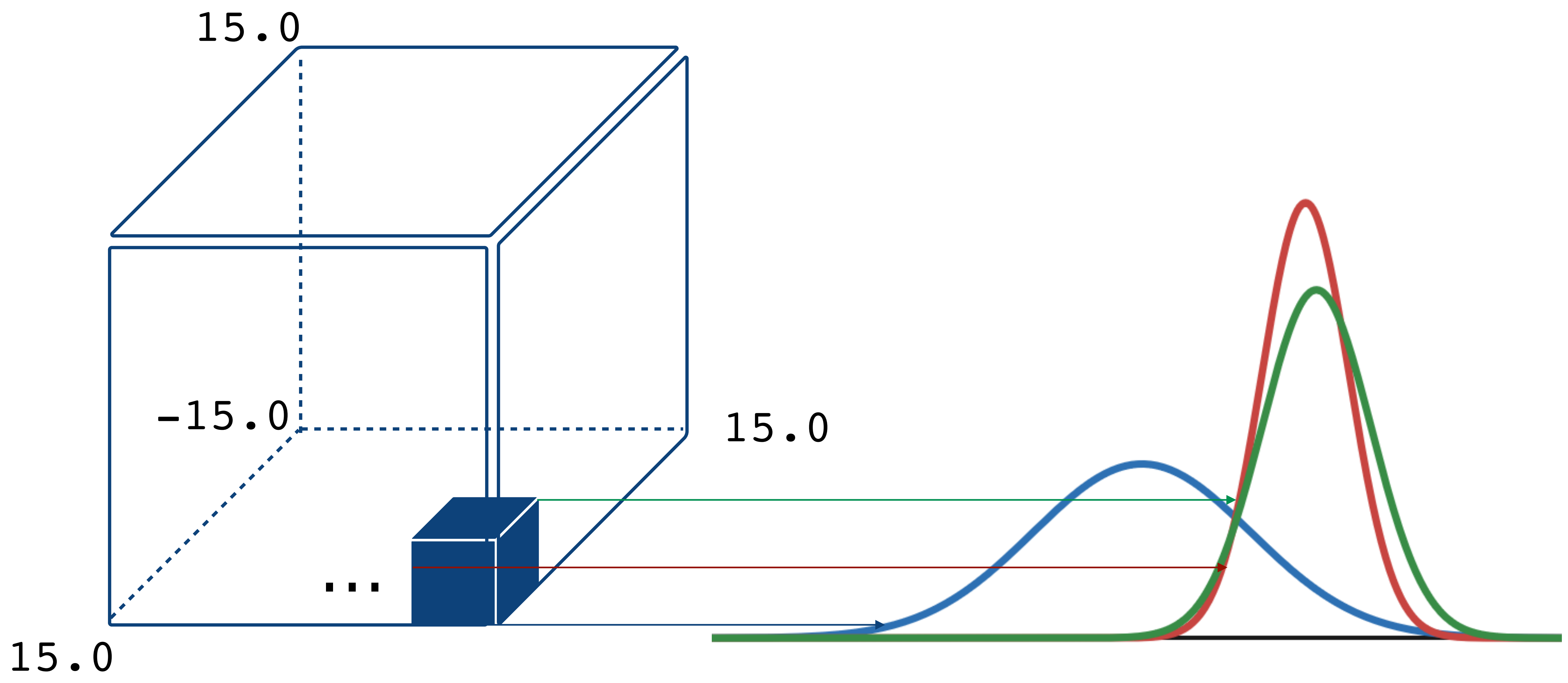


finite-precision program with probabilistic inputs

probabilistic interval subdivision

probabilistic error analysis

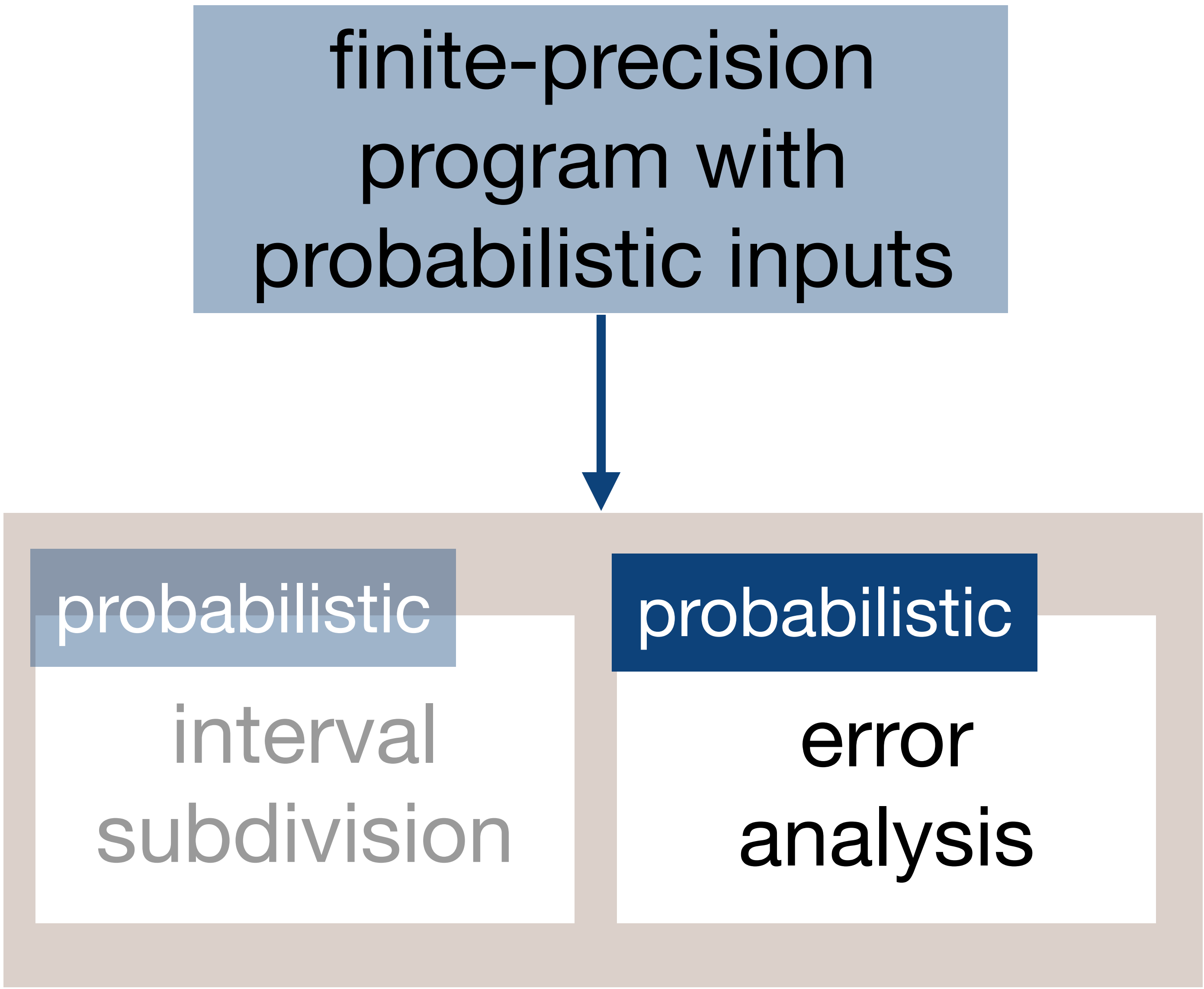
```
require (-15.0 <= x, y, z <= 15.0)
```



subdomain with a probability taking Cartesian Product:

$$\forall i \in x, \forall j \in y, \forall k \in z, p_{ijk} = x_i \times y_j \times z_k$$

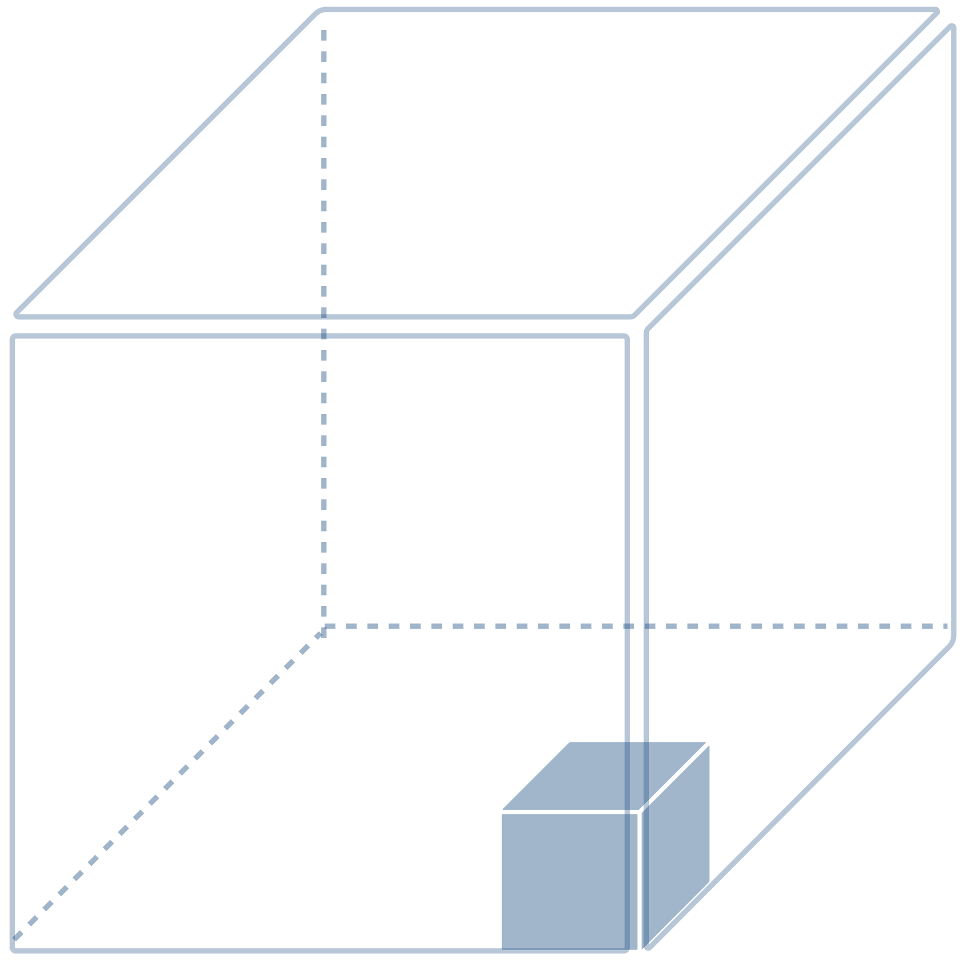
Probabilistic Error Analysis



finite-precision program with probabilistic inputs

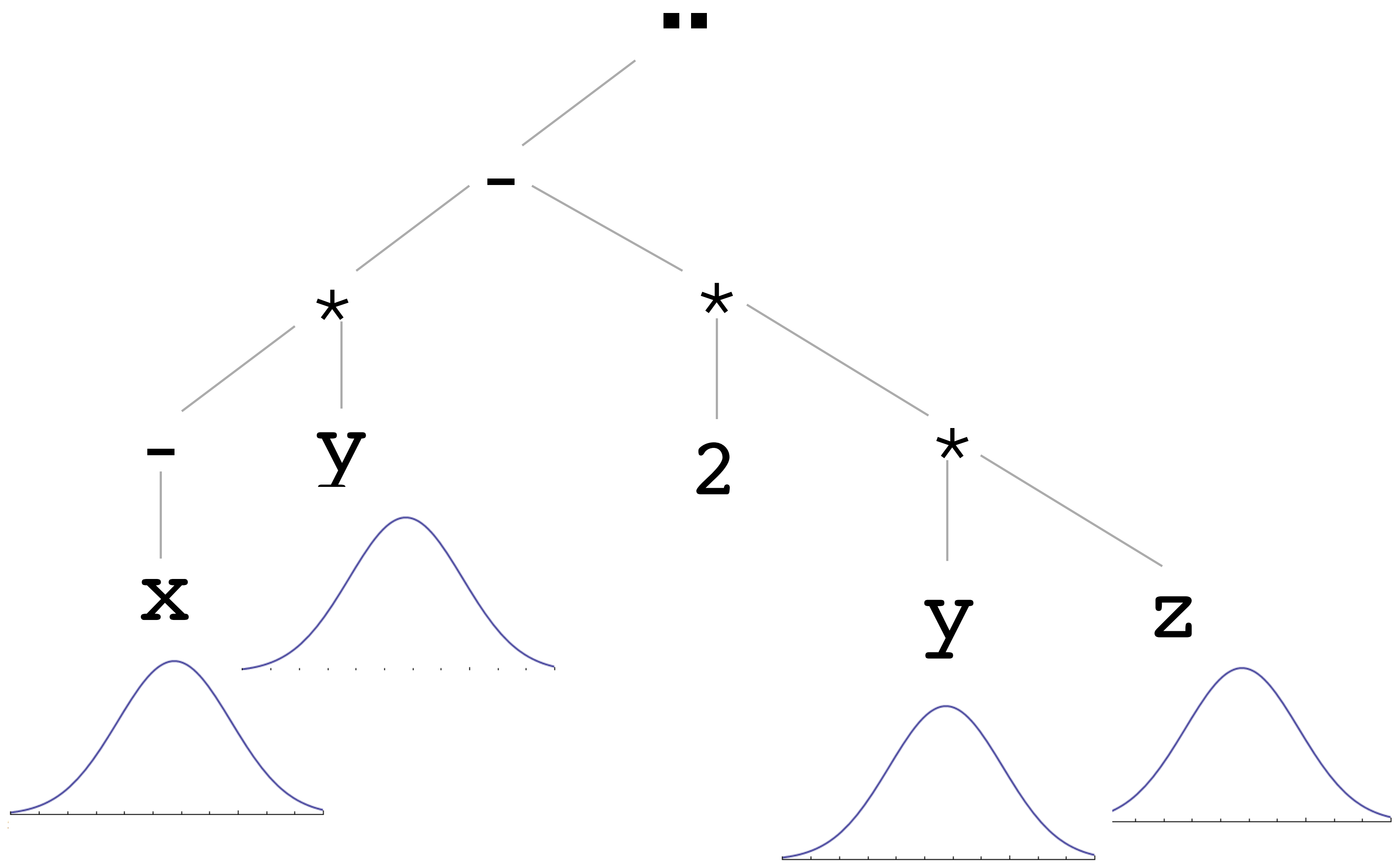
probabilistic interval subdivision

probabilistic error analysis

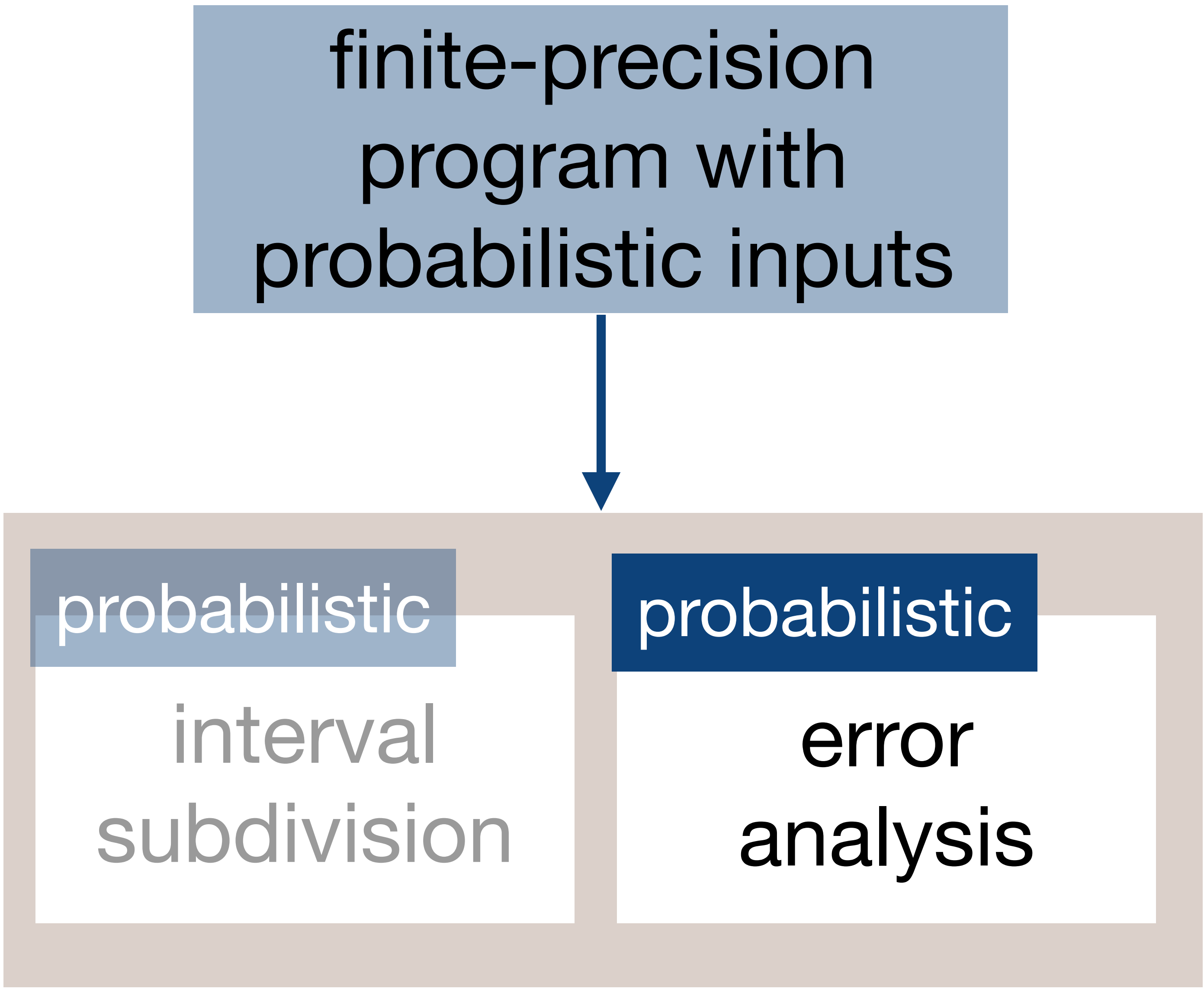


$< s_{ijk}, p_{ijk} >$

-x*y - 2*y*z - x - z



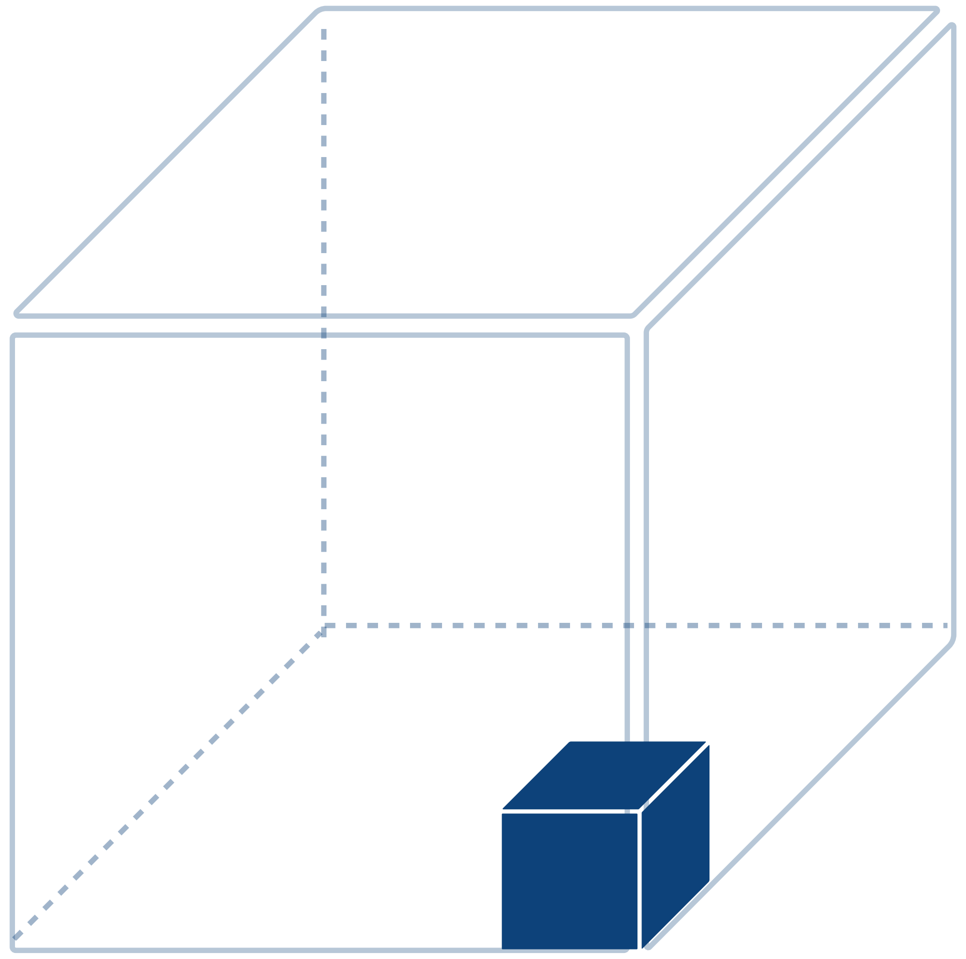
Probabilistic Error Analysis



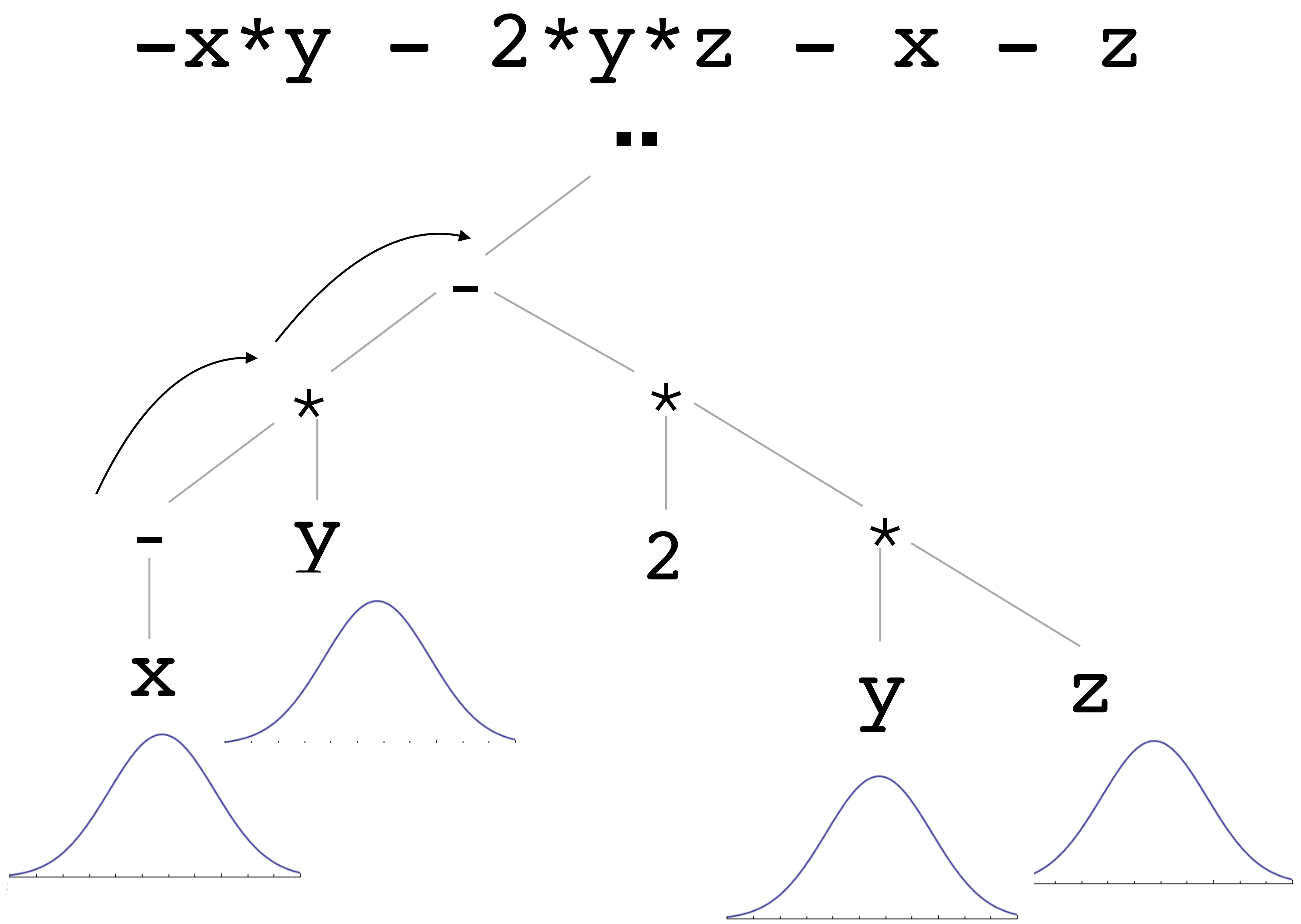
finite-precision program with probabilistic inputs

