Beyond Floating Point: Next-Generation Computer Arithmetic

John L. Gustafson

Professor, A*STAR and National University of Singapore

Why worry about floating-point?

Find the scalar product $a \cdot b$:

$$a = (3.2\text{e}7,\ 1,\ -1,\ \ 8.0\text{e}7)$$
$$b = (4.0\text{e}7,\ 1,\ -1,\ -1.6\text{e}7)$$

Note: All values are integers that can be expressed exactly in the IEEE 754 Standard floating-point format (single or double precision)

Single Precision, 32 bits: $a \cdot b = 0$

Double Precision, 64 bits: $a \cdot b = 0$

Why worry about floating-point?

Find the scalar product $a \cdot b$:

$$a = (3.2\text{e}7,\ 1,\ -1,\ \ 8.0\text{e}7)$$
$$b = (4.0\text{e}7,\ 1,\ -1,\ -1.6\text{e}7)$$

Note: All values are integers that can be expressed exactly in the IEEE 754 Standard floating-point format (single or double precision)

Single Precision, 32 bits:	$a \cdot b = 0$
Double Precision, 64 bits:	$a \cdot b = 0$
Double Precision with binary sum collapse:	$a \cdot b = 1$

Why worry about floating-point?

Find the scalar product $a \cdot b$:

$$a = (3.2e7, 1, -1, \quad 8.0e7)$$
$$b = (4.0e7, 1, -1, -1.6e7)$$

Note: All values are integers that can be expressed exactly in the IEEE 754 Standard floating-point format (single or double precision)

Single Precision, 32 bits:	$a \cdot b = 0$
Double Precision, 64 bits:	$a \cdot b = 0$
Double Precision with binary sum collapse:	$a \cdot b = 1$
Correct answer:	$a \cdot b = 2$

Most linear algebra is unstable with floats!

What's wrong with IEEE 754? (1)

- It's a *guideline*, not a *standard*
- **No guarantee of identical results across systems**
- Invisible rounding errors; the "inexact" flag is useless
- Breaks algebra laws, like $a+(b+c) = (a+b)+c$
- Overflows to infinity, underflows to zero
- No way to express most of the real number line

A Key Idea: The Ubit

We have *always* had a way of expressing infinite-decimal reals correctly with a finite set of symbols.

Incorrect: $\pi = 3.14$

Correct: $\pi = 3.14\cdots$

The latter means $3.14 < \pi < 3.15$, a **true statement**.