probabilistic interval subdivision

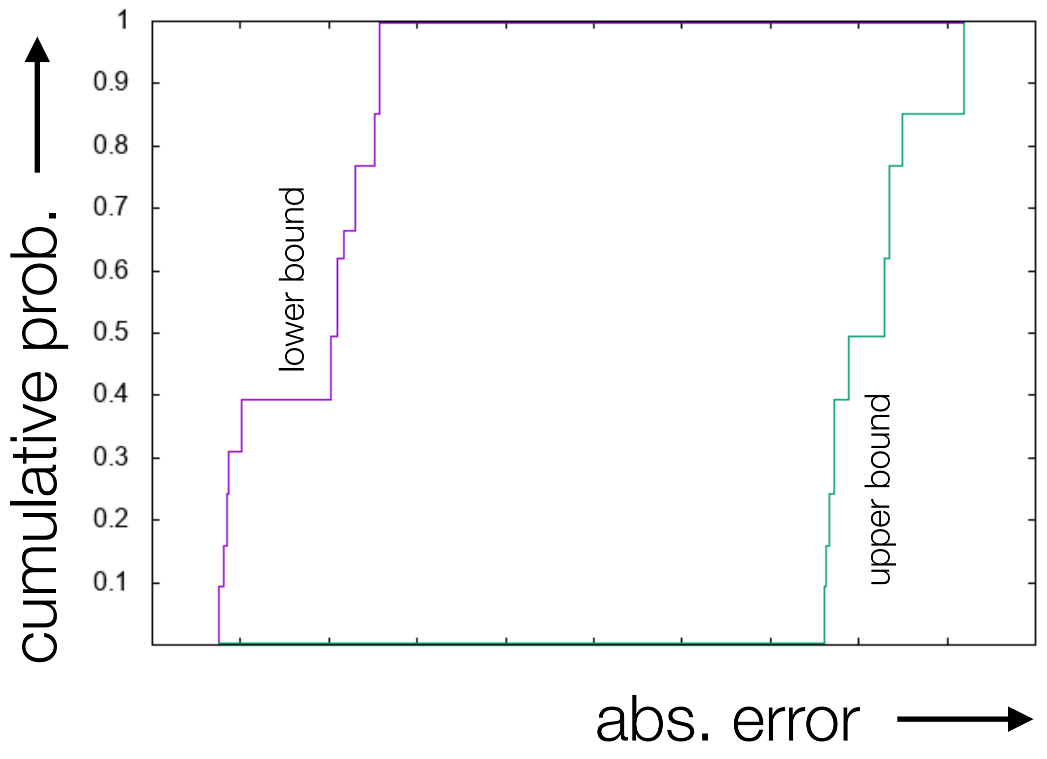
probabilistic error analysis



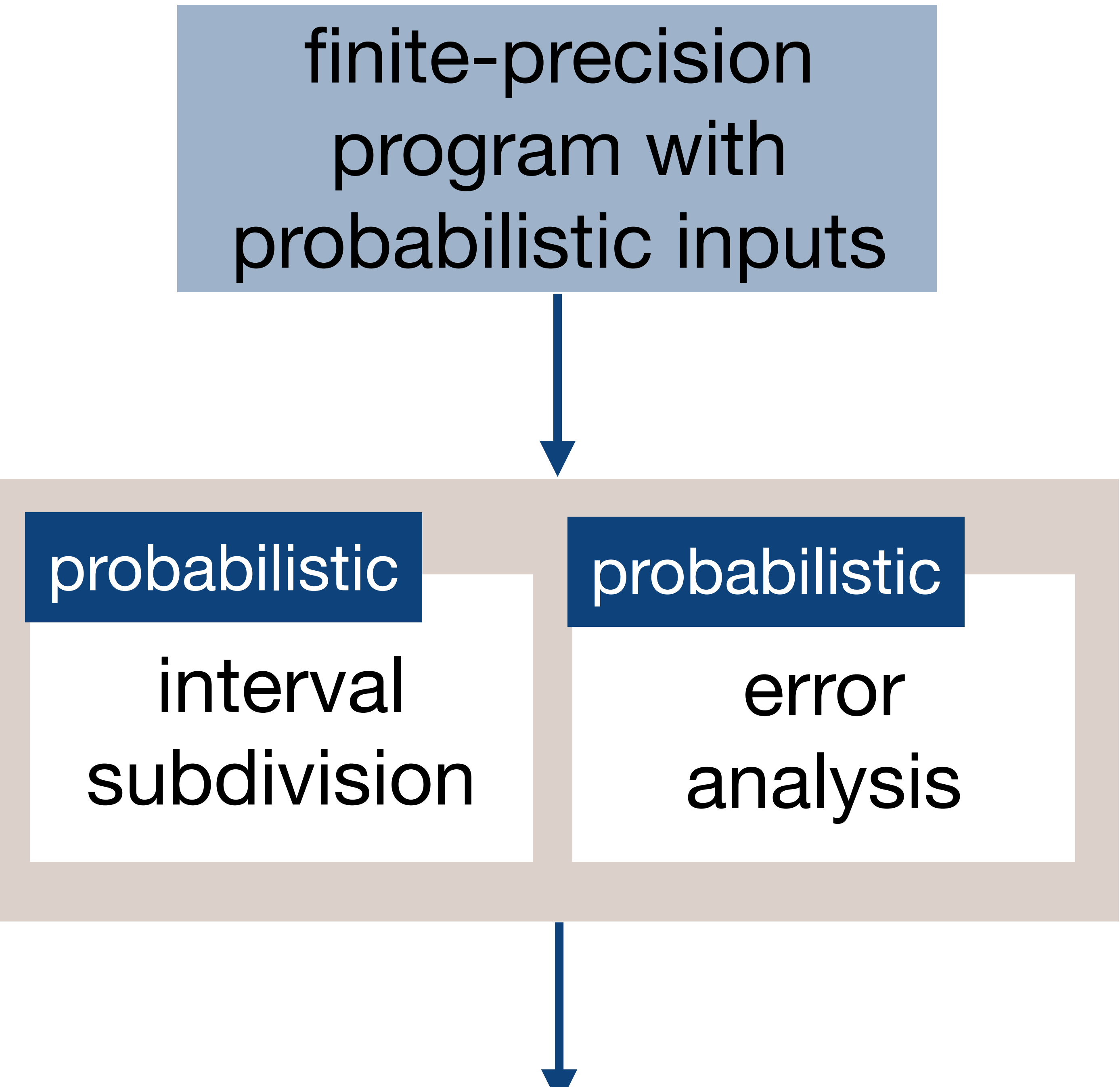
$< s_{ijk}, p_{ijk} >$



error distribution:



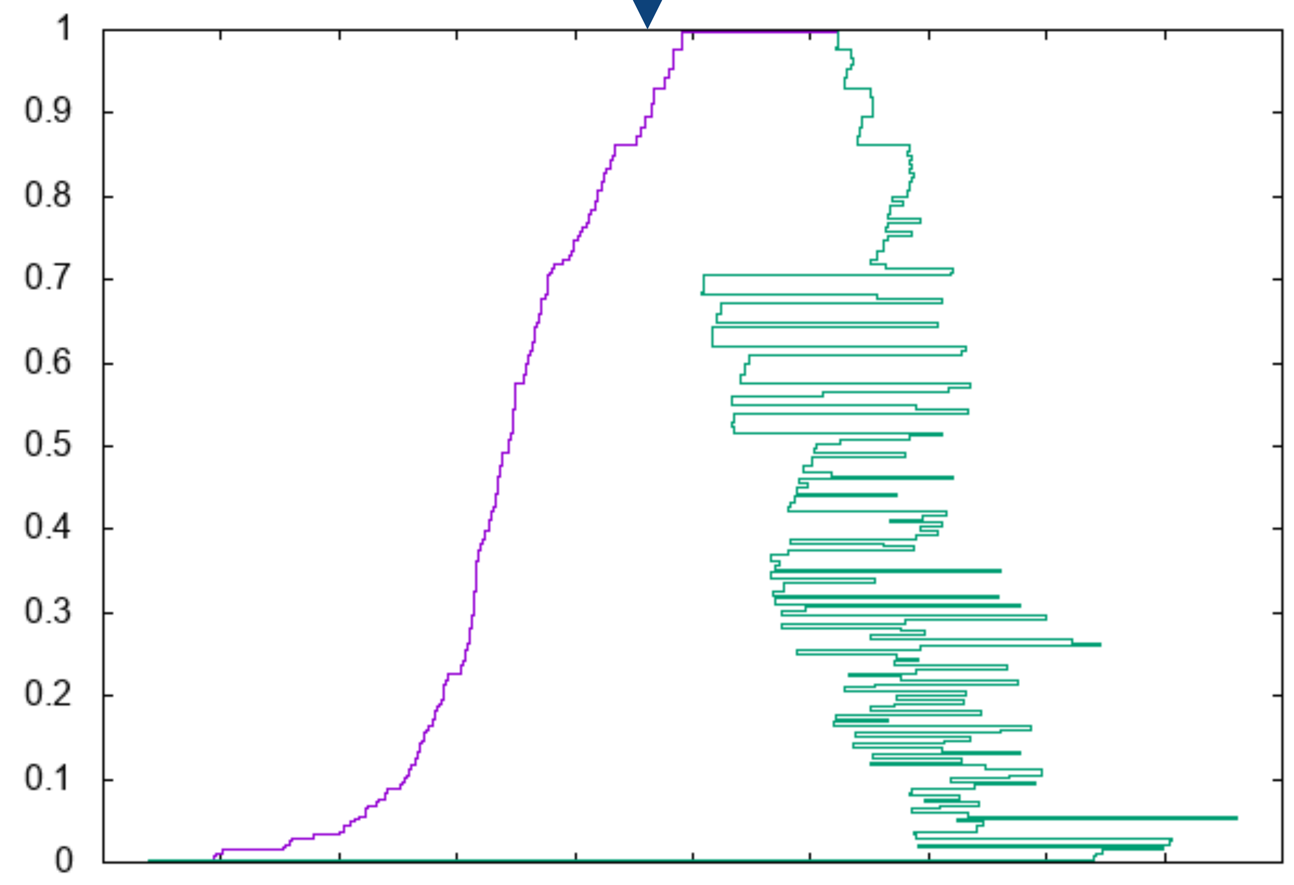
Probability Distribution of Errors

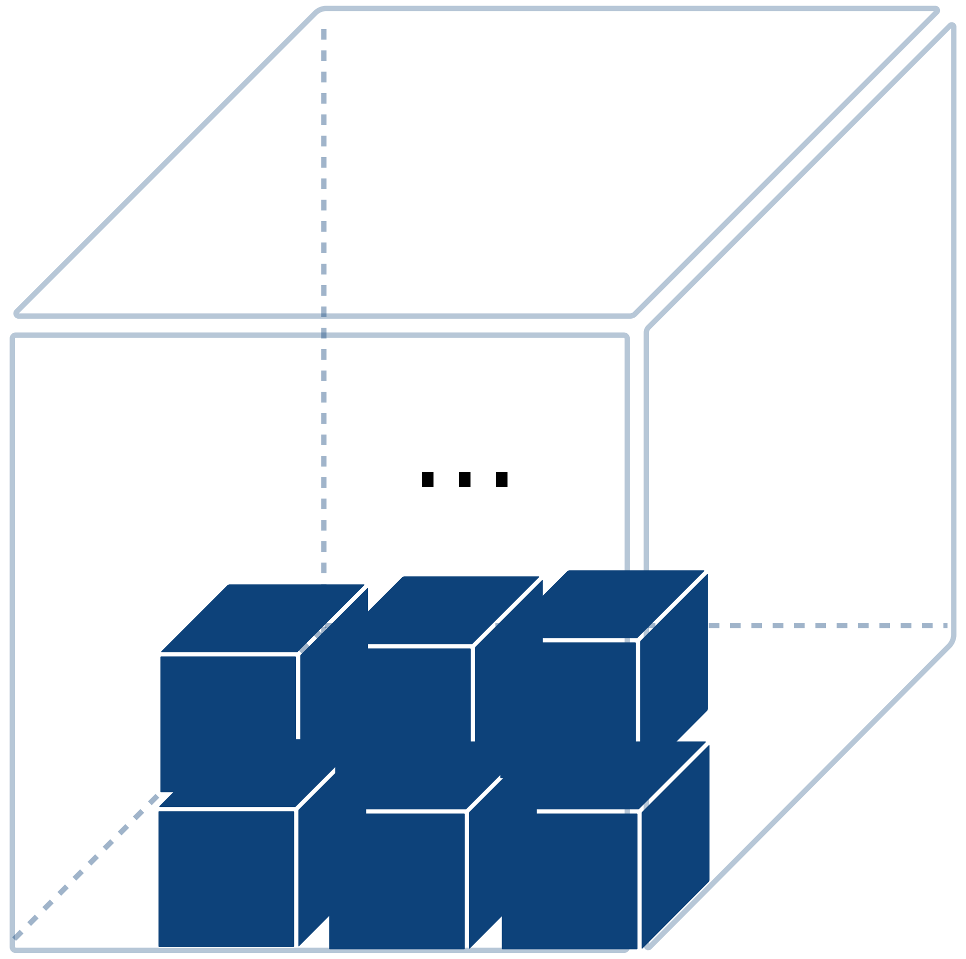


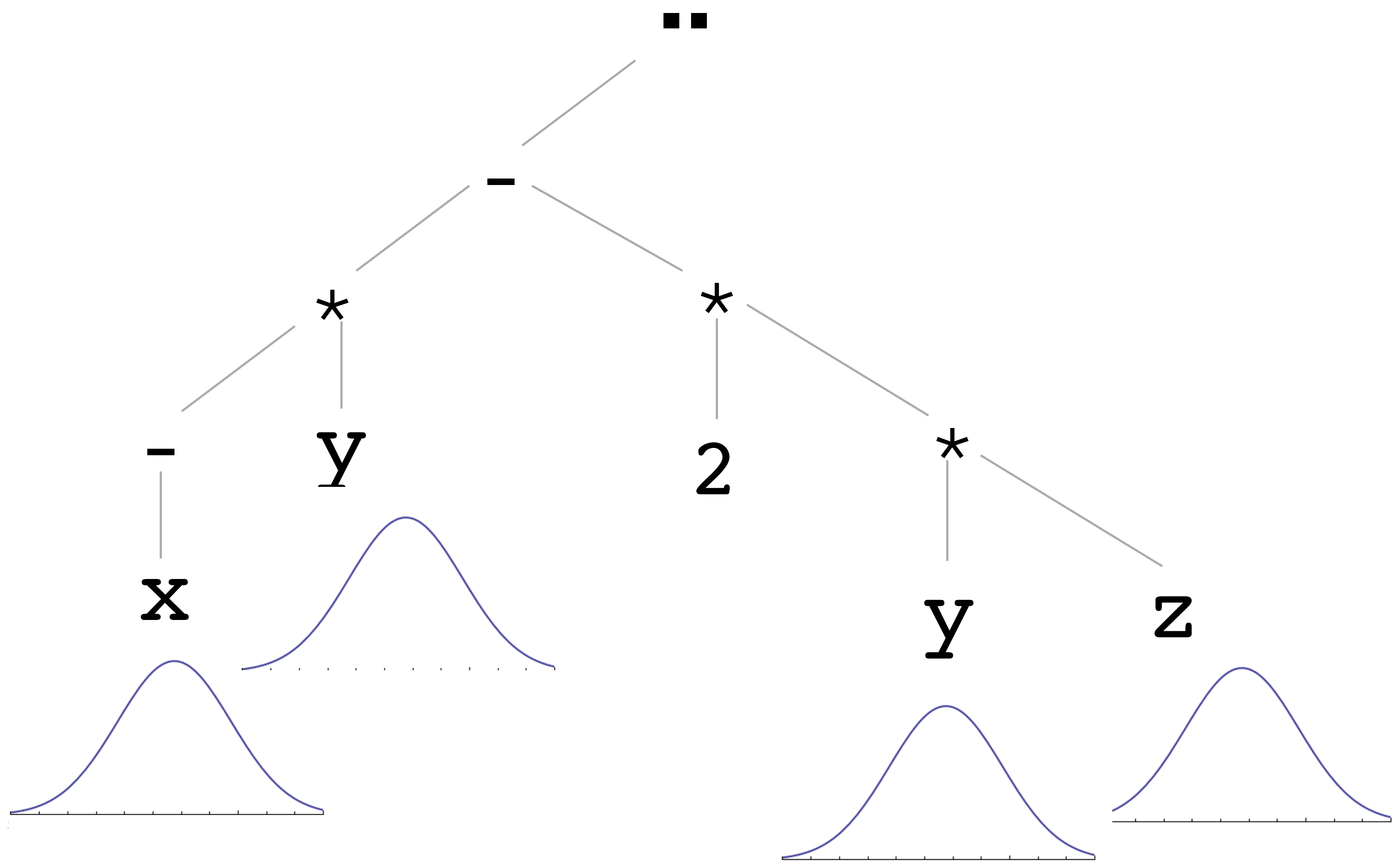
finite-precision program with probabilistic inputs

probabilistic interval subdivision

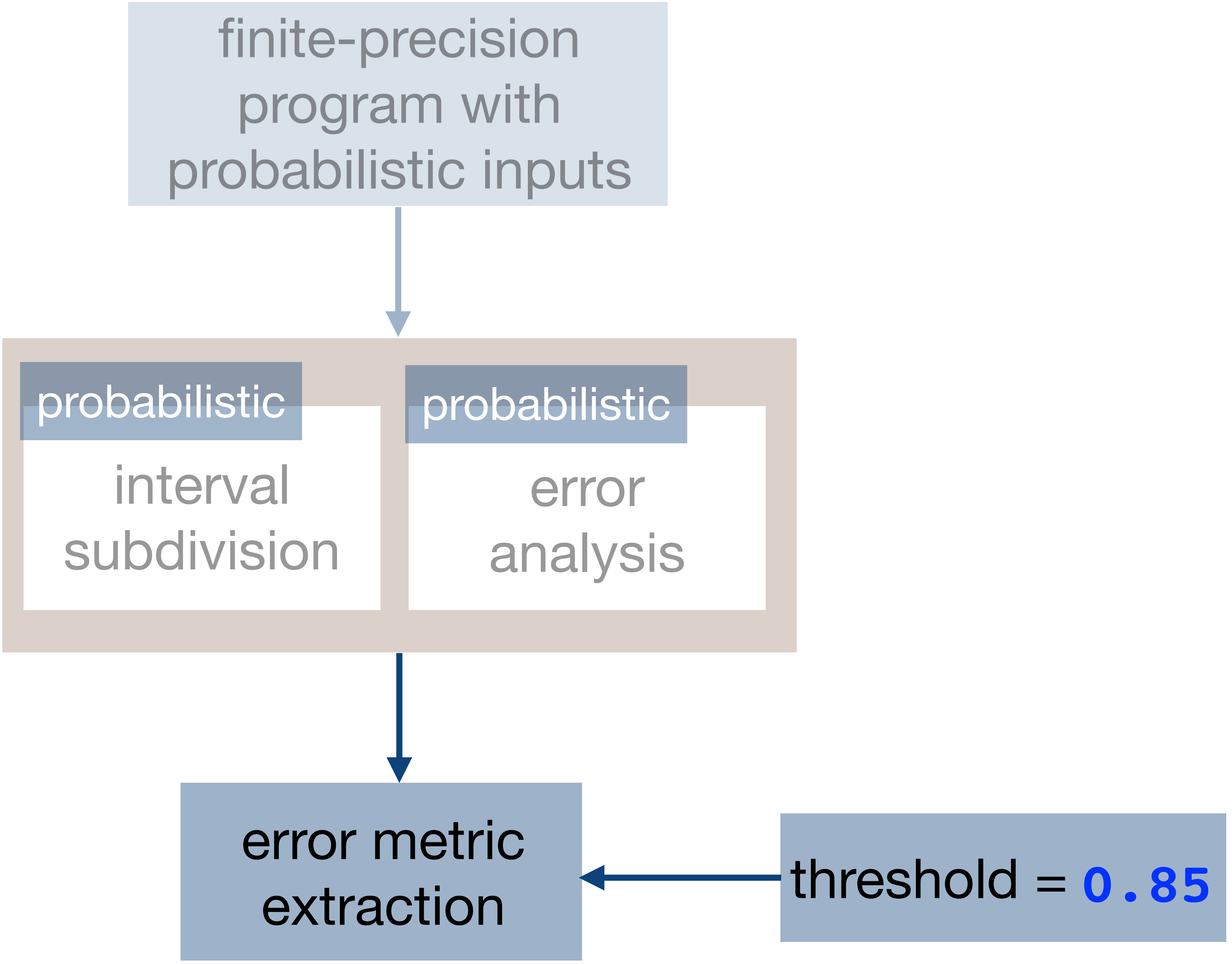
probabilistic error analysis







Refined Error Bounds



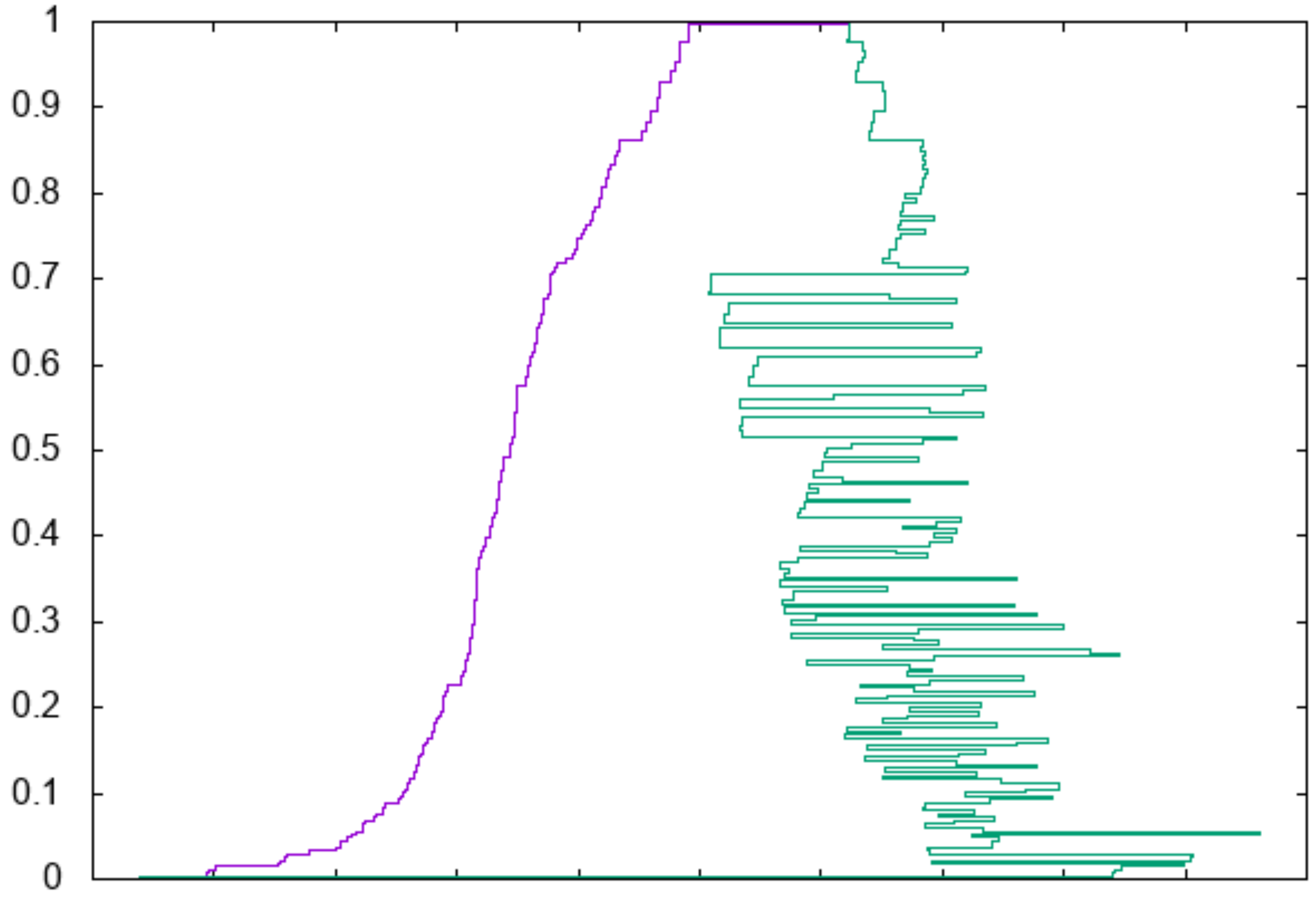
finite-precision program with probabilistic inputs

probabilistic interval subdivision

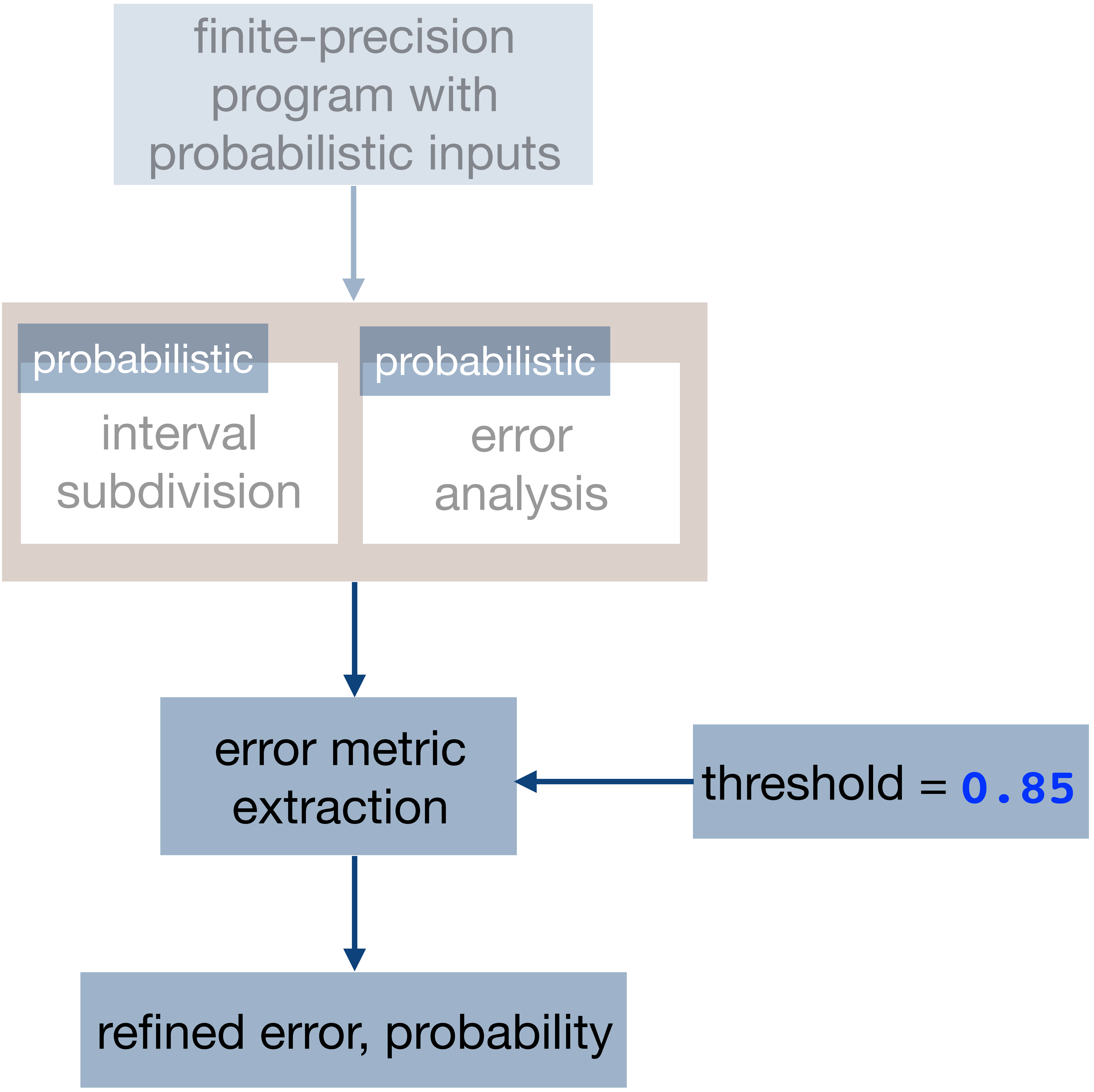
probabilistic error analysis

error metric extraction

threshold = `0.85`



Refined Error Bounds



finite-precision program with probabilistic inputs

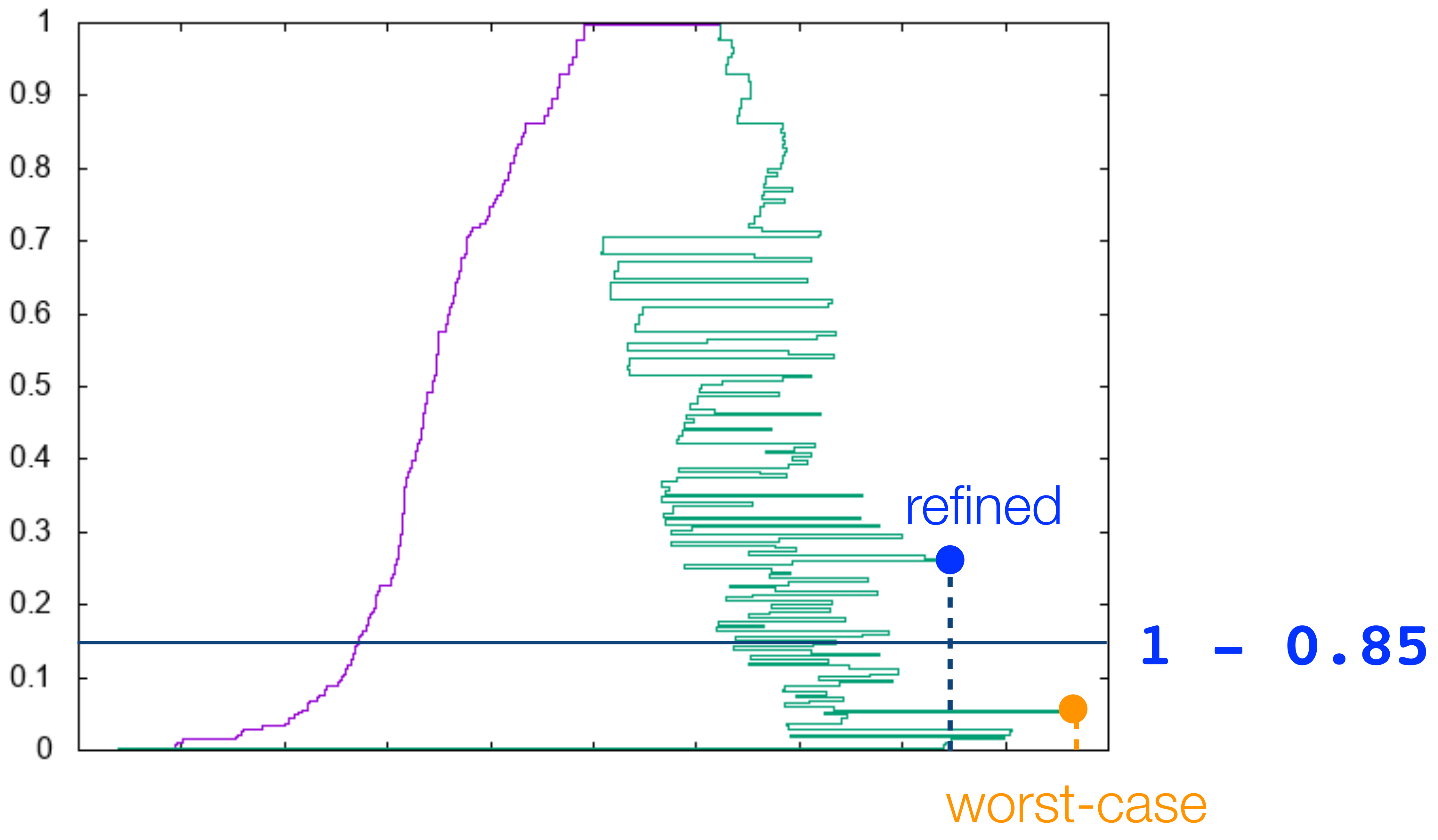
probabilistic interval subdivision

probabilistic error analysis

error metric extraction

threshold = **0.85**

refined error, probability



Summary of Results: Probabilistic Error Analysis

threshold probability: `0.85`, `32`-bit floating-point error

#benchmarks	#inputs	#arith-ops
25	1 - 9	4 - 25

Summary of Results: Probabilistic Error Analysis

threshold probability: `0.85`, `32`-bit floating-point error

			error reduction (%) from worst-case to the largest frequent			
#benchmarks	#inputs	#arith-ops	average		max	
			gaussian	uniform	gaussian	uniform
25	1 - 9	4 - 25	17.0	16.2	48.9	45.1

Summary of Results: Probabilistic Error Analysis

threshold probability: `0.85, 32`-bit floating-point error

			error reduction (%) from worst-case to the largest frequent			
#benchmarks	#inputs	#arith-ops	average		max	
			gaussian	uniform	gaussian	uniform
25	1 - 9	4 - 25	17.0	16.2	48.9	45.1

Reductions up to 73.1%
with approximate hardware specifications!

Takeaways

- Not all applications need worst-case guarantees
- Providing bounds on most frequent errors can be resource-efficient
- An automated probabilistic error analyzer: PrAn zenodo
  - strikes a balance between accuracy and complexity
  - handles different distributions, dependencies, and thresholds

Today's Talk: Probabilistic Error Analysis and NN Quantization

Accuracy Analysis

✓ iFM '19 EMSOFT '18

Probabilistic Analysis

TACAS '21

Static + Dynamic Analysis

~~worst-case error~~ analysis for small programs

Daisy FLUCTUAT Rosa

FPTaylor PRECiSA …

Optimization

EMSOFT '23

NN Quantization

worst-case tuning for ~~small (floating-point) programs~~

Daisy FPTuner

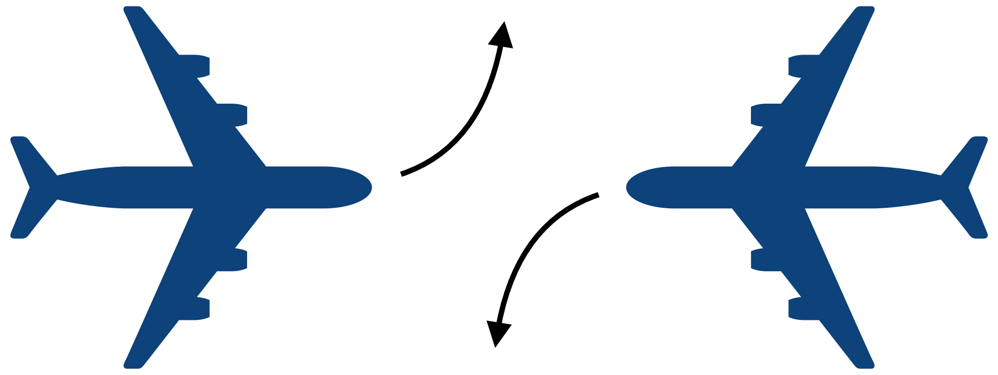
Sound Mixed Fixed-Point Quantization of Neural Networks

EMSOFT'23

How do we generate quantized implementations for neural networks that meet specified worst-case error bounds?

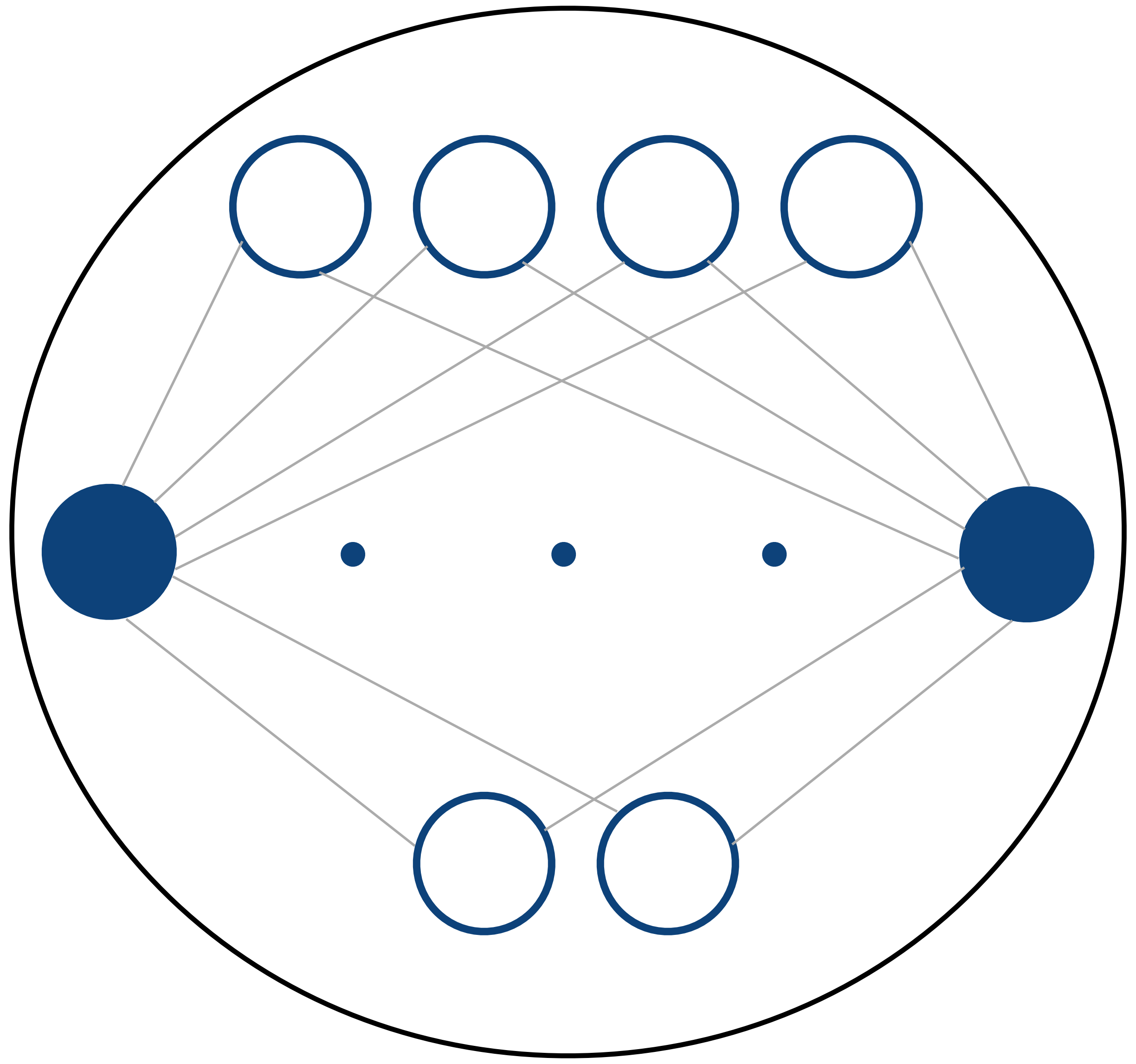
Neural networks are ubiquitous in safety-critical systems!

Adaptive Cruise Control



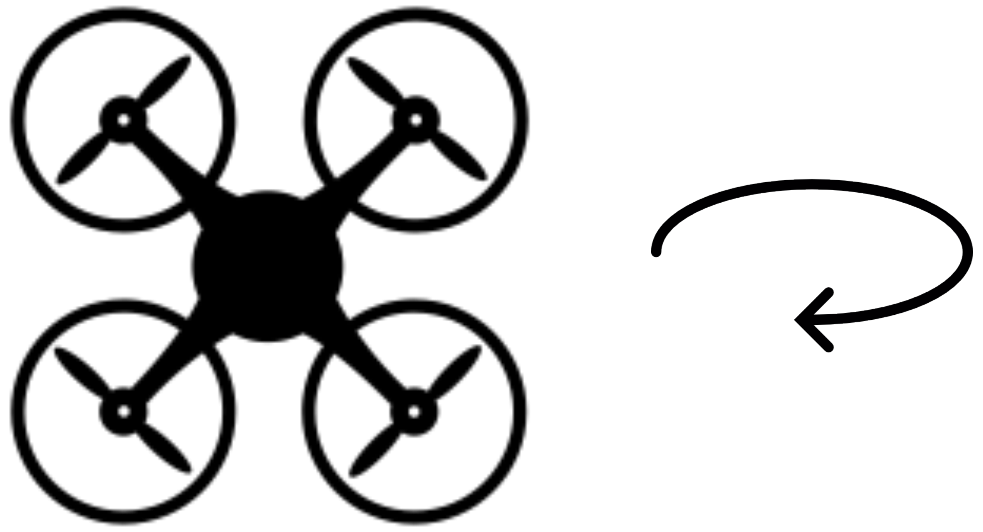
Collision Avoidance System

. . .



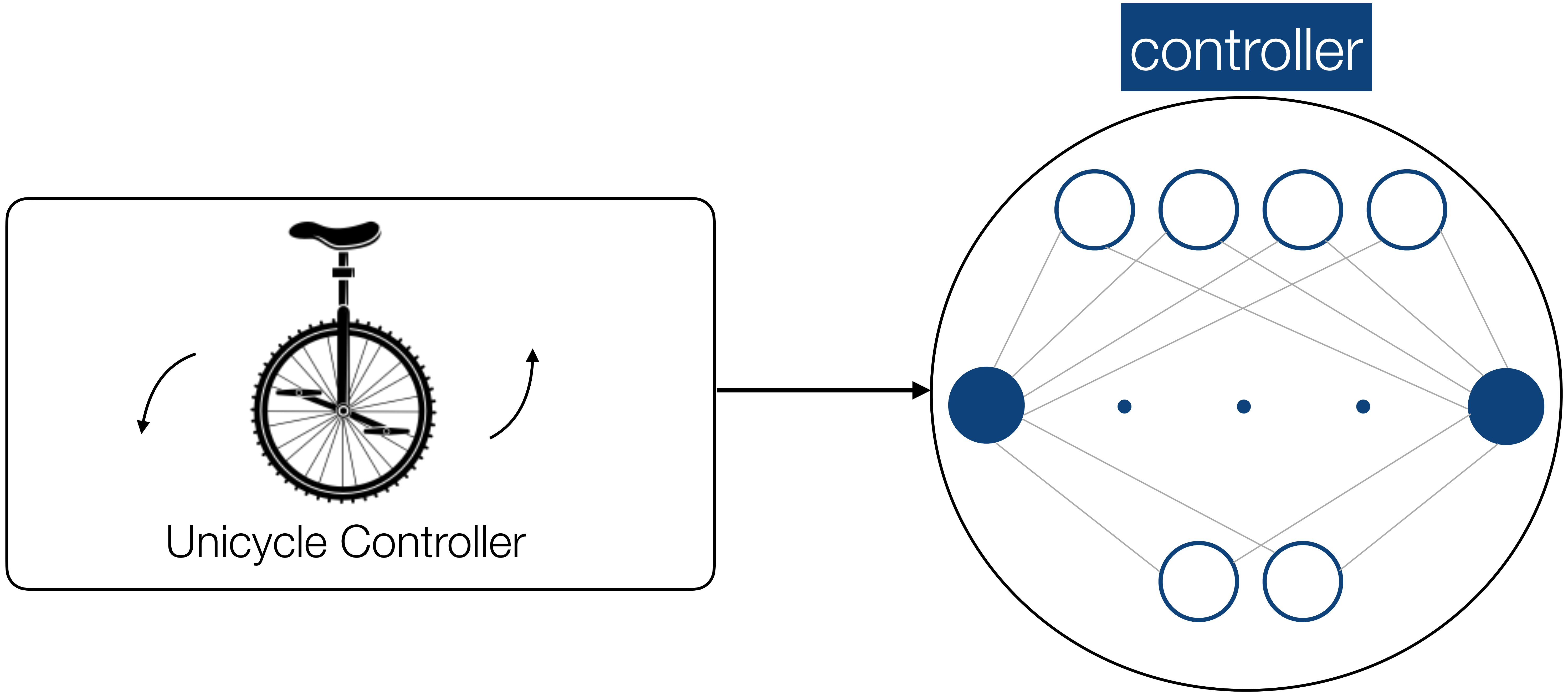
. . .

Unicycle Controller

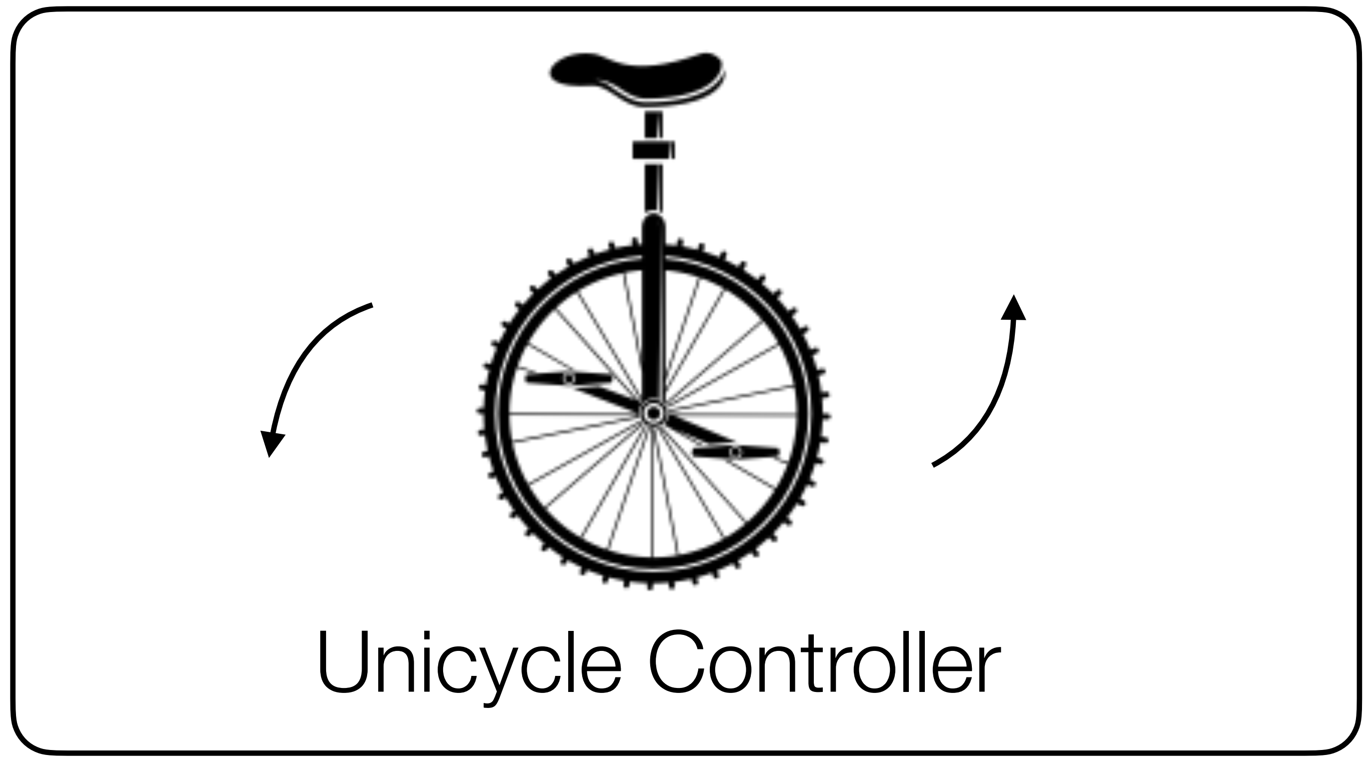


Drone Controller

Neural Networks as Controllers



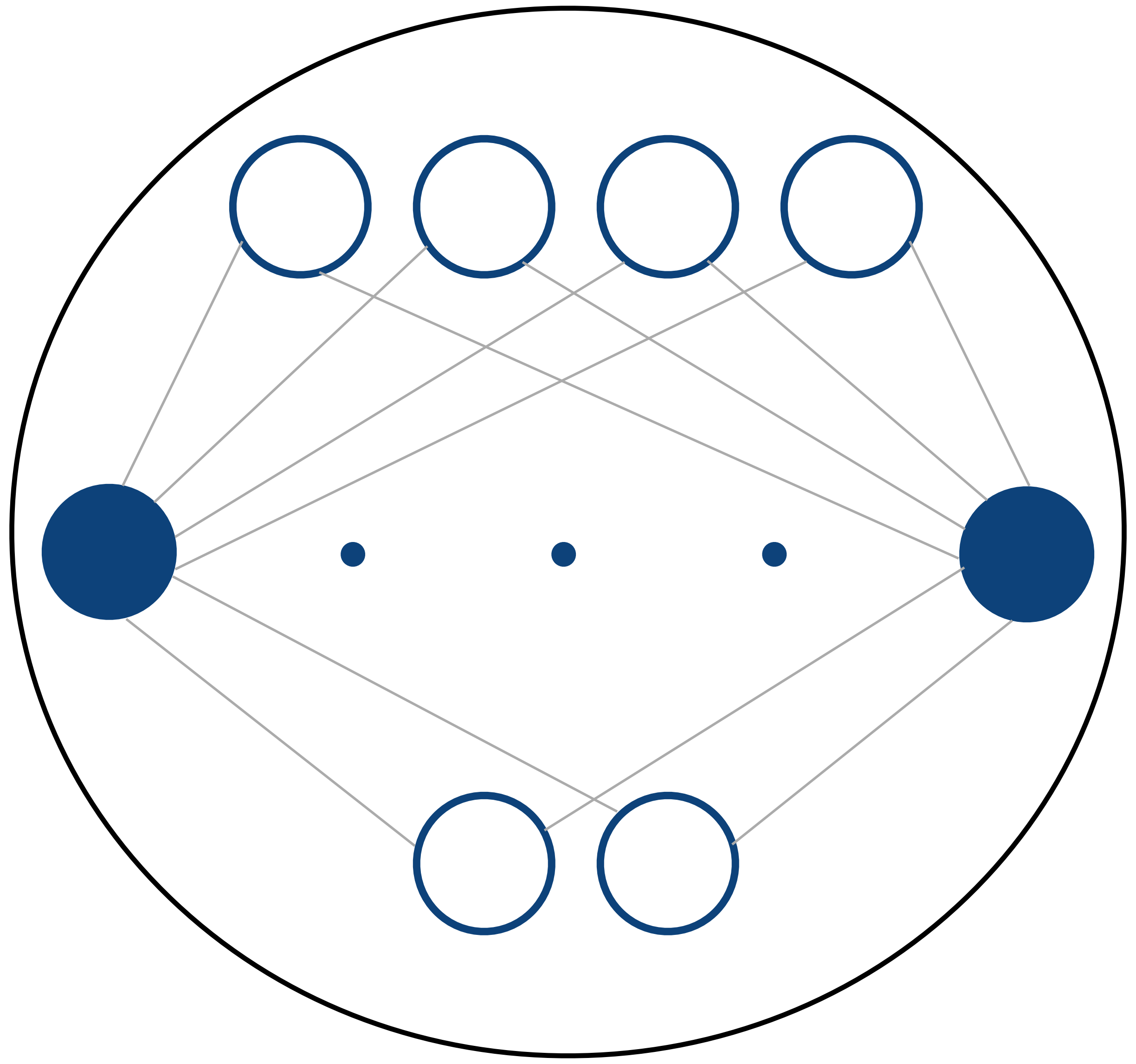
Neural Networks as Controllers



```
def UnicycleController(in: Vector): Vector = {
  weights1 = Matrix[...]
  weights2 = Matrix[...]
  bias1 = Vector(…)
  bias2 = Vector(...)
  x1 = relu(weights1 * in + bias1)
  out = linear(weights2 * x1 + bias2)
 return out
}
```

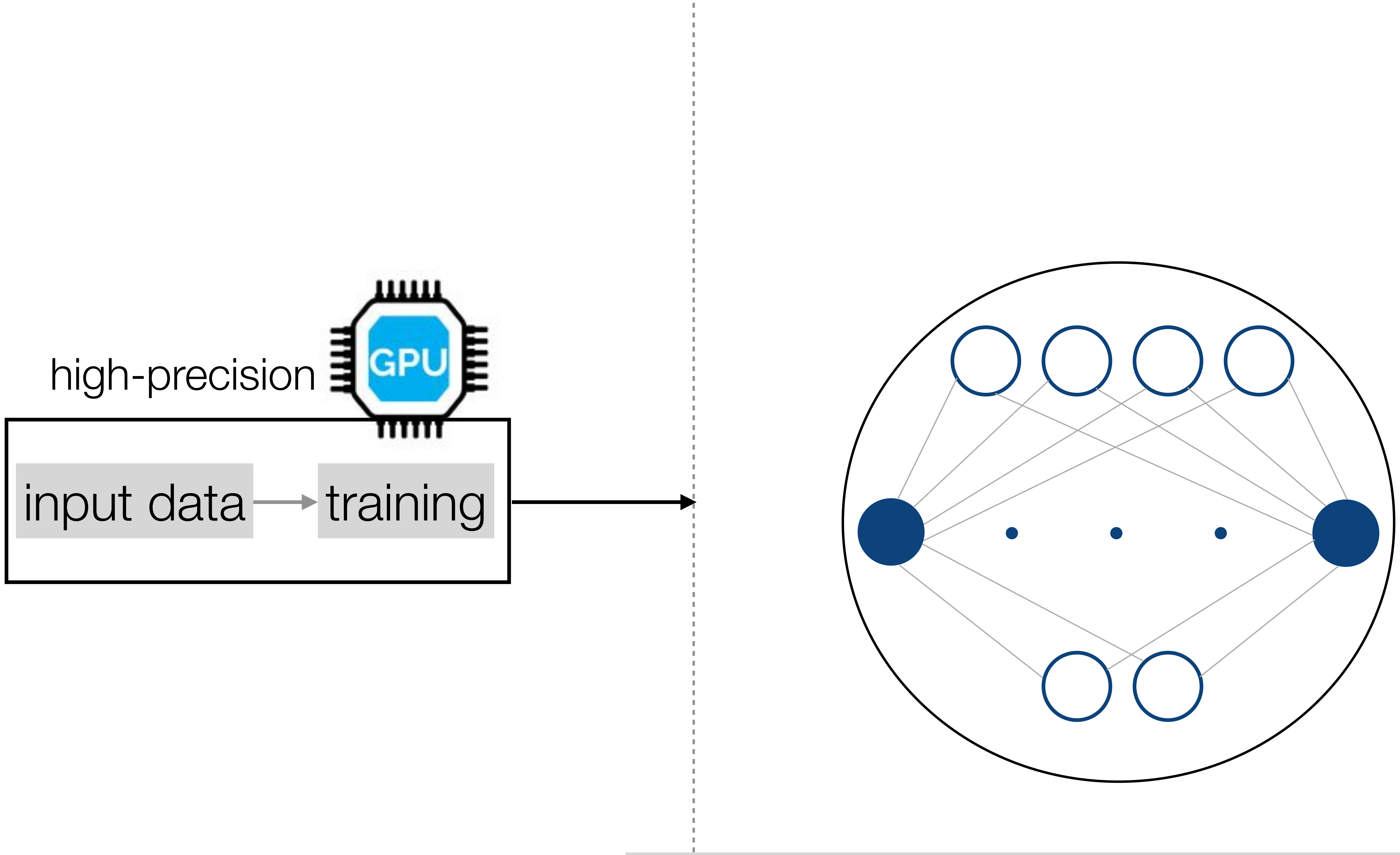
feed-forward regression models

Models are trained in High-Precision



```
x1 = relu([ W1 ] * [ in ] + [ b1 ])
out = linear([ W2 ] * [ x1 ] + [ b2 ])
```

Models are trained in High-Precision

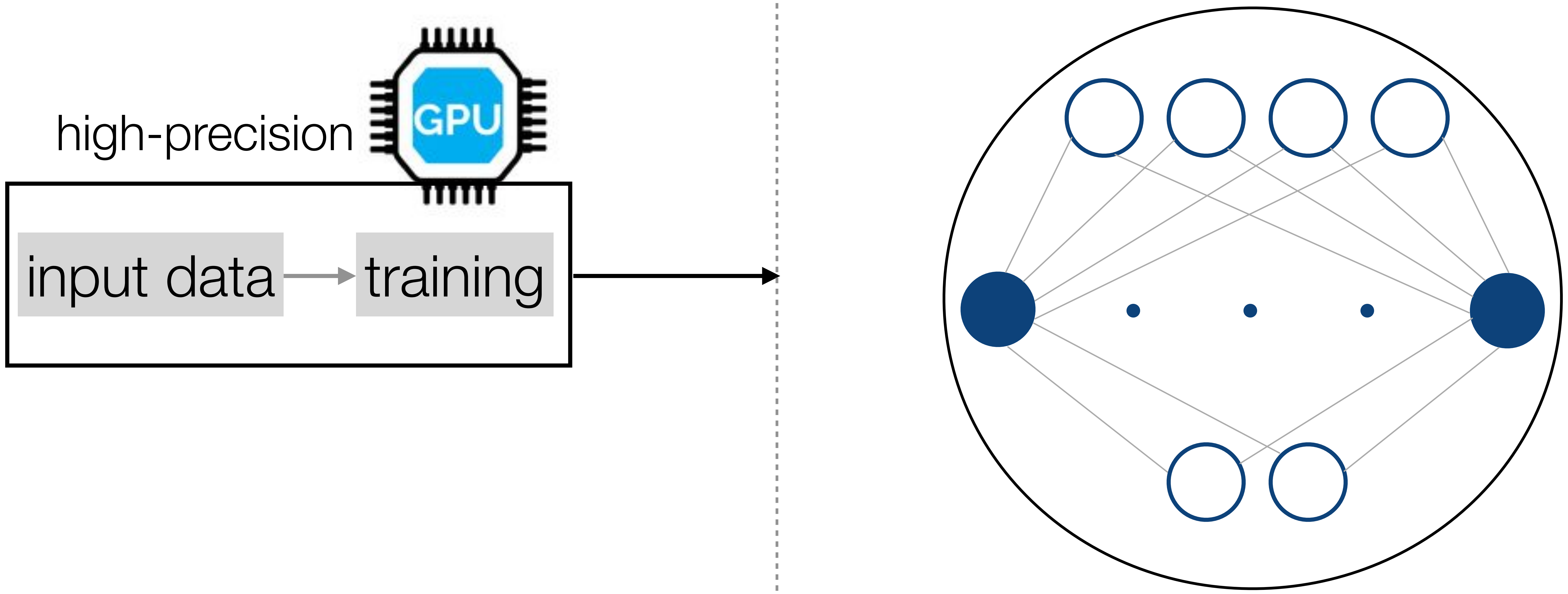


high-precision

input data → training

```
x1 = relu([ W1 ] * [ in ] + [ b1 ])
out = linear([ W2 ] * [ x1 ] + [ b2 ])
```

Models are trained in High-Precision

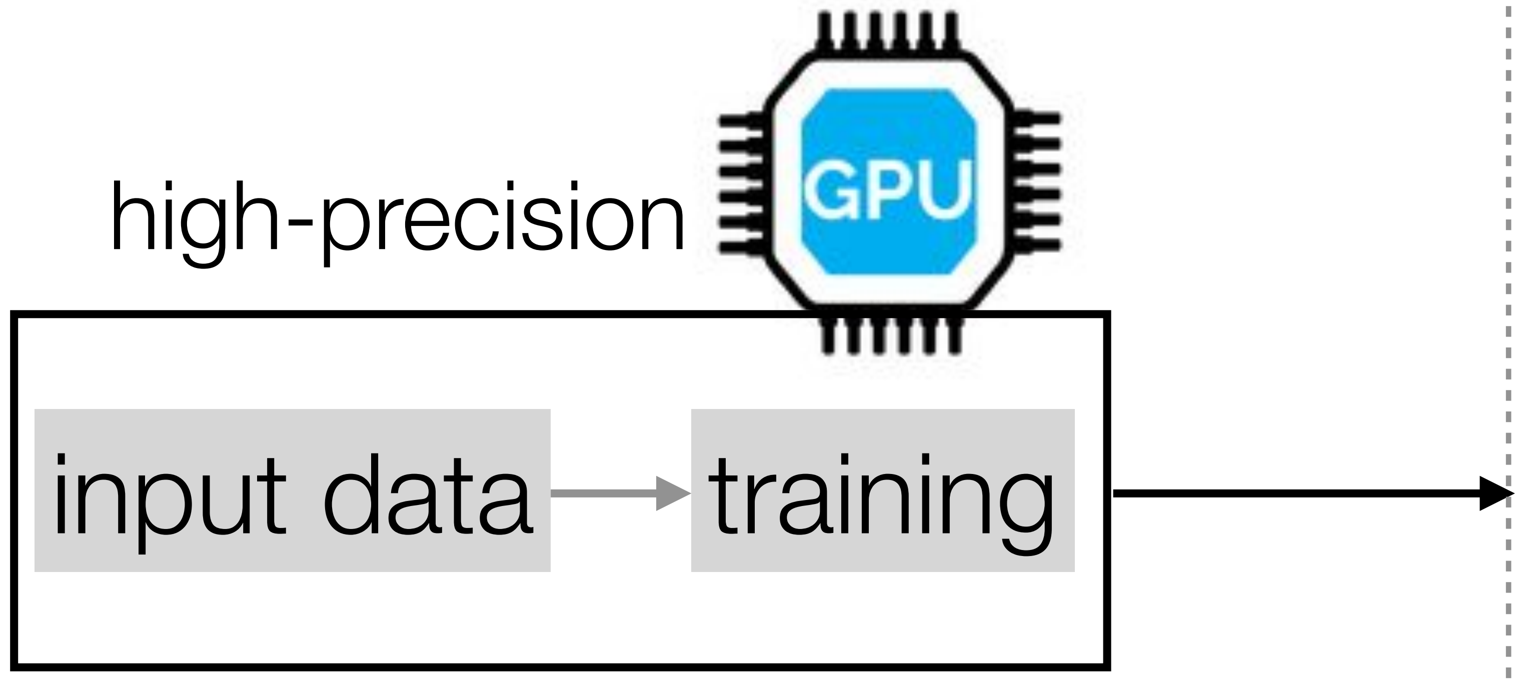


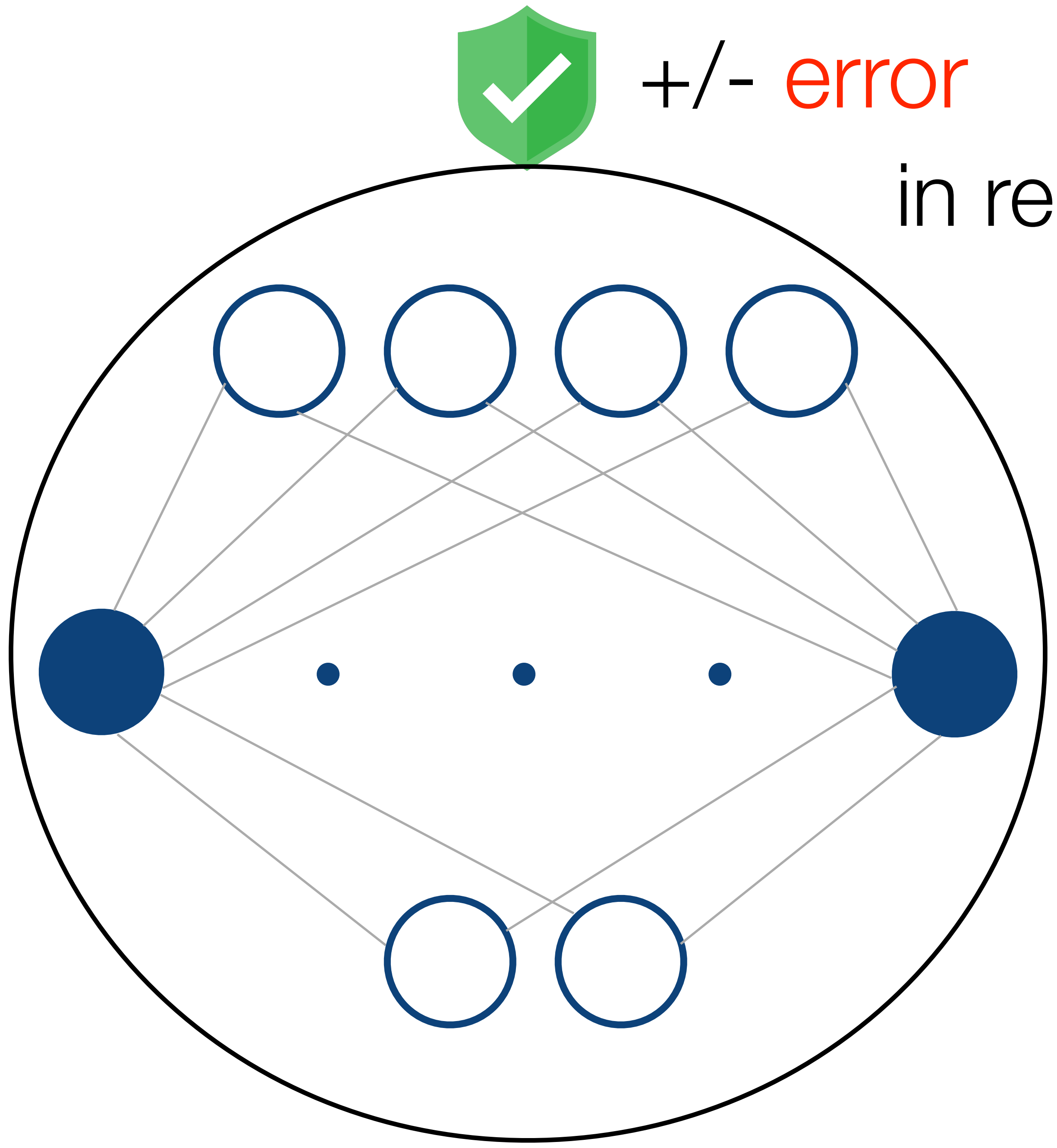
```
x1 = relu([ W1 ]
out = linear([ W2
```

```
⎡-5.23724322e-03 ... 1.30853499e-04 ⎤
                 ...
⎣-7.29779880e-01 ... -2.27958648e-04⎦
```

64-bit floating-point

Models are trained in High-Precision





+/- error

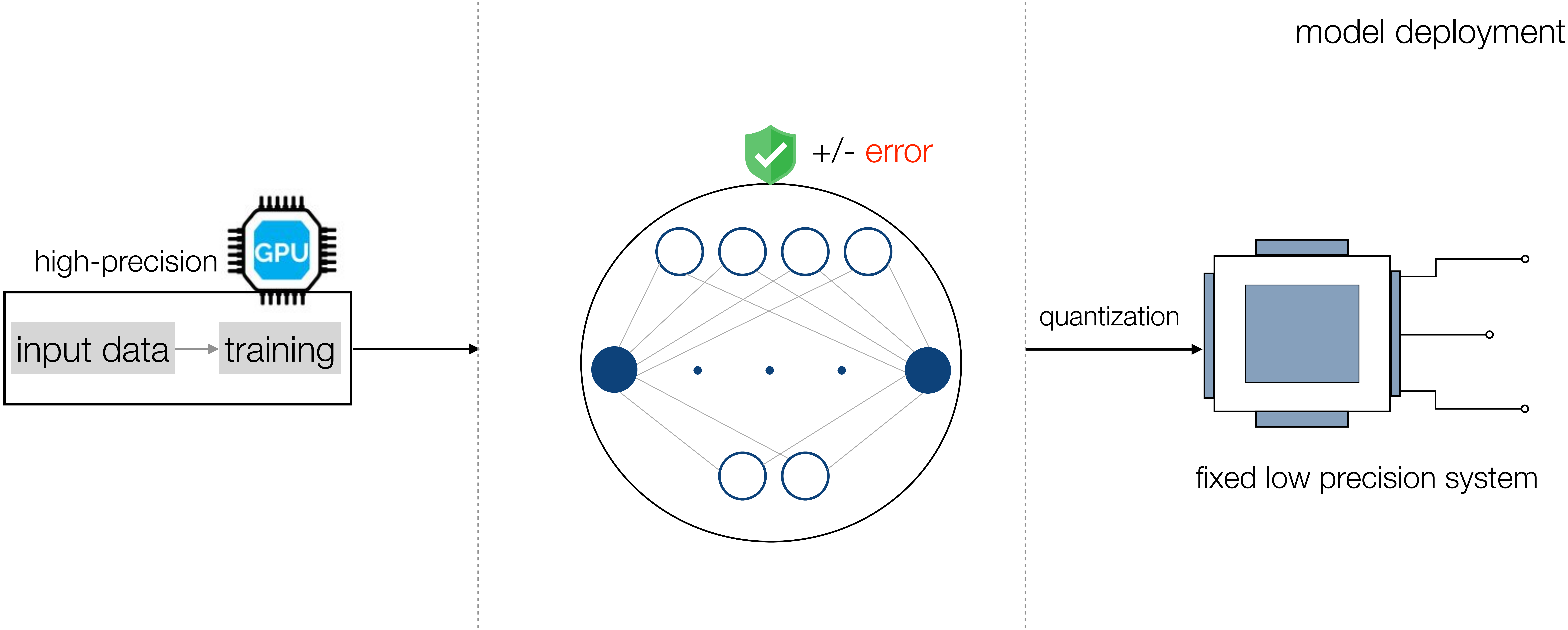
in real-valued arithmetic

```
x1 = relu([ W1 ]
out = linear([ W2
```

```
┌-5.23724322e-03 ... 1.30853499e-04 ┐
                ...
└-7.29779880e-01 ... -2.27958648e-04┘
```

64-bit floating-point

Model Deployment requires Quantization



model deployment

high-precision

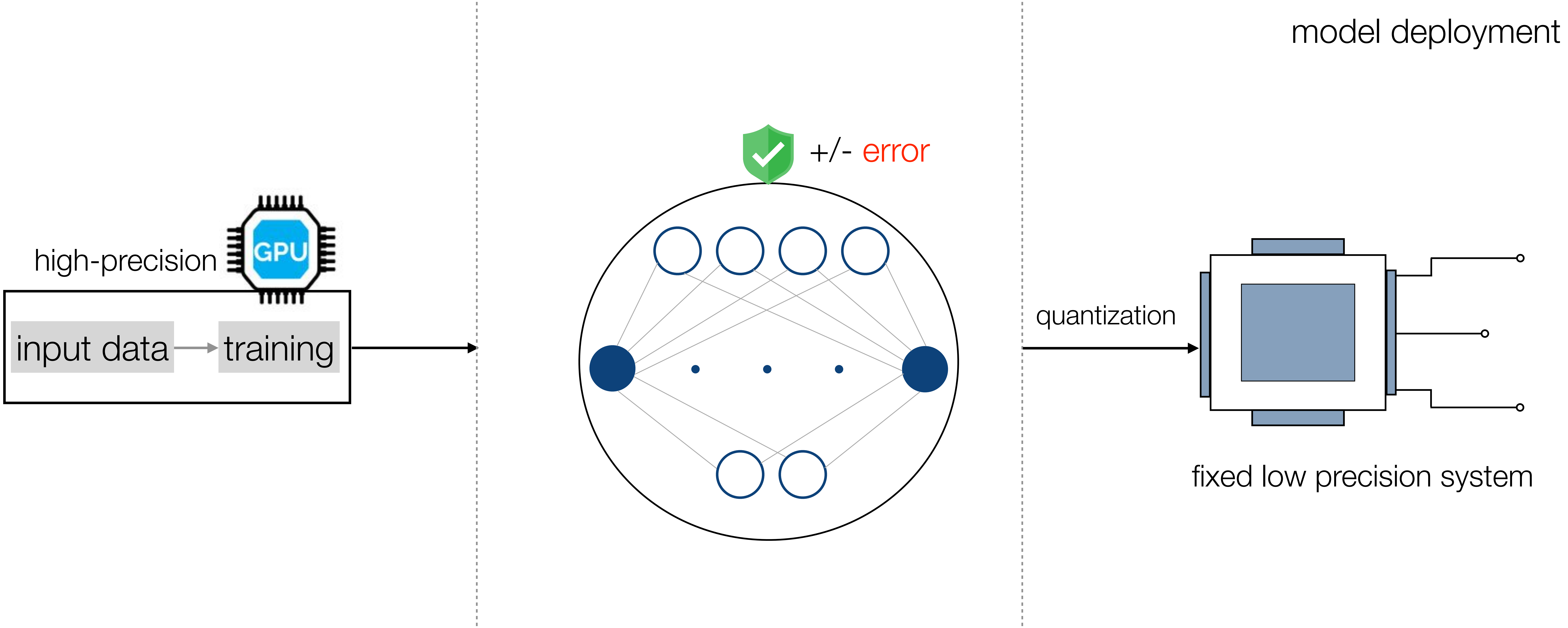
input data → training

+/- error

quantization

fixed low precision system

Model Deployment requires Quantization

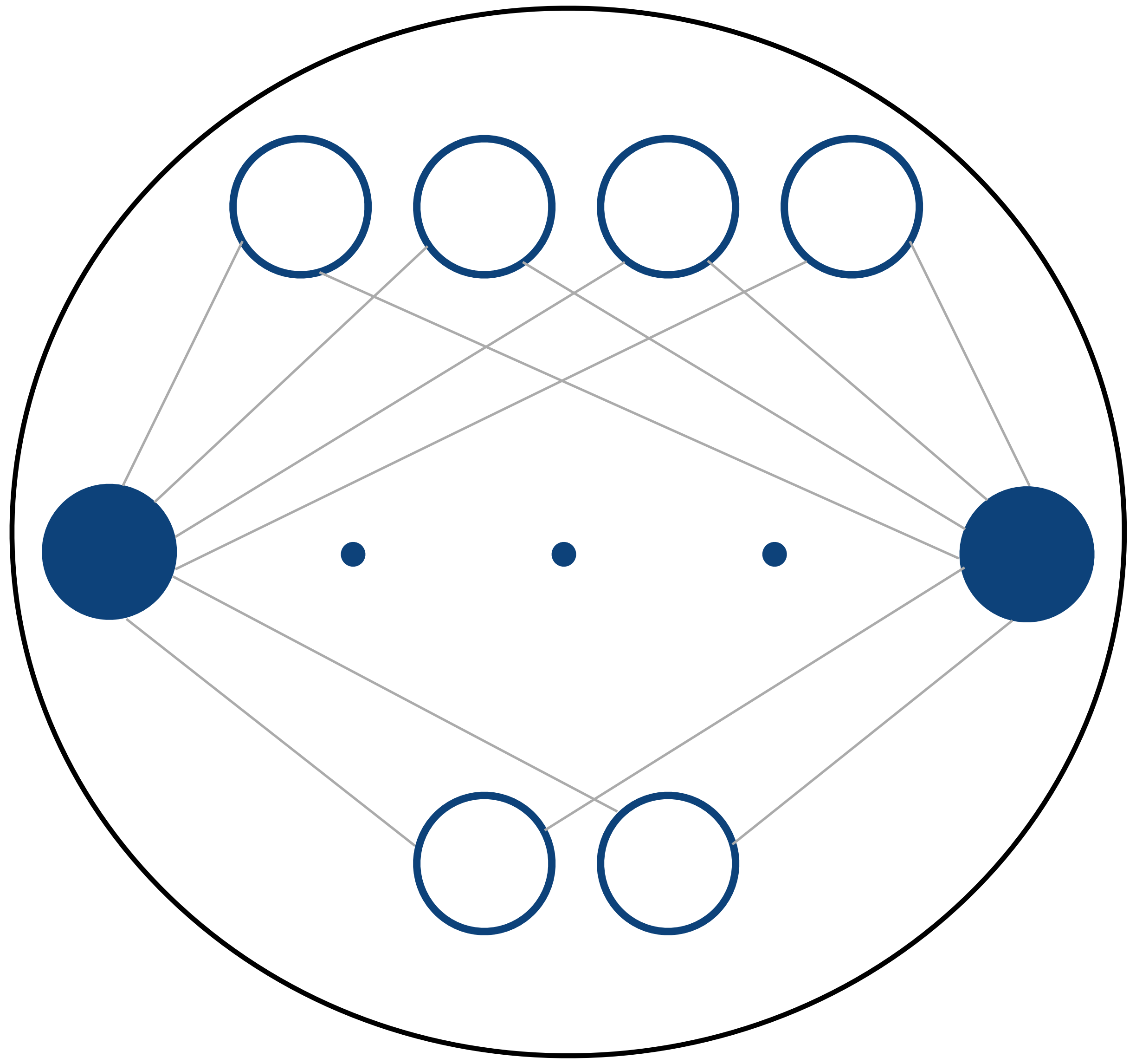


We need to quantize respecting the error bound!

Sound Mixed Fixed-Point Quantization

Unicycle Controller

```
-0.6 <= in1 <= 9.55
-4.5 <= in2 <= 0.2
-0.06 <= in3 <= 2.11
-0.3 <= in4 <= 1.51
```



```
res +/- 1e-3
```

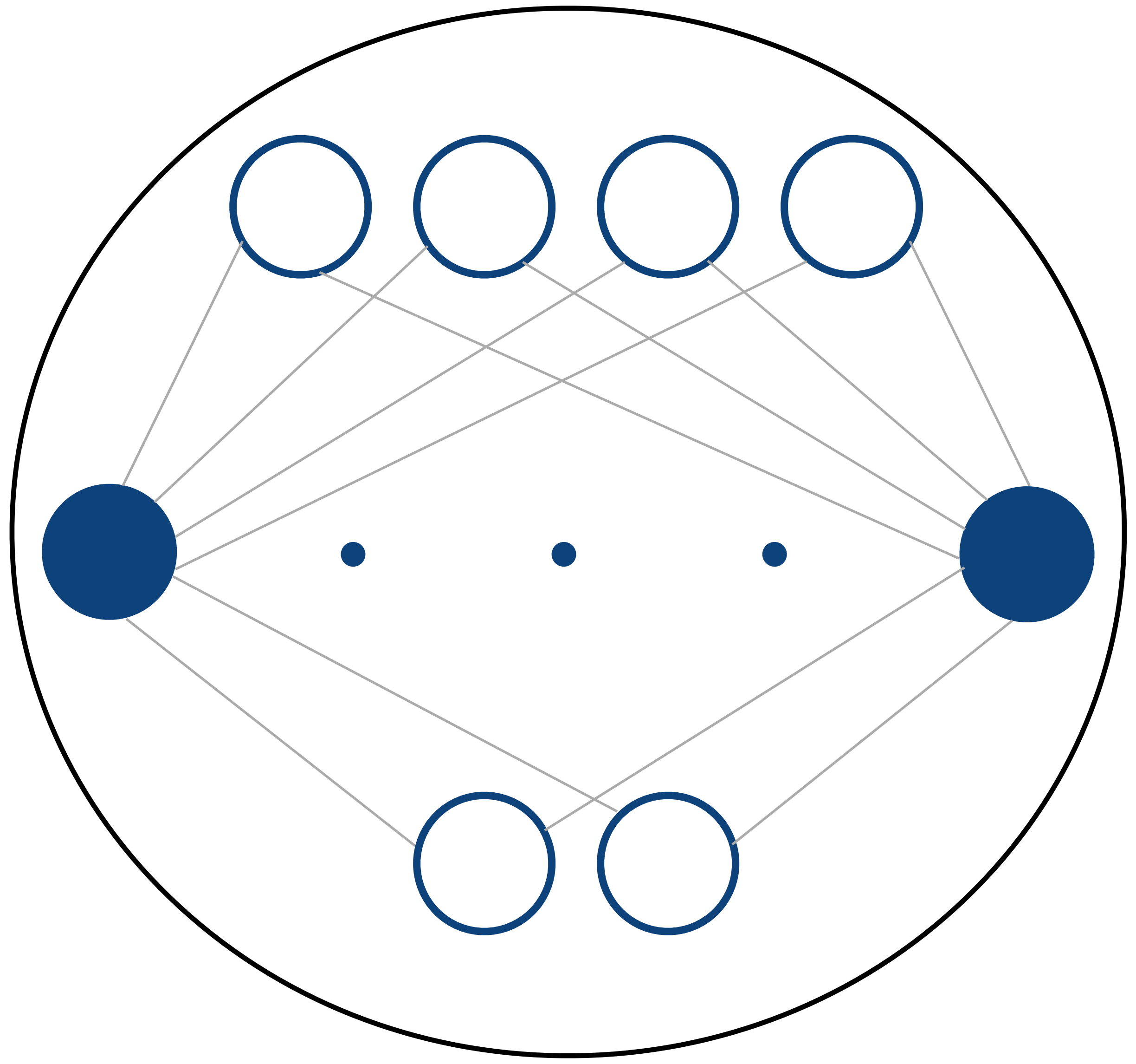
mixed precision fixed-point code

directly synthesized

XILINX

State-of-the-art is not enough!

```
-0.6 <= in1 <= 9.55
-4.5 <= in2 <= 0.2
-0.06 <= in3 <= 2.11
-0.3 <= in4 <= 1.51
```



```
res +/- 1e-3
```

no fixed-point support!

✗ FPTuner

! Daisy

- not scalable
- needs unrolled structures
- over-approximates a lot

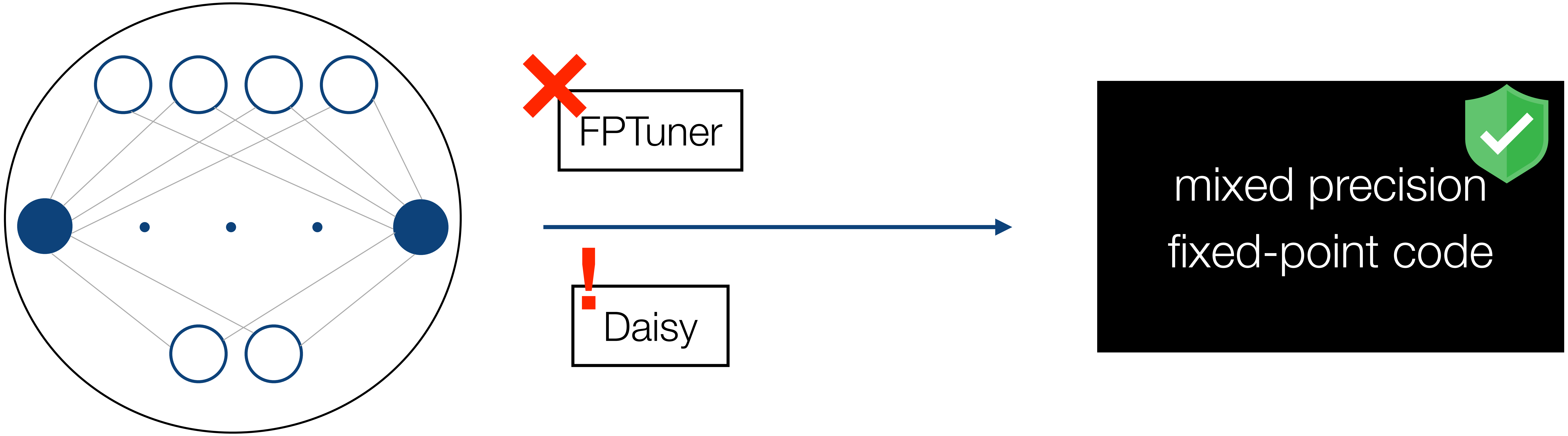
mixed precision fixed-point code ✓

directly synthesized

XILINX

State-of-the-art is not enough!

```
-0.6 <= in1 <= 9.55
-4.5 <= in2 <= 0.2
-0.06 <= in3 <= 2.11
-0.3 <= in4 <= 1.51
```



FPTuner

Daisy

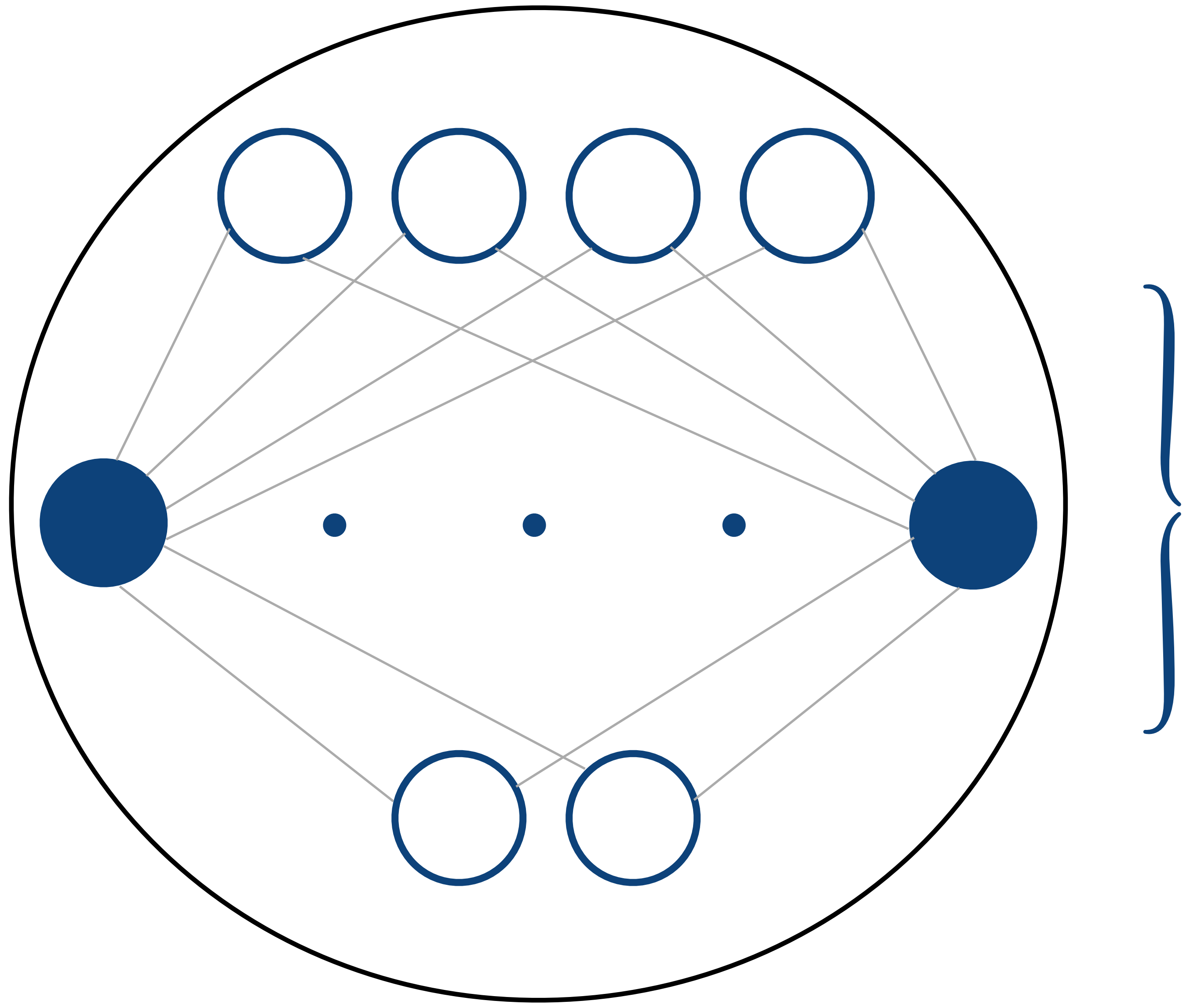
mixed precision fixed-point code

```
res +/- 1e-3
```

Our Contribution: Sound Scalable Quantizer for NNs

Key Idea: Quantization for efficiency is an optimization problem!

```
-0.6 <= in1 <= 9.55
-4.5 <= in2 <= 0.2
-0.06 <= in3 <= 2.11
-0.3 <= in4 <= 1.51
```



```
minimize: precision
```

cost function

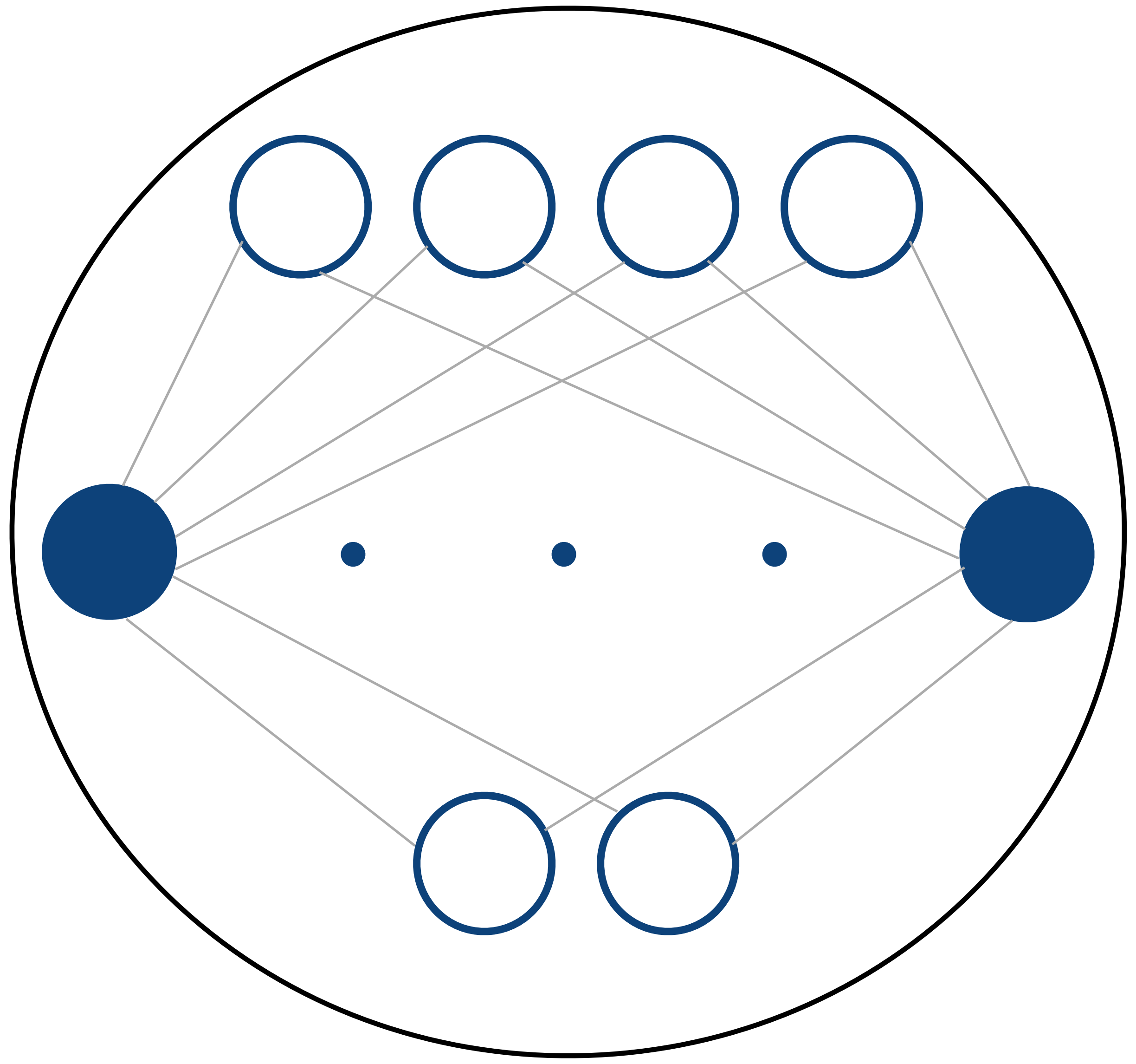
```
res +/- 1e-3
```

$$\text{minimize:} \quad \gamma = \sum_{i=1}^{n} \gamma_i^{dot} + \gamma_i^{bias} + \gamma_i^{\alpha}$$

- integer-valued cost

Key Idea: Quantization for efficiency is an optimization problem!

```
-0.6 <= in1 <= 9.55
-4.5 <= in2 <= 0.2
-0.06 <= in3 <= 2.11
-0.3 <= in4 <= 1.51
```



```
res +/- 1e-3
```

error constraint

minimize: $\gamma = \sum_{i=1}^{n} \gamma_i^{dot} + \gamma_i^{bias} + \gamma_i^{\alpha}$

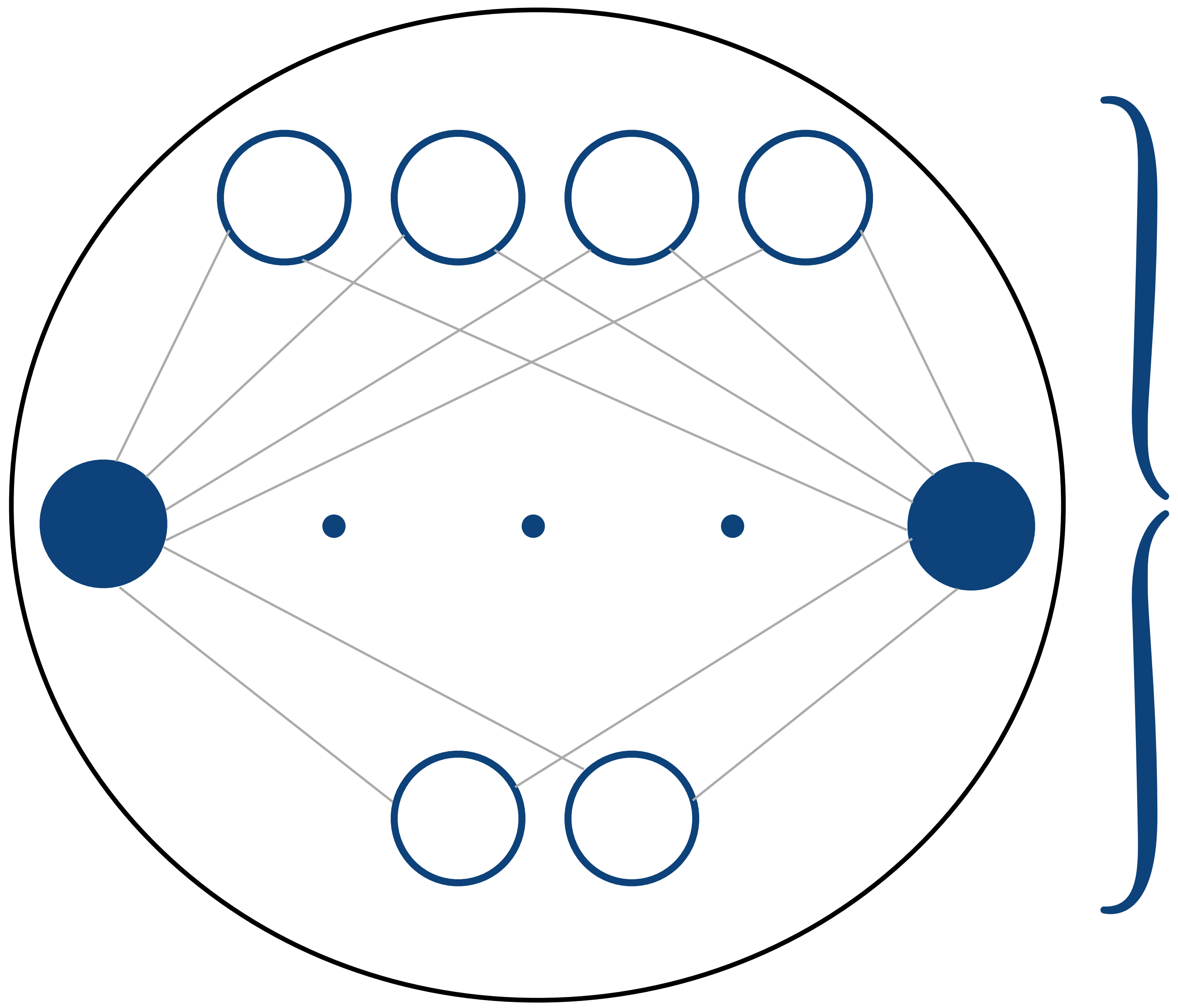
subject to:

$$\epsilon_n \leq \epsilon_{target}$$

- integer-valued cost
- real-valued error constraint

Key Idea: Quantization for efficiency is an optimization problem!

```
-0.6 <= in1 <= 9.55
-4.5 <= in2 <= 0.2
-0.06 <= in3 <= 2.11
-0.3 <= in4 <= 1.51
```



```
ensure: no overflow
```

```
res +/- 1e-3
```

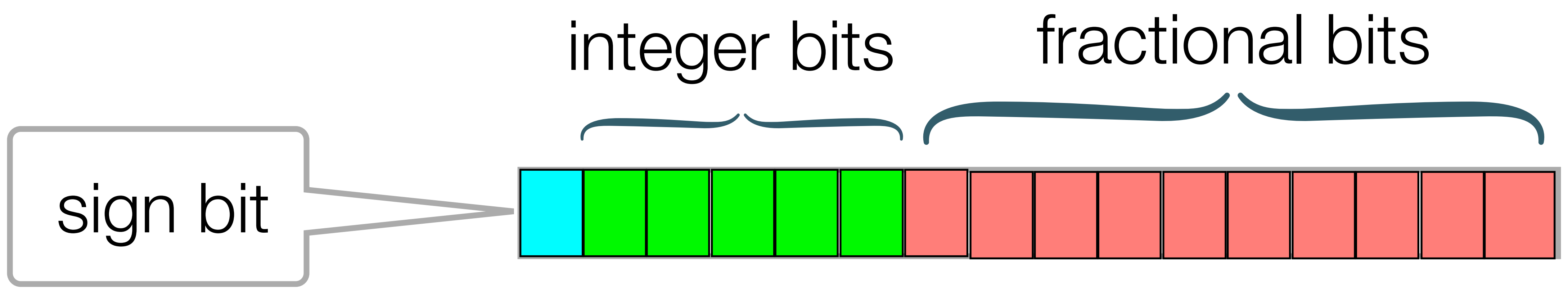
minimize: $\gamma = \sum_{i=1}^{n} \gamma_i^{dot} + \gamma_i^{bias} + \gamma_i^{\alpha}$

subject to:

$$\epsilon_n \leq \epsilon_{target}$$

$$I_i^{op} \geq \text{intBits}\left(R_i^{op} + \epsilon_i\right)$$

- integer-valued cost
- real-valued error constraint
- integer-valued range constraint



Sound Mixed Fixed-Point Quantization

mixed-integer problem

minimize: $\gamma = \sum_{i=1}^{n} \gamma_i^{dot} + \gamma_i^{bias} + \gamma_i^{\alpha}$

subject to:

$$\epsilon_n \leq \epsilon_{target}$$

$$I_i^{op} \geq \text{intBits}\left(R_i^{op} + \epsilon_i\right)$$

Sound Mixed Fixed-Point Quantization

mixed-integer problem

minimize: $\gamma = \sum_{i=1}^{n} \gamma_i^{dot} + \gamma_i^{bias} + \gamma_i^{\alpha}$

subject to:

$$\epsilon_n \leq \epsilon_{target}$$

$$I_i^{op} \geq \text{intBits}\left(R_i^{op} + \epsilon_i\right)$$

mixed-integer non-linear hard problem!

Sound Mixed Fixed-Point Quantization

mixed-integer problem

minimize: $\gamma = \sum_{i=1}^{n} \gamma_i^{dot} + \gamma_i^{bias} + \gamma_i^{\alpha}$

subject to:

$$\epsilon_n \leq \epsilon_{target}$$

$$I_i^{op} \geq \text{intBits}\left(R_i^{op} + \epsilon_i\right)$$

mixed-integer non-linear hard problem!

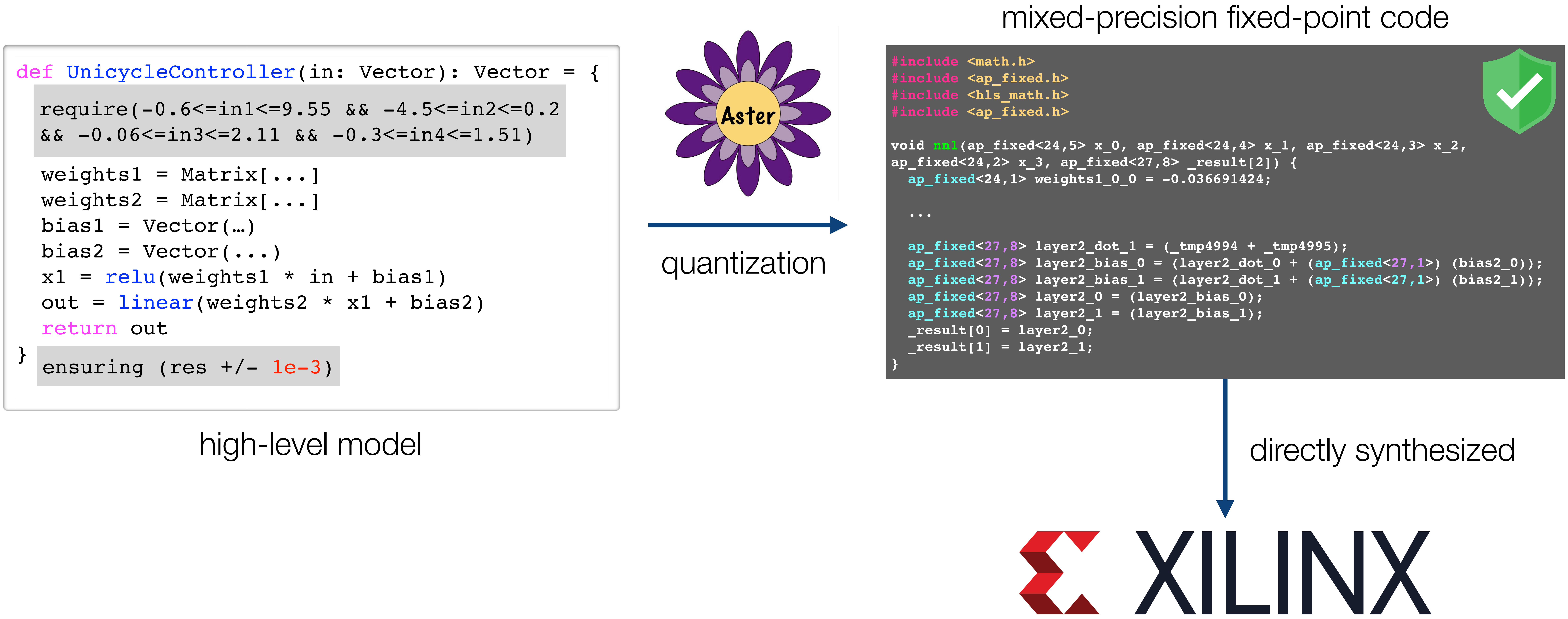
Our Solution: Reduce to Mixed Integer Linear Programming (MILP) Problem!

Aster: Sound Quantizer for NNs

```
def UnicycleController(in: Vector): Vector = {
  require(-0.6<=in1<=9.55 && -4.5<=in2<=0.2
  && -0.06<=in3<=2.11 && -0.3<=in4<=1.51)
  weights1 = Matrix[...]
  weights2 = Matrix[...]
  bias1 = Vector(…)
  bias2 = Vector(...)
  x1 = relu(weights1 * in + bias1)
  out = linear(weights2 * x1 + bias2)
  return out
} ensuring (res +/- 1e-3)
```

high-level model

Aster: Sound Quantizer for NNs



```
def UnicycleController(in: Vector): Vector = {
  require(-0.6<=in1<=9.55 && -4.5<=in2<=0.2
  && -0.06<=in3<=2.11 && -0.3<=in4<=1.51)
  weights1 = Matrix[...]
  weights2 = Matrix[...]
  bias1 = Vector(…)
  bias2 = Vector(...)
  x1 = relu(weights1 * in + bias1)
  out = linear(weights2 * x1 + bias2)
  return out
} ensuring (res +/- 1e-3)
```

high-level model

mixed-precision fixed-point code

```
#include <math.h>
#include <ap_fixed.h>
#include <hls_math.h>
#include <ap_fixed.h>

void nn1(ap_fixed<24,5> x_0, ap_fixed<24,4> x_1, ap_fixed<24,3> x_2,
ap_fixed<24,2> x_3, ap_fixed<27,8> _result[2]) {
  ap_fixed<24,1> weights1_0_0 = -0.036691424;

  ...

  ap_fixed<27,8> layer2_dot_1 = (_tmp4994 + _tmp4995);
  ap_fixed<27,8> layer2_bias_0 = (layer2_dot_0 + (ap_fixed<27,1>) (bias2_0));
  ap_fixed<27,8> layer2_bias_1 = (layer2_dot_1 + (ap_fixed<27,1>) (bias2_1));
  ap_fixed<27,8> layer2_0 = (layer2_bias_0);
  ap_fixed<27,8> layer2_1 = (layer2_bias_1);
  _result[0] = layer2_0;
  _result[1] = layer2_1;
}
```

Summary of Results: Mixed Fixed-Point Quantization of NNs

target error: 1e-3, max precision: 32-bit, TO: 5 hours

#benchmarks	#params	analysis time	
		Daisy	Aster
mid-sized (14)	60 - 3920		
large (4)	12K - 44.5K		

Summary of Results: Mixed Fixed-Point Quantization of NNs

target error: 1e-3, max precision: 32-bit, TO: 5 hours

#benchmarks	#params	analysis time	
		Daisy	Aster
mid-sized (14)	60 - 3920	4s - 2h 46m 20s	**2s - 50s**
large (4)	12K - 44.5K	TO	**12m 7s - 3h 49m 31s**

Summary of Results: Mixed Fixed-Point Quantization of NNs

target error: 1e-3, max precision: 32-bit, TO: 5 hours

#benchmarks	#params	analysis time		latency (clock-cycles)	
		Daisy	Aster	Daisy	Aster
mid-sized (14)	60 - 3920	4s - 2h 46m 20s	**2s - 50s**	12 - 178	**12 - 27**
large (4)	12K - 44.5K	TO	**12m 7s - 3h 49m 31s**	TO	**8K - 13K**

Summary of Results: Mixed Fixed-Point Quantization of NNs

target error: 1e-3, max precision: 32-bit, TO: 5 hours

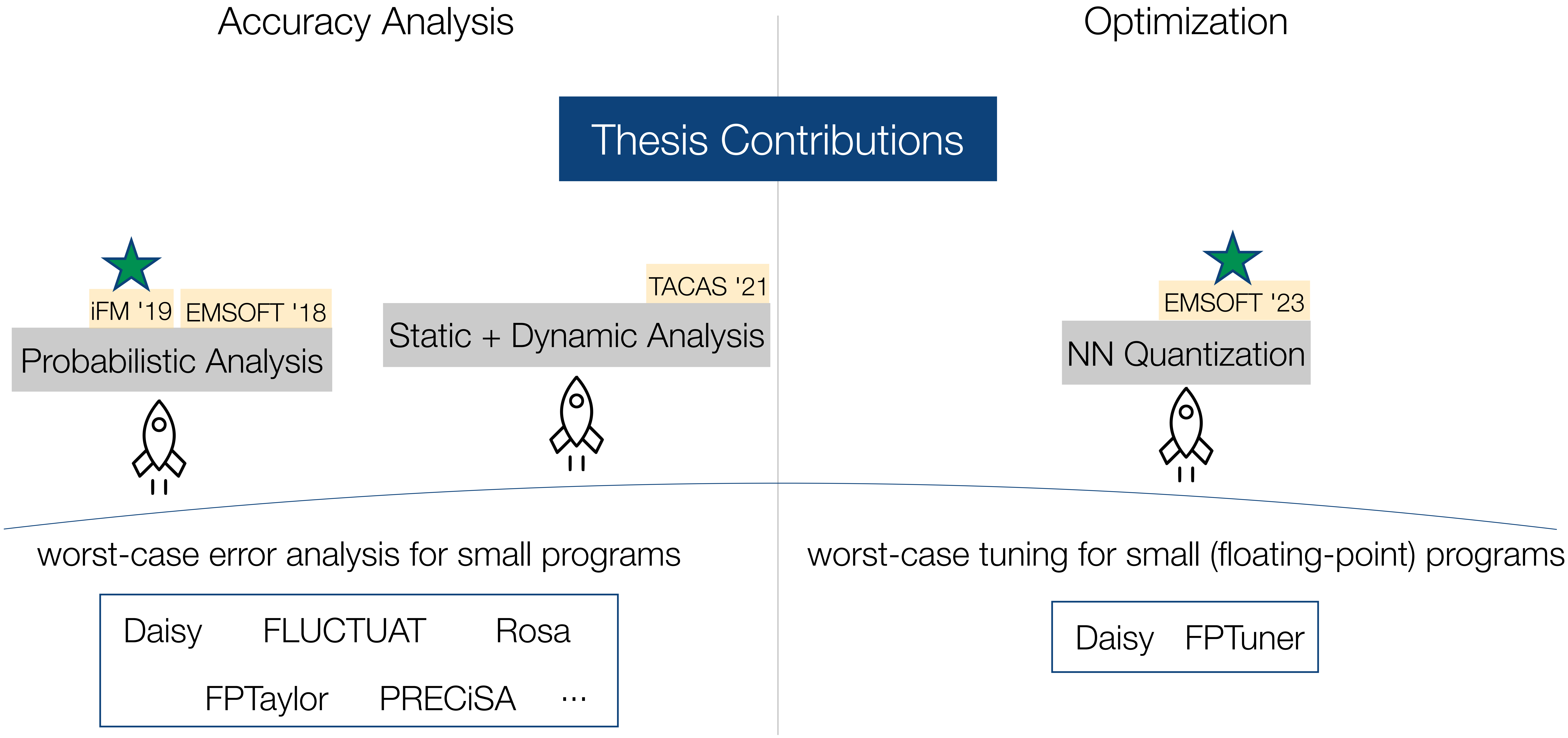
#benchmarks	#params	analysis time		latency (clock-cycles)	
		Daisy	Aster	Daisy	Aster
mid-sized (14)	60 – 3920	4s – 2h 46m 20s	**2s – 50s**	12 – 178	**12 – 27**
large (4)	12K – 44.5K	TO	**12m 7s – 3h 49m 31s**	TO	**8K – 13K**

Aster is more precise than Daisy —
Daisy reports 5 infeasibility, Aster reports **3**!

Takeaways

- Specializing optimization in application contexts can be beneficial
- Optimization with linearizations and abstractions is effective for NNs
- An automated NN quantizer: Aster zenodo
    - generates sound quantized code that can be directly synthesized in Xilinx
    - is precise and scalable

Expanding the Horizons of Finite-Precision Analysis



Accuracy Analysis

Optimization

Thesis Contributions

iFM '19 EMSOFT '18

Probabilistic Analysis

TACAS '21

Static + Dynamic Analysis

EMSOFT '23

NN Quantization

worst-case error analysis for small programs

Daisy FLUCTUAT Rosa

FPTaylor PRECiSA ···

worst-case tuning for small (floating-point) programs

Daisy FPTuner

Future Research Directions

- Scalable Accuracy Analysis
  - considering probabilistic inputs
  - by combining static, dynamic analysis and machine learning techniques

Future Research Directions

- Scalable Accuracy Analysis
  - considering probabilistic inputs
  - by combining static, dynamic analysis and machine learning techniques

- Scalable Optimization
  - considering probabilistic inputs
  - specialize in other application contexts

Future Research Directions

- Scalable Accuracy Analysis
  - considering probabilistic inputs
  - by combining static, dynamic analysis and machine learning techniques
- Scalable Optimization
  - considering probabilistic inputs
  - specialize in other application contexts
- Finite-precision in the context of
  - heterogeneous HPC systems

… and others!

Collaborators

UPPSALA UNIVERSITET

Eva Darulova

ÉCOLE POLYTECHNIQUE UNIVERSITÉ PARIS-SACLAY

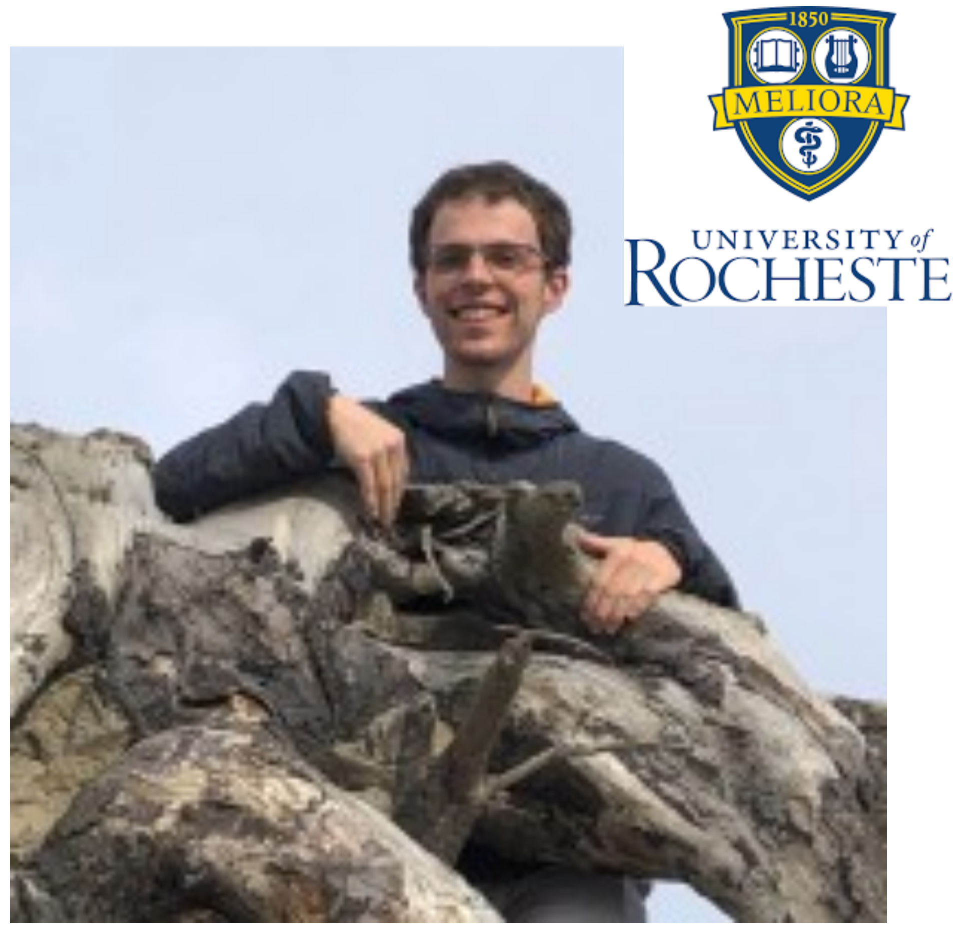
Sylvie Putot

Eric Goubault

THE UNIVERSITY OF EDINBURGH

Milos Prokop

UNIVERSITY of ROCHESTER



Joshua Sobel

MAX PLANCK INSTITUTE FOR SOFTWARE SYSTEMS

Clothilde Jeangoudoux

Maria Christakis

Nantes Université

Anastasia Volkova

Thank You

For Your Attention!

BACKUP SLIDES

Probabilistic Error Analysis

Scenario 2: Approximate Hardware Specifications

```
def controller(x:Float32, y:Float32, z:Float32): Float32 = {
 require (-15.0 <= x, y, z <= 15.0)
 val res = -x*y - 2*y*z - x - z
   return res
}ensuring (error <= 1.5e-4, 0.85)
```

resource efficient but has probabilistic error specification:

<$4 \times \epsilon_m$, 0.1>, <ϵ_m, 0.9>

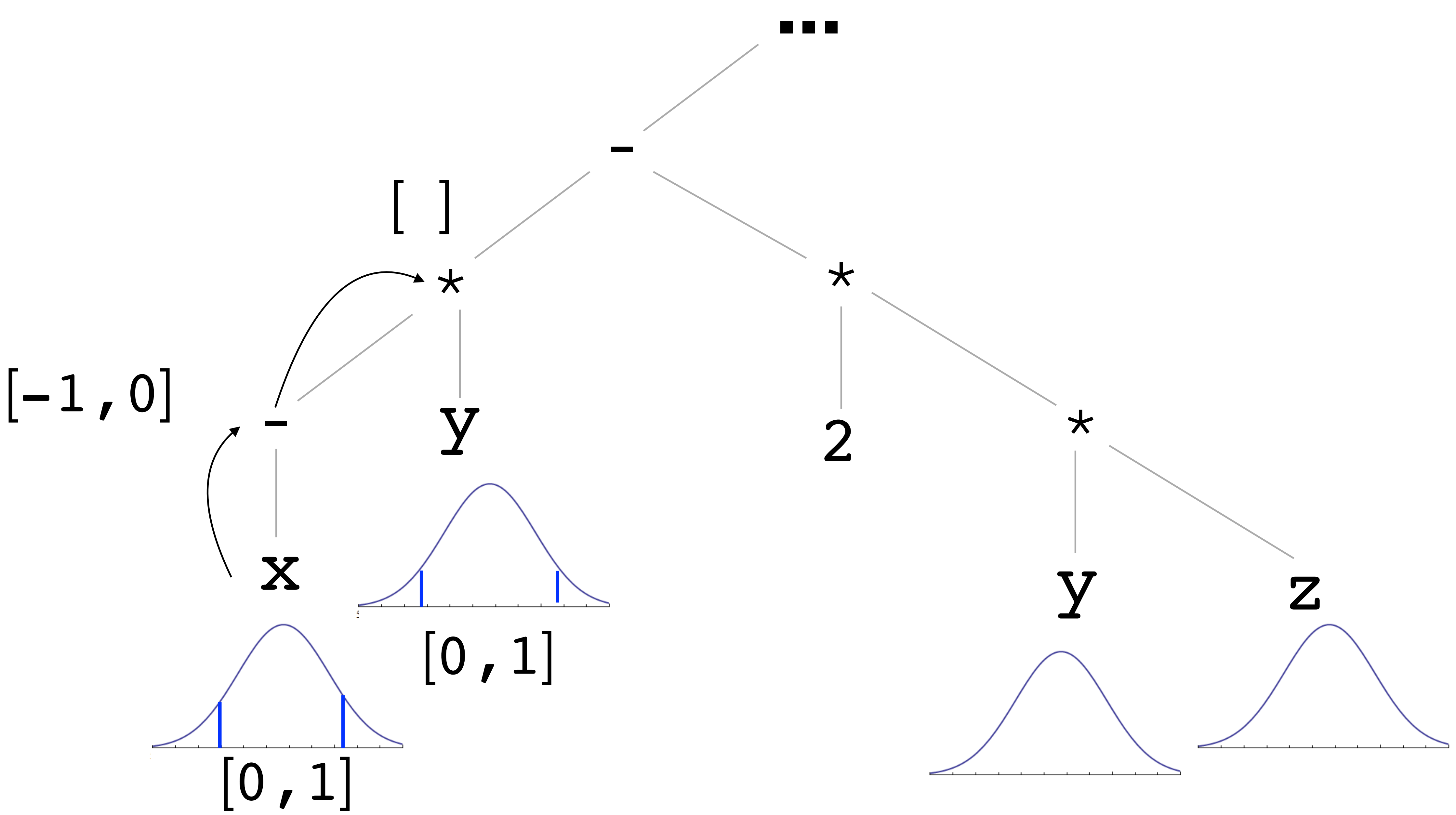
Scenario 2: Approximate Hardware Specifications

```
def controller(x:Float32, y:Float32, z:Float32): Float32 = {
 require (-15.0 <= x, y, z <= 15.0)
 val res = -x*y - 2*y*z - x - z
   return res
}ensuring (error <= 1.5e-4, 0.85)
```

resource efficient but has probabilistic error specification:
<$4 \times \epsilon_m$, 0.1>, <ϵ_m, 0.9>

The worst-case assumes $4 \times \epsilon_m$ error occurs always!

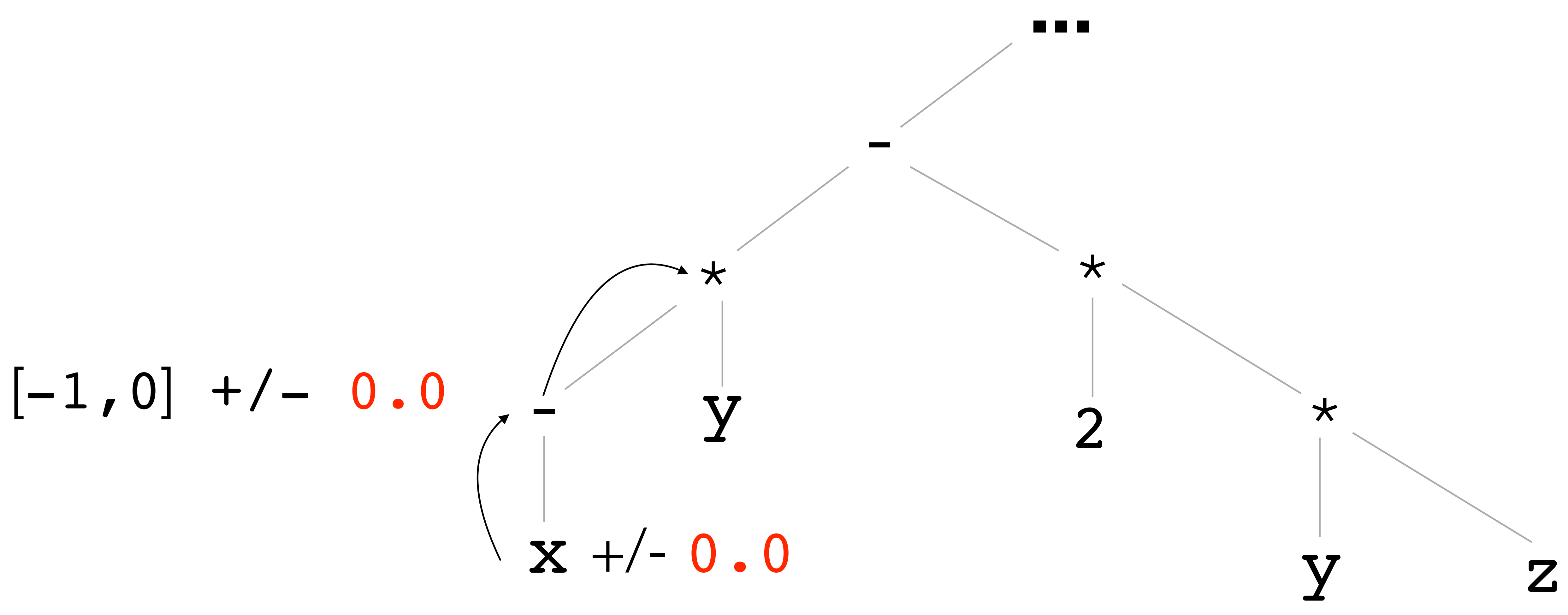
Probabilistic Error Analysis



For each arithmetic operation:

step 1: compute range for intermediate value starting with initial distributions

Probabilistic Error Analysis

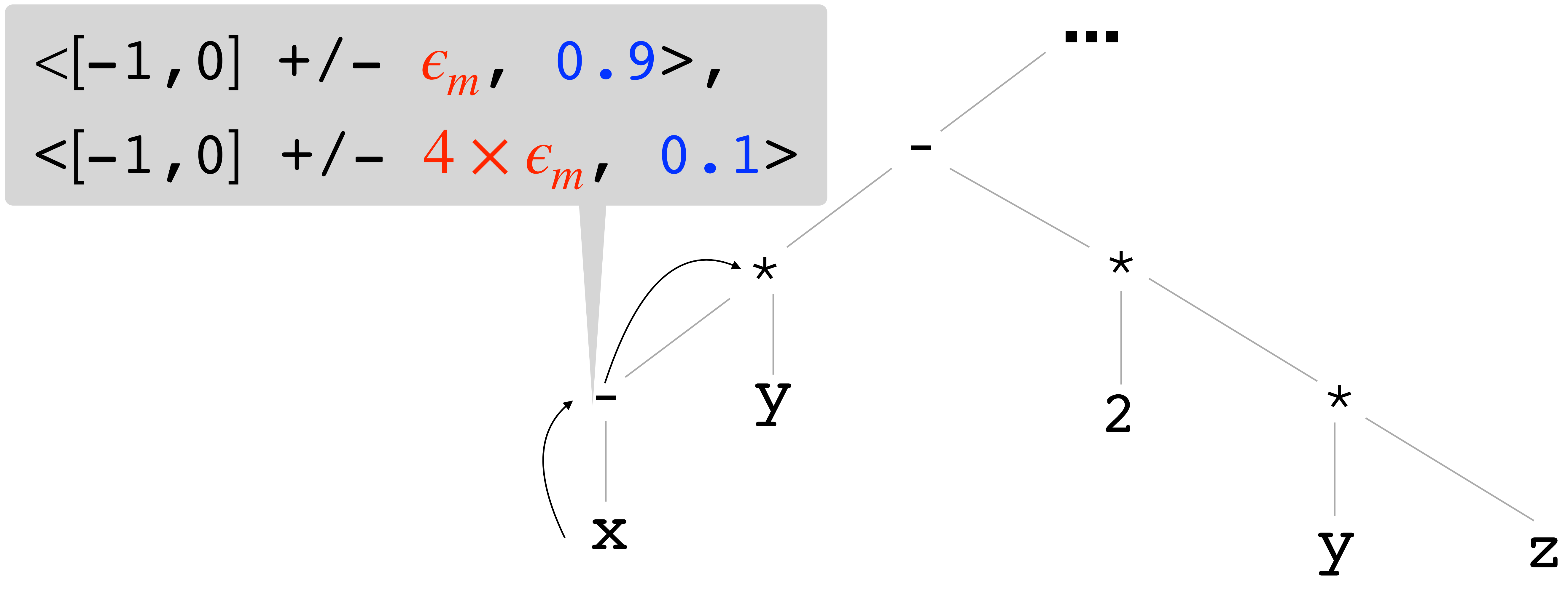


For each arithmetic operation:

step 1: compute range for intermediate value starting with initial distributions

step 2: propagate existing errors — probabilistic affine arithmetic

Probabilistic Error Analysis



<[-1,0] +/- ϵ_m, 0.9>,

<[-1,0] +/- $4 \times \epsilon_m$, 0.1>

For each arithmetic operation:

step 1: compute range for intermediate value starting with initial distributions

step 2: propagate existing errors

step 3: compute new errors — as multiple fresh noise terms

Sound Mixed Fixed-Point Quantization

Overview: Reduction to MILP

minimize: $\gamma = \sum_{i=1}^{n} \gamma_i^{dot} + \gamma_i^{bias} + \gamma_i^{\alpha}$

subject to:

$$\epsilon_n \leq \epsilon_{target}$$

$$I_i^{op} \geq \text{intBits}\left(R_i^{op} + \epsilon_i\right)$$

Overview: Reduction to MILP

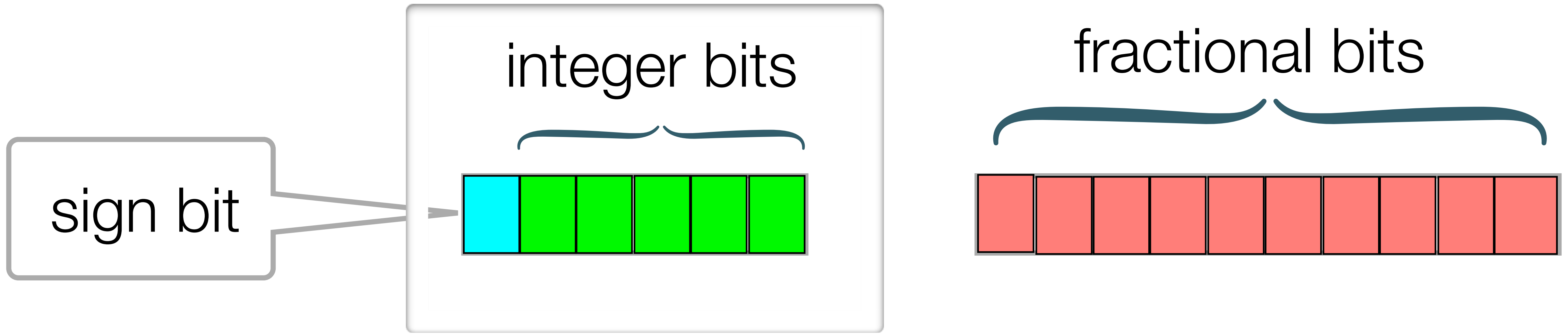
minimize: $\gamma = \sum_{i=1}^{n} \gamma_i^{dot} + \gamma_i^{bias} + \gamma_i^{\alpha}$

subject to:

$$\epsilon_n \leq \epsilon_{target}$$

$$I_i^{op} \geq \text{intBits}\left(R_i^{op} + \epsilon_i\right)$$

over-approximate integer bits separately using interval arithmetic



fixed-point representation

Overview: Reduction to MILP

minimize: $\gamma = \sum_{i=1}^{n} \gamma_i^{dot} + \gamma_i^{bias} + \gamma_i^{\alpha}$

subject to:

$$\epsilon_n \leq \epsilon_{target}$$

$$I_i^{op} \geq \text{intBits}\left(R_i^{op} + \epsilon_i\right)$$

linearize exactly with additional constraints

- over-approximate integer bits separately

Linearization Step 2: Exact Linearization of Cost

minimize: $\gamma = \sum_{i=1}^{n} \gamma_i^{dot} + \gamma_i^{bias} + \gamma_i^{\alpha}$

subject to:

$$\epsilon_n \leq \epsilon_{target}$$

$$I_i^{op} \geq \text{intBits}\left(R_i^{op} + \epsilon_i\right)$$

$$\gamma_i^{bias} = \max(\pi_i^{dot}, \pi_i^{bias})$$

non-linear function

Linearization Step 2: Exact Linearization of Cost

minimize: $\gamma = \sum_{i=1}^{n} \gamma_i^{dot} + \gamma_i^{bias} + \gamma_i^{\alpha}$

subject to:

$$\epsilon_n \leq \epsilon_{target}$$

$$I_i^{op} \geq \text{intBits}\left(R_i^{op} + \epsilon_i\right)$$

$$\gamma_i^{bias} = \text{max}(\pi_i^{dot}, \pi_i^{bias})$$

$$\downarrow$$

c1: $\gamma_i^{bias} \geq \pi_i^{dot}$

c2: $\gamma_i^{bias} \geq \pi_i^{bias}$

Overview: Reduction to MILP

abstract dot product assuming a precision and correcting later

minimize: $\gamma = \sum_{i=1}^{n} \gamma_i^{dot} + \gamma_i^{bias} + \gamma_i^{\alpha}$

subject to:

$$\epsilon_n \leq \epsilon_{target}$$

$$I_i^{op} \geq \text{intBits}\left(R_i^{op} + \epsilon_i\right)$$

- over-approximate integer bits separately
- linearize bias cost and error constraint exactly

Overview: Reduction to MILP

Linearized Problem

minimize: $\gamma = \sum_{i=1}^{n} \gamma_i^{dot} + \gamma_i^{bias} + \gamma_i^{\alpha}$

subject to:

$$\epsilon_n \leq \epsilon_{target}$$

$$I_i^{op} \geq \text{intBits}\left(R_i^{op} + \epsilon_i\right)$$

- over-approximate integer bits separately
- linearize bias cost and error constraint exactly
- abstract dot product

Optimizing Fractional Bits for Dot and Bias Products

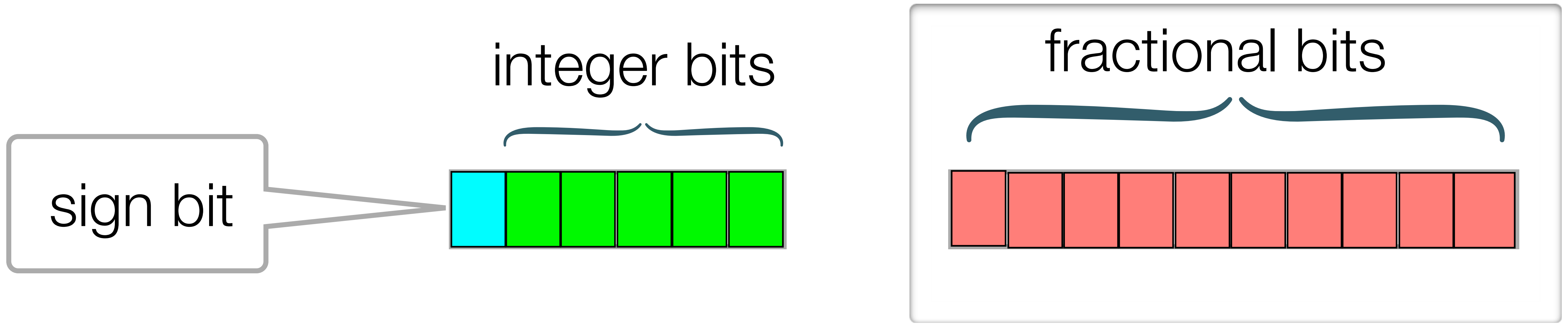
Linearized Problem

minimize: $\gamma = \sum_{i=1}^{n} \gamma_i^{dot} + \gamma_i^{bias} + \gamma_i^{\alpha}$

subject to:

$$\epsilon_n \leq \epsilon_{target}$$

$$I_i^{op} \geq \text{intBits}\left(R_i^{op} + \epsilon_i\right)$$



Optimizing Fractional Bits for Dot and Bias Products

Linearized Problem

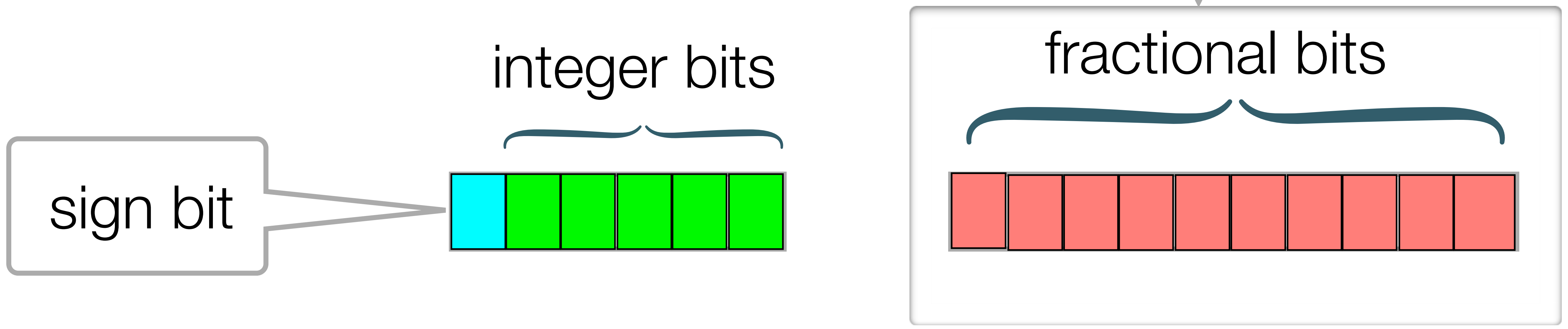
minimize: $\gamma = \sum_{i=1}^{n} \gamma_i^{dot} + \gamma_i^{bias} + \gamma_i^{\alpha}$

subject to:

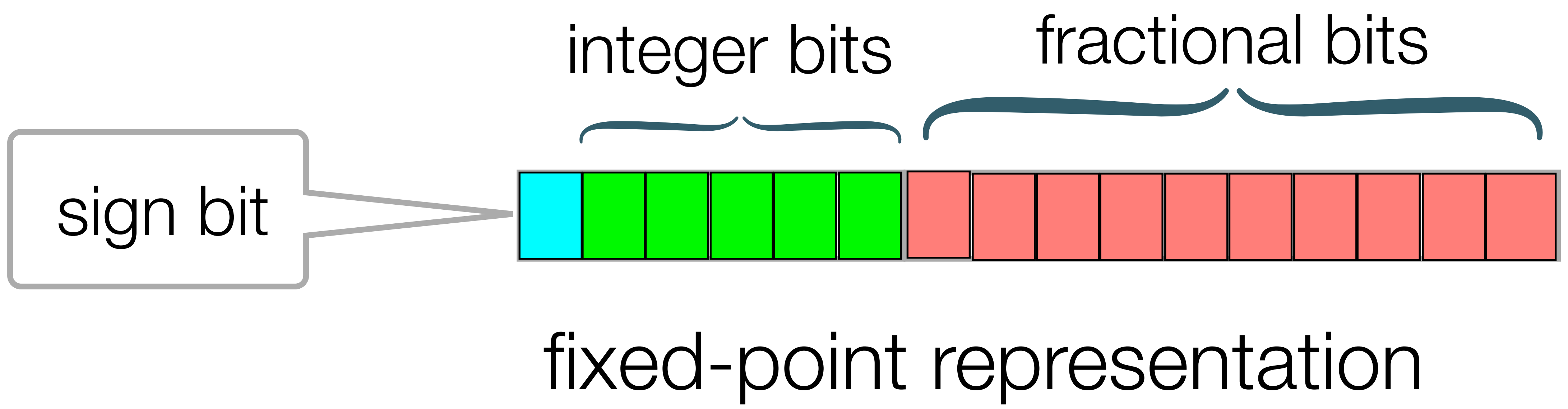
$$\epsilon_n \leq \epsilon_{target}$$

$$I_i^{op} \geq \text{intBits}\left(R_i^{op} + \epsilon_i\right)$$

MILP solver

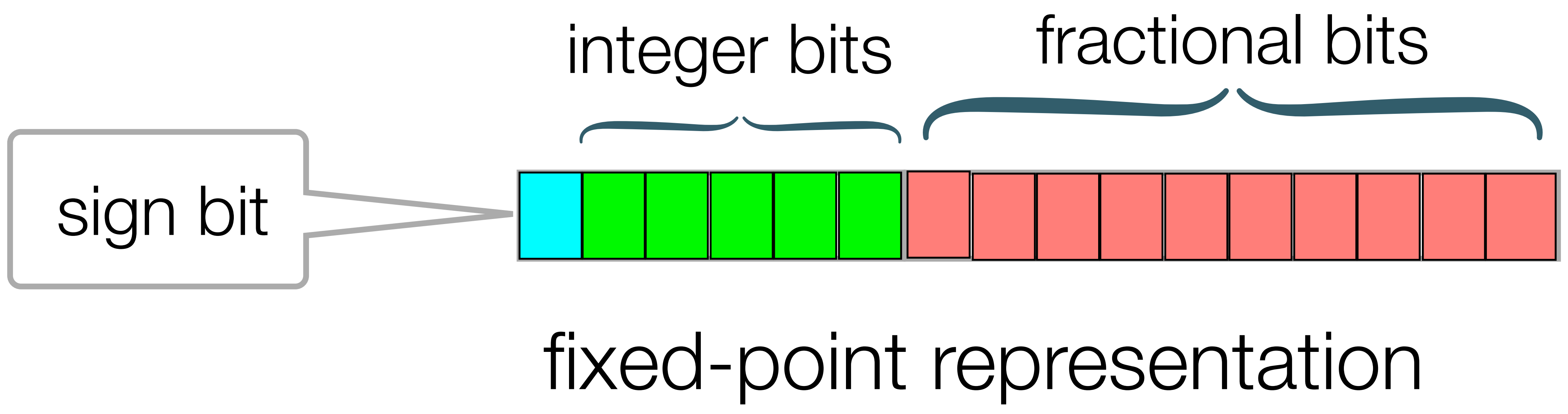


Generate Full-Fledged Quantized Implementation



- reduced to MILP problem
- optimized fractional bits for dot and bias results assuming precision of weights
- assigning correctly rounded precision for all variables and constants

Generate Full-Fledged Quantized Implementation



- reduced to MILP problem
- optimized fractional bits for dot and bias results assuming precision of weights
- assigning correctly rounded precision for all variables and constants

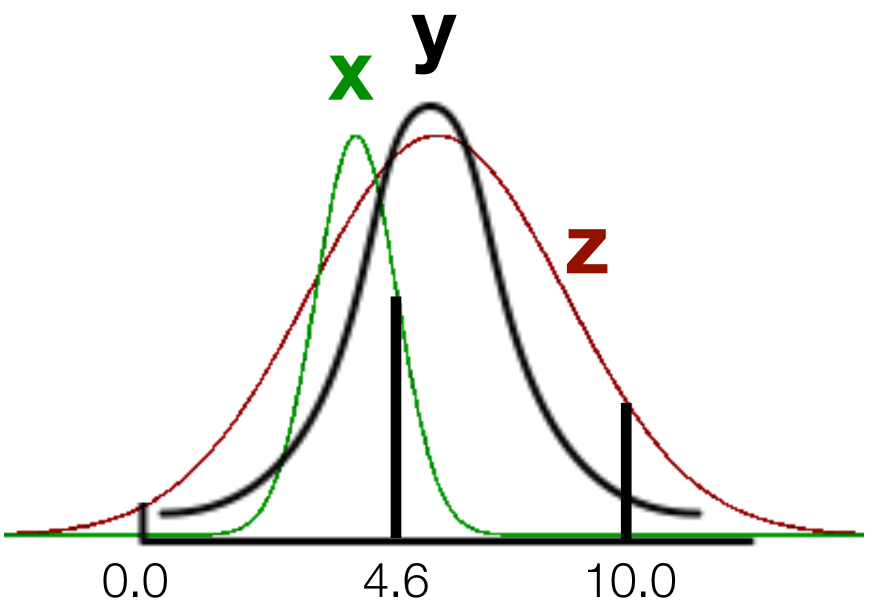
using fixed-point sum of products by constants*

* A Correctly-Rounded Fixed-Point-Arithmetic DotProduct Algorithm, Sylvie Boldo, Diane Gallois-Wong, and Thibault Hilaire, ARITH 2020

Discrete Choice in the Presence of Numerical Uncertainties

Scenario 1: Wrong Discrete Decisions

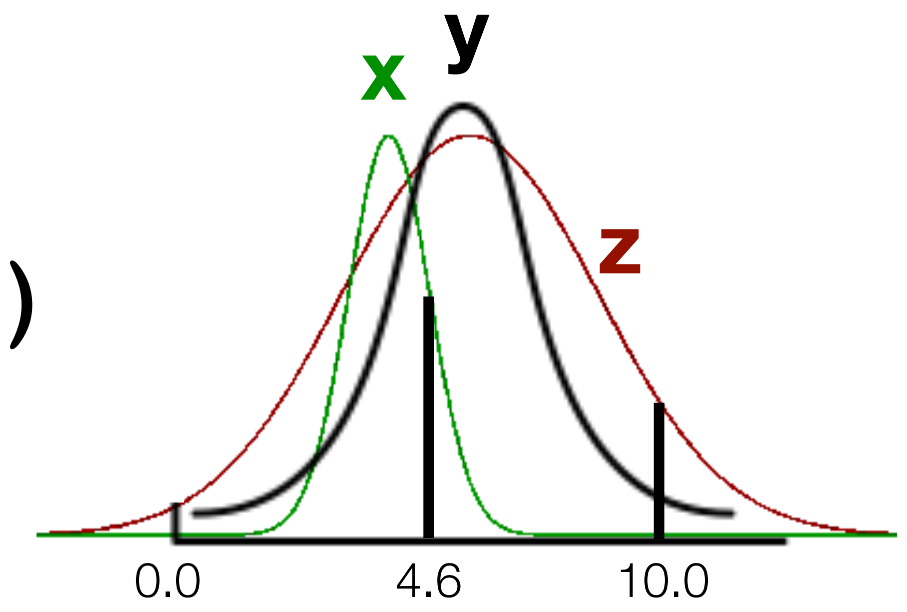
```
def controller(x:Float32, y:Float32, z:Float32): Float32 = {
 require (0.0 <= x <= 4.6 &&  0.0 <= y, z <= 10.0)

  x:= gaussian(4.0, 0.5)
  y:= gaussian(4.75, 2.0)
  z:= gaussian(4.8, 2.5)

  val res = -x*y - 2*y*z - x - z


  if (res <= 0.0)
    raise_alarm()
  else
    do_nothing()
}
```



real-valued execution

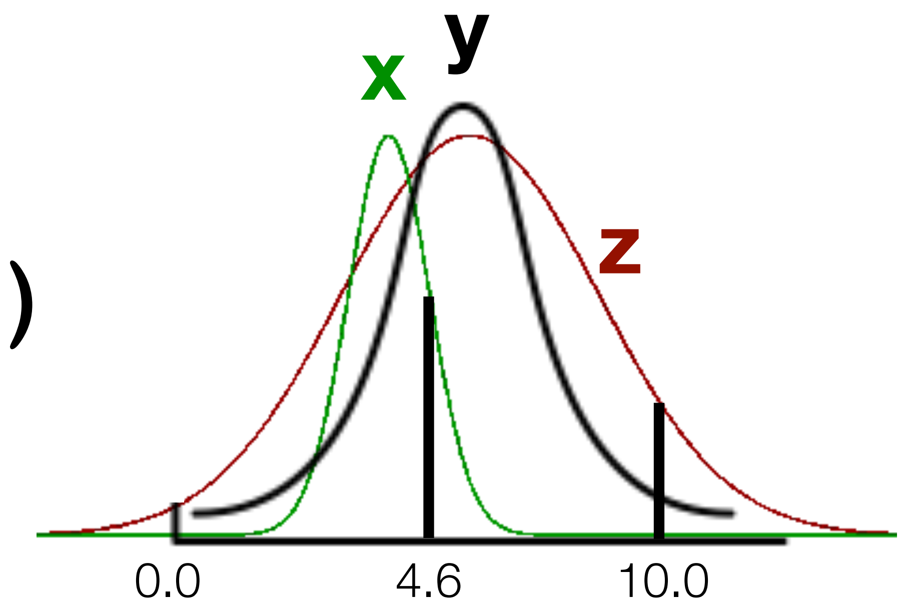
Scenario 1: Wrong Discrete Decisions

```
def controller(x:Float32, y:Float32, z:Float32): Float32 = {
 require (0.0 <= x <= 4.6 &&  0.0 <= y, z <= 10.0)

  x:= gaussian(4.0, 0.5)
  y:= gaussian(4.75, 2.0)
  z:= gaussian(4.8, 2.5)

  val res = -x*y - 2*y*z - x - z


  if (res <= 0.0)
    raise_alarm()  finite-precision execution
  else
    do_nothing()  real-valued execution
}
```



Scenario 1: Wrong Discrete Decisions

```
def controller(x:Float32, y:Float32, z:Float32): Float32 = {
 require (0.0 <= x <= 4.6 &&  0.0 <= y, z <= 10.0)

  x:= gaussian(4.0, 0.5)
  y:= gaussian(4.75, 2.0)
  z:= gaussian(4.8, 2.5)

  val res = -x*y - 2*y*z - x - z


  if (res <= 0.0)
    raise_alarm()
  else
    do_nothing()
}
```



finite-precision execution

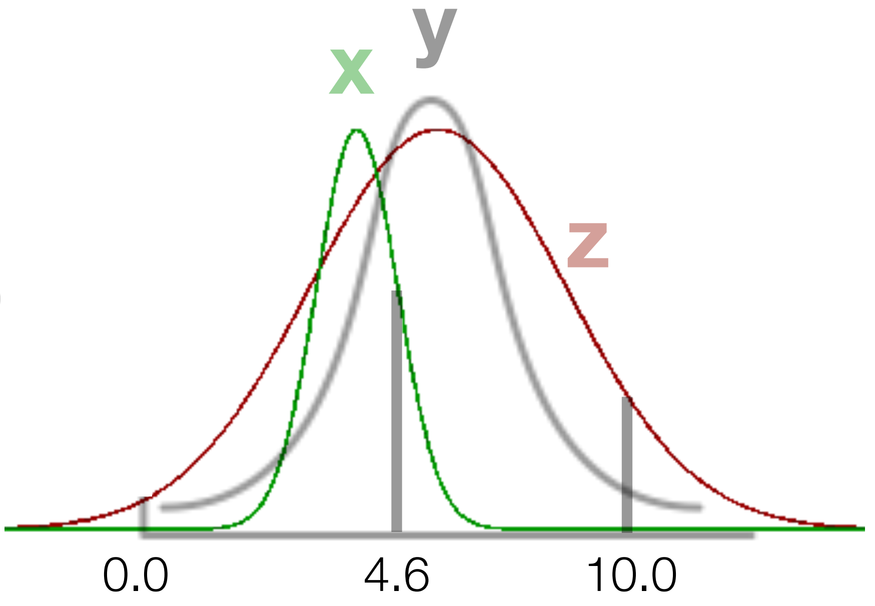
real-valued execution

Program always takes the wrong decision in the worst-case!

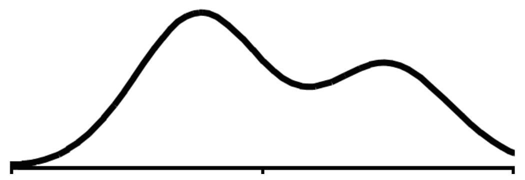
Probabilistic Analysis for Discrete Decisions

EMSOFT ’18

```
def controller(x:Float32, y:Float32, z:Float32): Float32 = {
 require (0.0 <= x <= 4.6 &&  0.0 <= y, z <= 10.0)

  x:= gaussian(4.0, 0.5)
  y:= gaussian(4.75, 2.0)
  z:= gaussian(4.8, 2.5)
```



```
  val res = -x*y - 2*y*z - x - z
```



```
  if (res <= 0.0)
    raise_alarm()
  else
    do_nothing()
}
```

how often?

Summary: Probabilistic Analysis for Discrete Decisions

scalability of probabilistic analysis for numerical programs

#benchmark	#ops	#vars	uniform	gaussian	over-approx.
24	4-25	1-9	48s-7m 28s	42s-11m 1s	~e-4(7)*

* compared our analysis with symbolic inference

zenodo https://doi.org/10.5281/zenodo.8042198

Summary: Probabilistic Analysis for Discrete Decisions

scalability of probabilistic analysis for numerical programs

#benchmark	#ops	#vars	uniform	gaussian	over-approx.
24	4-25	1-9	48s-7m 28s	42s-11m 1s	~e-4(7)*

* compared our analysis with symbolic inference

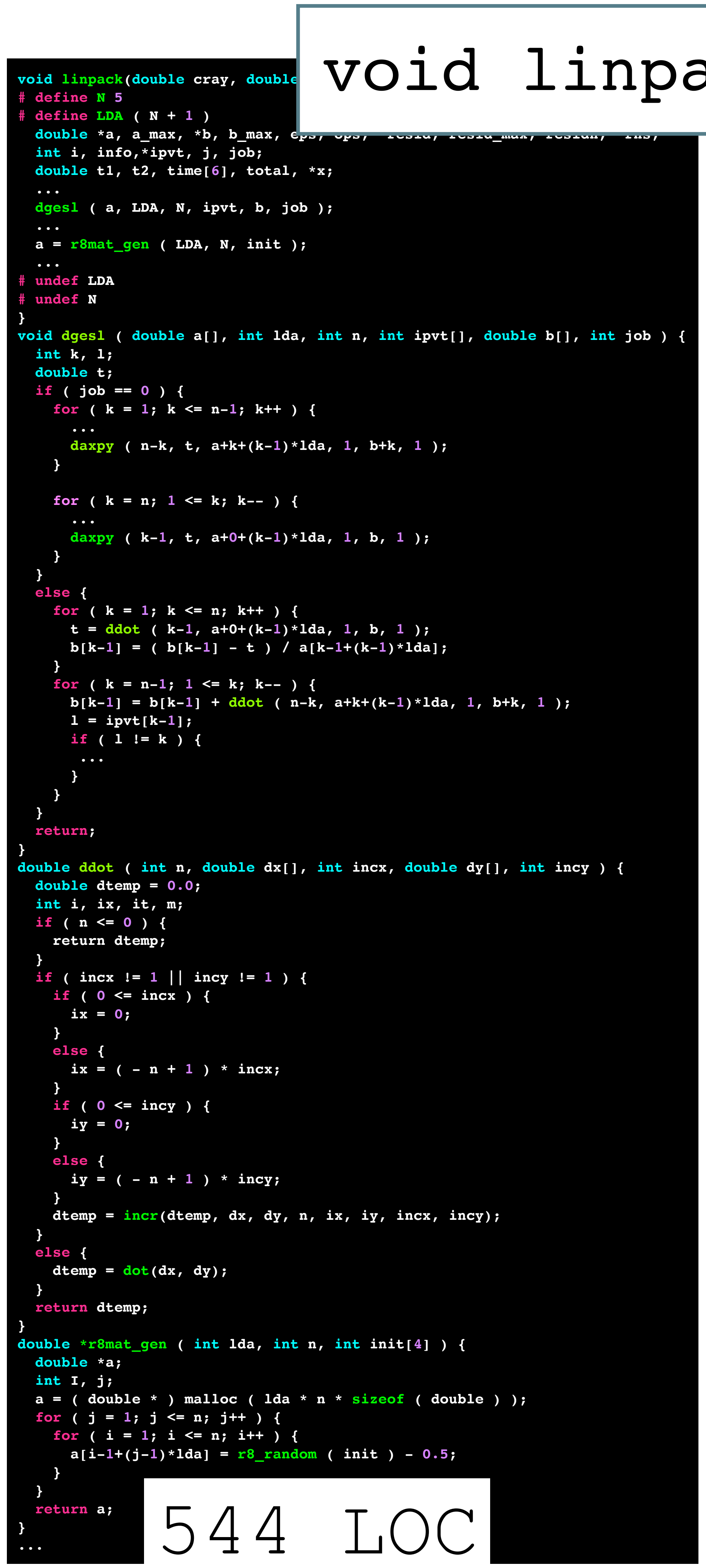
zenodo https://doi.org/10.5281/zenodo.8042198

Sound and precise WPPs for small programs with different distributions

Two-Phase Approach for Conditional Floating-Point Verification

Large Floating-Point Applications

```
void linpack(double cray, double init[4])
```

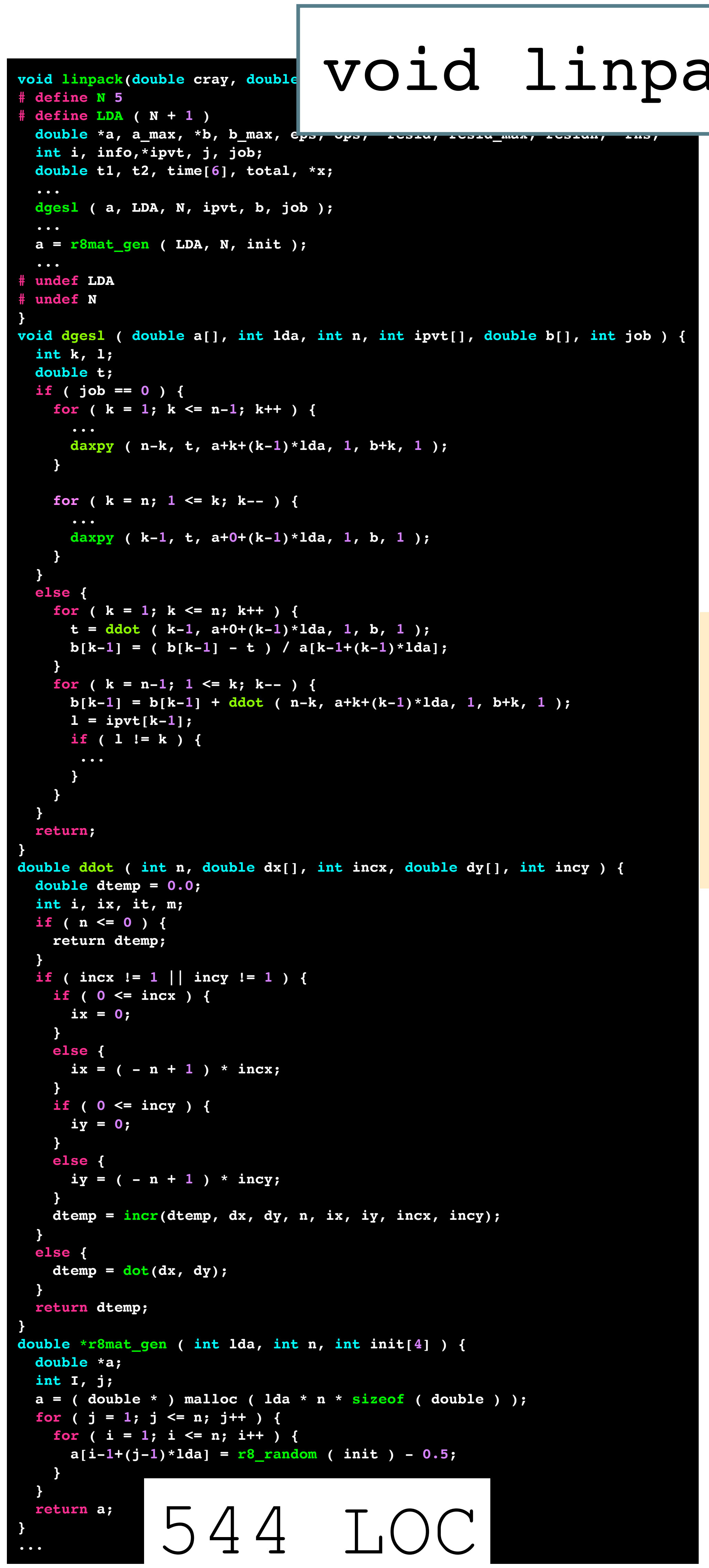


```
void linpack(double cray, double
# define N 5
# define LDA ( N + 1 )
  double *a, a_max, *b, b_max, eps, ops, resid, resid_max, residn, rhs;
  int i, info,*ipvt, j, job;
  double t1, t2, time[6], total, *x;
  ...
  dgesl ( a, LDA, N, ipvt, b, job );
  ...
  a = r8mat_gen ( LDA, N, init );
  ...
# undef LDA
# undef N
}
void dgesl ( double a[], int lda, int n, int ipvt[], double b[], int job ) {
  int k, l;
  double t;
  if ( job == 0 ) {
    for ( k = 1; k <= n-1; k++ ) {
      ...
      daxpy ( n-k, t, a+k+(k-1)*lda, 1, b+k, 1 );
    }

    for ( k = n; 1 <= k; k-- ) {
      ...
      daxpy ( k-1, t, a+0+(k-1)*lda, 1, b, 1 );
    }
  }
  else {
    for ( k = 1; k <= n; k++ ) {
      t = ddot ( k-1, a+0+(k-1)*lda, 1, b, 1 );
      b[k-1] = ( b[k-1] - t ) / a[k-1+(k-1)*lda];
    }
    for ( k = n-1; 1 <= k; k-- ) {
      b[k-1] = b[k-1] + ddot ( n-k, a+k+(k-1)*lda, 1, b+k, 1 );
      l = ipvt[k-1];
      if ( l != k ) {
       ...
      }
    }
  }
  return;
}
double ddot ( int n, double dx[], int incx, double dy[], int incy ) {
  double dtemp = 0.0;
  int i, ix, it, m;
  if ( n <= 0 ) {
    return dtemp;
  }
  if ( incx != 1 || incy != 1 ) {
    if ( 0 <= incx ) {
      ix = 0;
    }
    else {
      ix = ( - n + 1 ) * incx;
    }
    if ( 0 <= incy ) {
      iy = 0;
    }
    else {
      iy = ( - n + 1 ) * incy;
    }
    dtemp = incr(dtemp, dx, dy, n, ix, iy, incx, incy);
  }
  else {
    dtemp = dot(dx, dy);
  }
  return dtemp;
}
double *r8mat_gen ( int lda, int n, int init[4] ) {
  double *a;
  int I, j;
  a = ( double * ) malloc ( lda * n * sizeof ( double ) );
  for ( j = 1; j <= n; j++ ) {
    for ( i = 1; i <= n; i++ ) {
      a[i-1+(j-1)*lda] = r8_random ( init ) - 0.5;
    }
  }
  return a;
}
...
```

544 LOC

Large Floating-Point Applications

```
void linpack(double cray, double init[4])
```

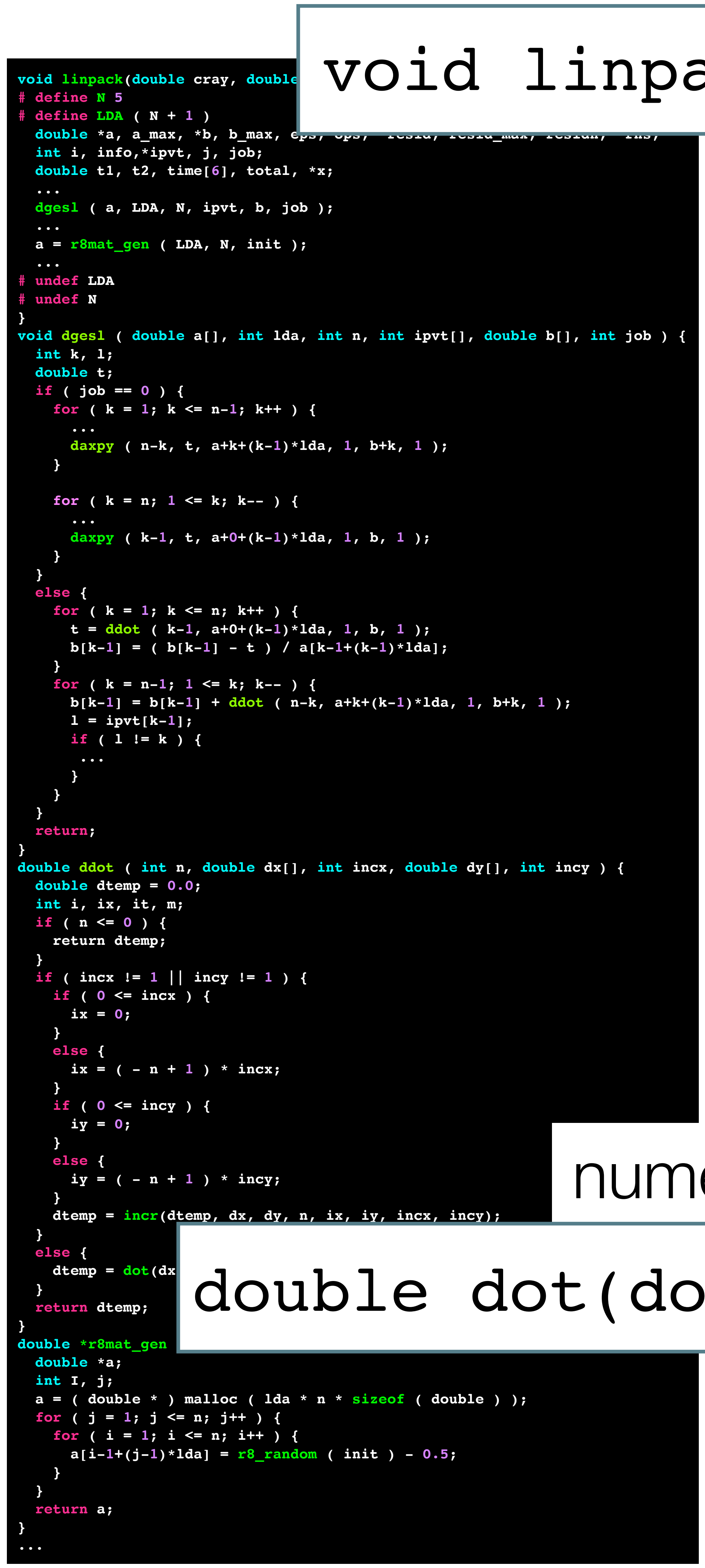


```
void linpack(double cray, double
# define N 5
# define LDA ( N + 1 )
  double *a, a_max, *b, b_max, eps, ops, resid, resid_max, residn, rhs;
  int i, info,*ipvt, j, job;
  double t1, t2, time[6], total, *x;
  ...
  dgesl ( a, LDA, N, ipvt, b, job );
  ...
  a = r8mat_gen ( LDA, N, init );
  ...
# undef LDA
# undef N
}
void dgesl ( double a[], int lda, int n, int ipvt[], double b[], int job ) {
  int k, l;
  double t;
  if ( job == 0 ) {
    for ( k = 1; k <= n-1; k++ ) {
      ...
      daxpy ( n-k, t, a+k+(k-1)*lda, 1, b+k, 1 );
    }

    for ( k = n; 1 <= k; k-- ) {
      ...
      daxpy ( k-1, t, a+0+(k-1)*lda, 1, b, 1 );
    }
  }
  else {
    for ( k = 1; k <= n; k++ ) {
      t = ddot ( k-1, a+0+(k-1)*lda, 1, b, 1 );
      b[k-1] = ( b[k-1] - t ) / a[k-1+(k-1)*lda];
    }
    for ( k = n-1; 1 <= k; k-- ) {
      b[k-1] = b[k-1] + ddot ( n-k, a+k+(k-1)*lda, 1, b+k, 1 );
      l = ipvt[k-1];
      if ( l != k ) {
       ...
      }
    }
  }
  return;
}
double ddot ( int n, double dx[], int incx, double dy[], int incy ) {
  double dtemp = 0.0;
  int i, ix, it, m;
  if ( n <= 0 ) {
    return dtemp;
  }
  if ( incx != 1 || incy != 1 ) {
    if ( 0 <= incx ) {
      ix = 0;
    }
    else {
      ix = ( - n + 1 ) * incx;
    }
    if ( 0 <= incy ) {
      iy = 0;
    }
    else {
      iy = ( - n + 1 ) * incy;
    }
    dtemp = incr(dtemp, dx, dy, n, ix, iy, incx, incy);
  }
  else {
    dtemp = dot(dx, dy);
  }
  return dtemp;
}
double *r8mat_gen ( int lda, int n, int init[4] ) {
  double *a;
  int I, j;
  a = ( double * ) malloc ( lda * n * sizeof ( double ) );
  for ( j = 1; j <= n; j++ ) {
    for ( i = 1; i <= n; i++ ) {
      a[i-1+(j-1)*lda] = r8_random ( init ) - 0.5;
    }
  }
  return a;
}
...
```

544 LOC

Numerical analyzers do not scale!

Kernels in Large Floating-Point Applications



```
void linpack(double cray, double
# define N 5
# define LDA ( N + 1 )
  double *a, a_max, *b, b_max, eps, ops, resid, resid_max, residn, rhs;
  int i, info,*ipvt, j, job;
  double t1, t2, time[6], total, *x;
  ...
  dgesl ( a, LDA, N, ipvt, b, job );
  ...
  a = r8mat_gen ( LDA, N, init );
  ...
# undef LDA
# undef N
}
void dgesl ( double a[], int lda, int n, int ipvt[], double b[], int job ) {
  int k, l;
  double t;
  if ( job == 0 ) {
    for ( k = 1; k <= n-1; k++ ) {
      ...
      daxpy ( n-k, t, a+k+(k-1)*lda, 1, b+k, 1 );
    }

    for ( k = n; 1 <= k; k-- ) {
      ...
      daxpy ( k-1, t, a+0+(k-1)*lda, 1, b, 1 );
    }
  }
  else {
    for ( k = 1; k <= n; k++ ) {
      t = ddot ( k-1, a+0+(k-1)*lda, 1, b, 1 );
      b[k-1] = ( b[k-1] - t ) / a[k-1+(k-1)*lda];
    }
    for ( k = n-1; 1 <= k; k-- ) {
      b[k-1] = b[k-1] + ddot ( n-k, a+k+(k-1)*lda, 1, b+k, 1 );
      l = ipvt[k-1];
      if ( l != k ) {
       ...
      }
    }
  }
  return;
}
double ddot ( int n, double dx[], int incx, double dy[], int incy ) {
  double dtemp = 0.0;
  int i, ix, it, m;
  if ( n <= 0 ) {
    return dtemp;
  }
  if ( incx != 1 || incy != 1 ) {
    if ( 0 <= incx ) {
      ix = 0;
    }
    else {
      ix = ( - n + 1 ) * incx;
    }
    if ( 0 <= incy ) {
      iy = 0;
    }
    else {
      iy = ( - n + 1 ) * incy;
    }
    dtemp = incr(dtemp, dx, dy, n, ix, iy, incx, incy);
  }
  else {
    dtemp = dot(dx
  }
  return dtemp;
}
double *r8mat_gen
  double *a;
  int I, j;
  a = ( double * ) malloc ( lda * n * sizeof ( double ) );
  for ( j = 1; j <= n; j++ ) {
    for ( i = 1; i <= n; i++ ) {
      a[i-1+(j-1)*lda] = r8_random ( init ) - 0.5;
    }
  }
  return a;
}
...
```

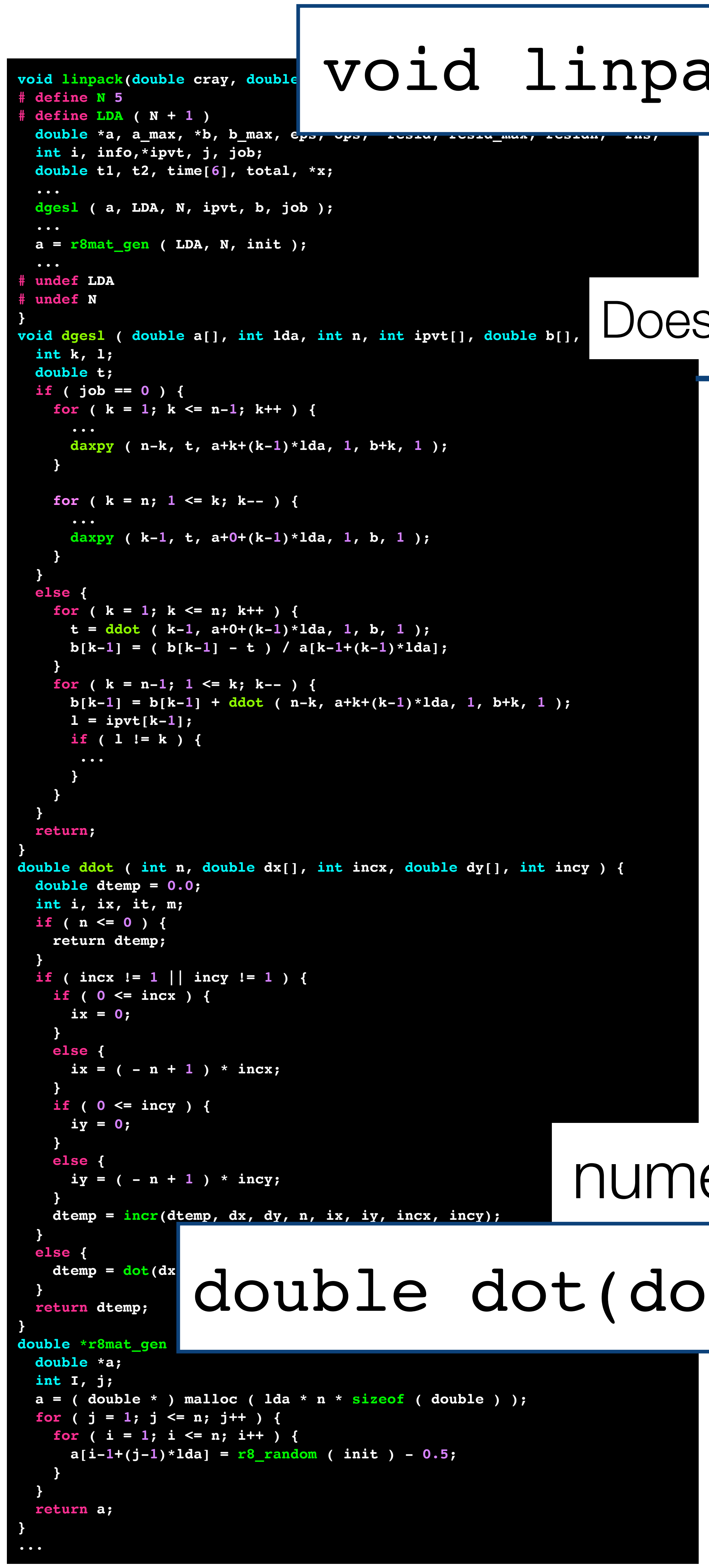
```
void linpack(double cray, double init[4])
```

numerically interesting: numerical kernel

```
double dot(double dx[4], double dy[4])
```

(Conditional) Floating-Point Verification

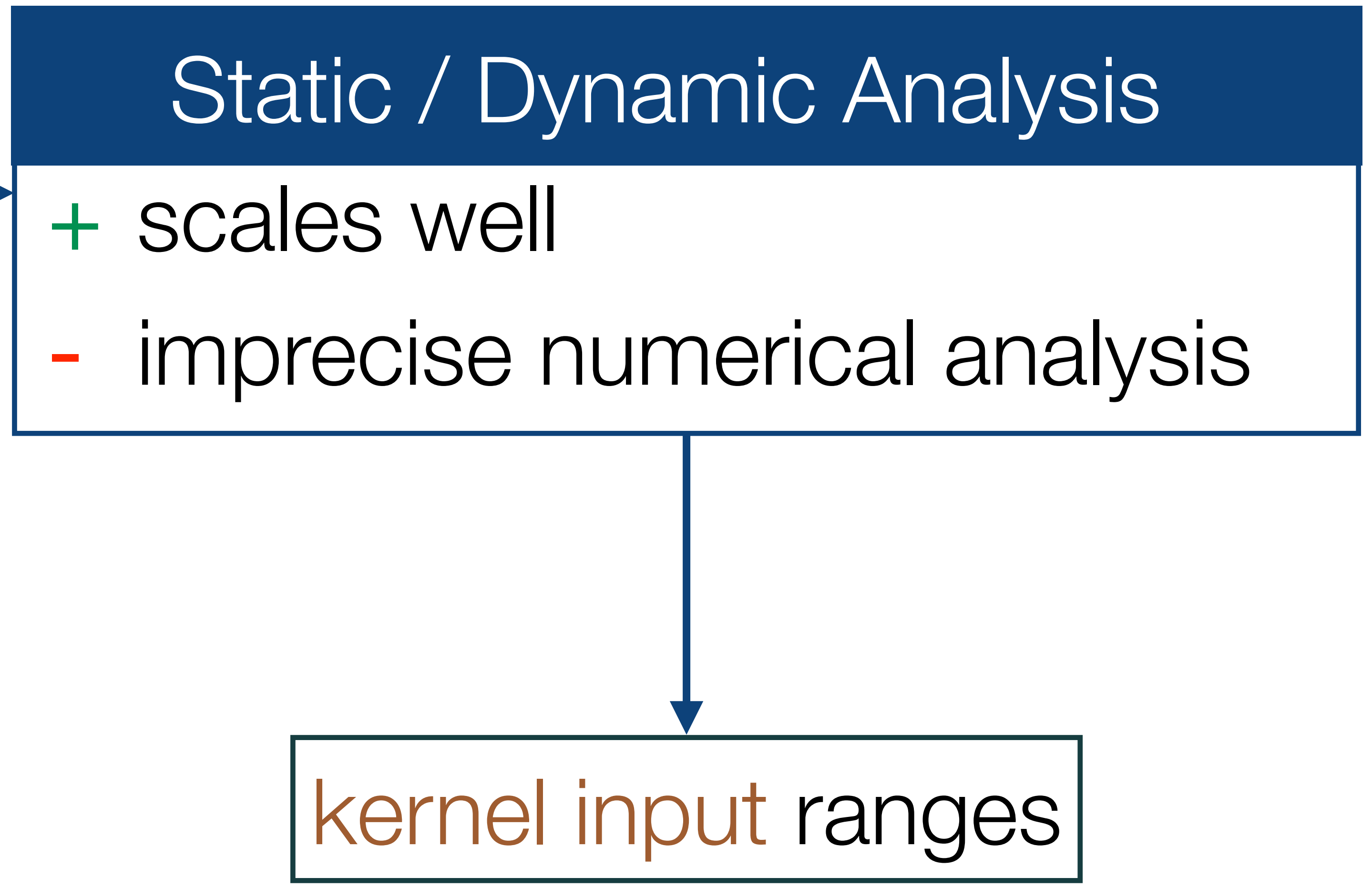
```
void linpack(double cray, double init[4])
```



```
void linpack(double cray, double
# define N 5
# define LDA ( N + 1 )
  double *a, a_max, *b, b_max, eps, ops, resid, resid_max, residn, rhs;
  int i, info,*ipvt, j, job;
  double t1, t2, time[6], total, *x;
  ...
  dgesl ( a, LDA, N, ipvt, b, job );
  ...
  a = r8mat_gen ( LDA, N, init );
  ...
# undef LDA
# undef N
}
void dgesl ( double a[], int lda, int n, int ipvt[], double b[],
  int k, l;
  double t;
  if ( job == 0 ) {
    for ( k = 1; k <= n-1; k++ ) {
      ...
      daxpy ( n-k, t, a+k+(k-1)*lda, 1, b+k, 1 );
    }

    for ( k = n; 1 <= k; k-- ) {
      ...
      daxpy ( k-1, t, a+0+(k-1)*lda, 1, b, 1 );
    }
  }
  else {
    for ( k = 1; k <= n; k++ ) {
      t = ddot ( k-1, a+0+(k-1)*lda, 1, b, 1 );
      b[k-1] = ( b[k-1] - t ) / a[k-1+(k-1)*lda];
    }
    for ( k = n-1; 1 <= k; k-- ) {
      b[k-1] = b[k-1] + ddot ( n-k, a+k+(k-1)*lda, 1, b+k, 1 );
      l = ipvt[k-1];
      if ( l != k ) {
       ...
      }
    }
  }
  return;
}
double ddot ( int n, double dx[], int incx, double dy[], int incy ) {
  double dtemp = 0.0;
  int i, ix, it, m;
  if ( n <= 0 ) {
    return dtemp;
  }
  if ( incx != 1 || incy != 1 ) {
    if ( 0 <= incx ) {
      ix = 0;
    }
    else {
      ix = ( - n + 1 ) * incx;
    }
    if ( 0 <= incy ) {
      iy = 0;
    }
    else {
      iy = ( - n + 1 ) * incy;
    }
    dtemp = incr(dtemp, dx, dy, n, ix, iy, incx, incy);
  }
  else {
    dtemp = dot(dx
  }
  return dtemp;
}
double *r8mat_gen
  double *a;
  int I, j;
  a = ( double * ) malloc ( lda * n * sizeof ( double ) );
  for ( j = 1; j <= n; j++ ) {
    for ( i = 1; i <= n; i++ ) {
      a[i-1+(j-1)*lda] = r8_random ( init ) - 0.5;
    }
  }
  return a;
}
...
```

Does not require complex numerical analysis

phase I: whole program analysis



Static / Dynamic Analysis

+ scales well

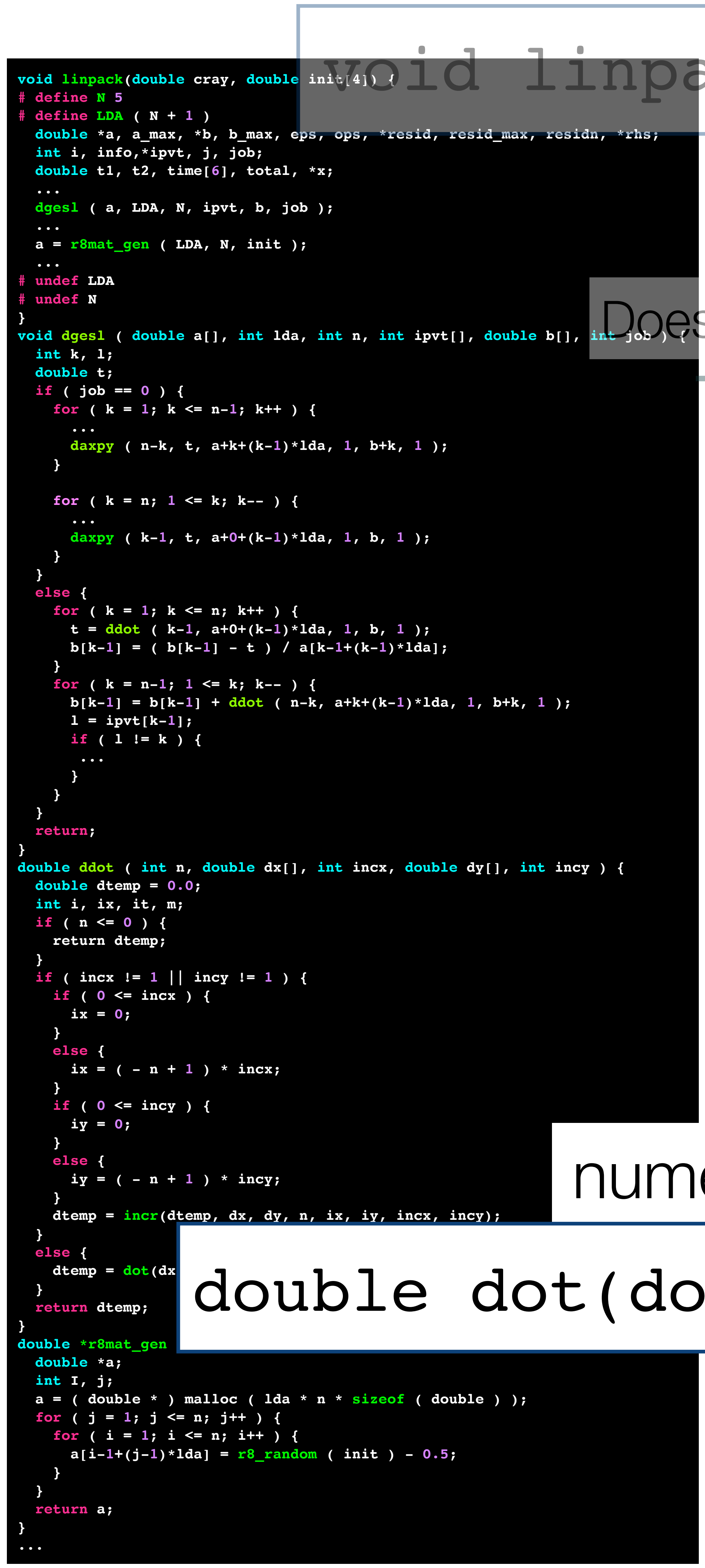
- imprecise numerical analysis

kernel input ranges

numerically interesting: numerical kernel

```
double dot(double dx[4], double dy[4])
```

(Conditional) Floating-Point Verification



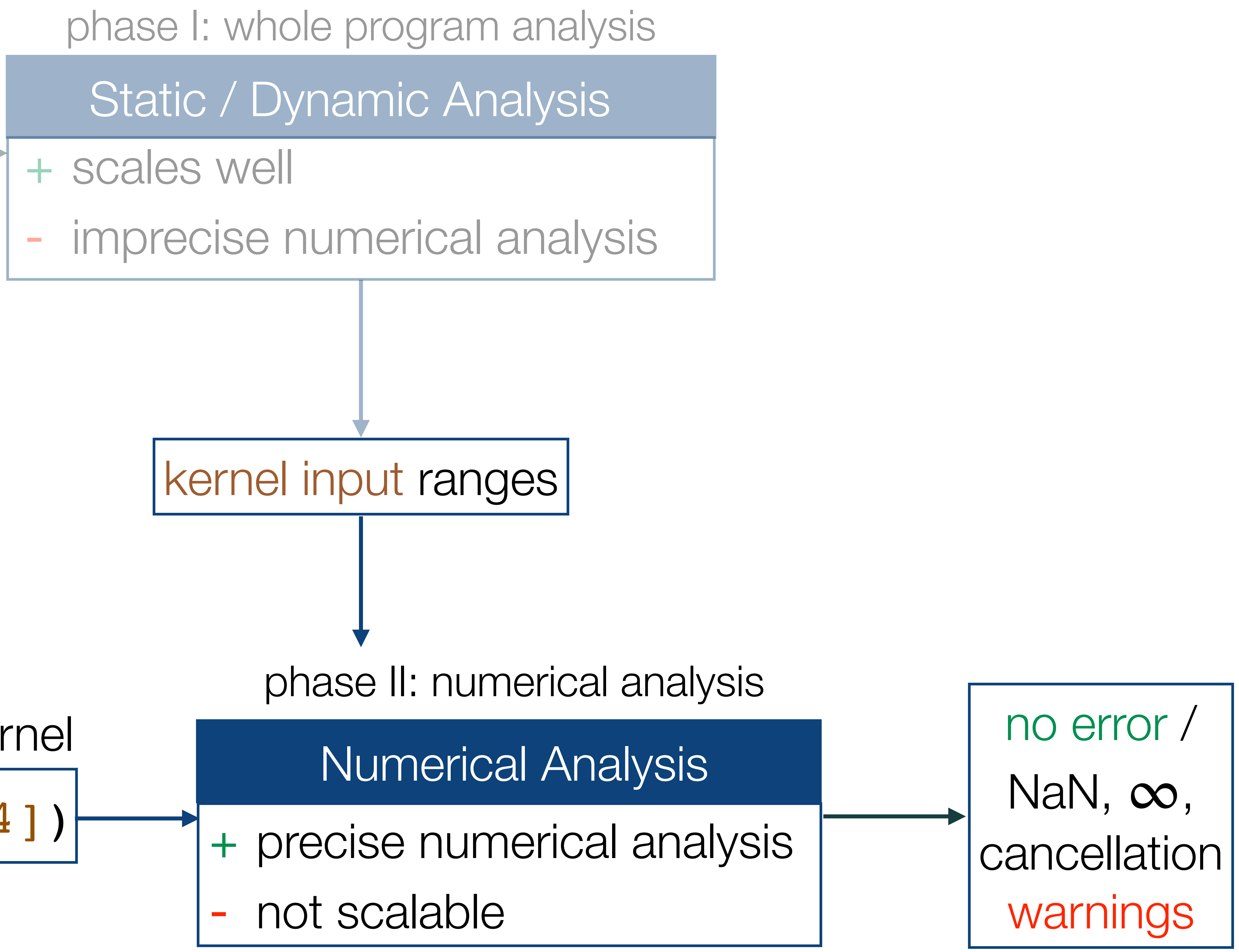
```
void linpack(double cray, double init[4]) {
# define N 5
# define LDA ( N + 1 )
  double *a, a_max, *b, b_max, eps, ops, *resid, resid_max, residn, *rhs;
  int i, info,*ipvt, j, job;
  double t1, t2, time[6], total, *x;
  ...
  dgesl ( a, LDA, N, ipvt, b, job );
  ...
  a = r8mat_gen ( LDA, N, init );
  ...
# undef LDA
# undef N
}
void dgesl ( double a[], int lda, int n, int ipvt[], double b[], int job ) {
  int k, l;
  double t;
  if ( job == 0 ) {
    for ( k = 1; k <= n-1; k++ ) {
      ...
      daxpy ( n-k, t, a+k+(k-1)*lda, 1, b+k, 1 );
    }

    for ( k = n; 1 <= k; k-- ) {
      ...
      daxpy ( k-1, t, a+0+(k-1)*lda, 1, b, 1 );
    }
  }
  else {
    for ( k = 1; k <= n; k++ ) {
      t = ddot ( k-1, a+0+(k-1)*lda, 1, b, 1 );
      b[k-1] = ( b[k-1] - t ) / a[k-1+(k-1)*lda];
    }
    for ( k = n-1; 1 <= k; k-- ) {
      b[k-1] = b[k-1] + ddot ( n-k, a+k+(k-1)*lda, 1, b+k, 1 );
      l = ipvt[k-1];
      if ( l != k ) {
       ...
      }
    }
  }
  return;
}
double ddot ( int n, double dx[], int incx, double dy[], int incy ) {
  double dtemp = 0.0;
  int i, ix, it, m;
  if ( n <= 0 ) {
    return dtemp;
  }
  if ( incx != 1 || incy != 1 ) {
    if ( 0 <= incx ) {
      ix = 0;
    }
    else {
      ix = ( - n + 1 ) * incx;
    }
    if ( 0 <= incy ) {
      iy = 0;
    }
    else {
      iy = ( - n + 1 ) * incy;
    }
    dtemp = incr(dtemp, dx, dy, n, ix, iy, incx, incy);
  }
  else {
    dtemp = dot(dx
  }
  return dtemp;
}
double *r8mat_gen
  double *a;
  int I, j;
  a = ( double * ) malloc ( lda * n * sizeof ( double ) );
  for ( j = 1; j <= n; j++ ) {
    for ( i = 1; i <= n; i++ ) {
      a[i-1+(j-1)*lda] = r8_random ( init ) - 0.5;
    }
  }
  return a;
}
...
```

```
void linpack(double cray, double init[4])
```

Does not require complex numerical analysis

numerically interesting: numerical kernel

```
double dot(double dx[4], double dy[4])
```



phase I: whole program analysis

Static / Dynamic Analysis

+ scales well

- imprecise numerical analysis

kernel input ranges

phase II: numerical analysis

Numerical Analysis

+ precise numerical analysis

- not scalable

no error / NaN, ∞, cancellation warnings

Summary: (Conditional) Floating-Point Verification

#benchmark	#kernel	lang	#in	LOC	(conditional) verification
11	24	C, C++	1-24	31-2187	14 verified, 10 warnings (2 cancellation, 8 NaN/∞)

zenodo https://doi.org/10.5281/zenodo.8043359

Summary: (Conditional) Floating-Point Verification

#benchmark	#kernel	lang	#in	LOC	(conditional) verification
11	24	C, C++	1-24	31-2187	14 verified, 10 warnings (2 cancellation, 8 NaN/∞)

zenodo https://doi.org/10.5281/zenodo.8043359

(Conditional) verification of floating-point kernels ‘hidden’ in large applications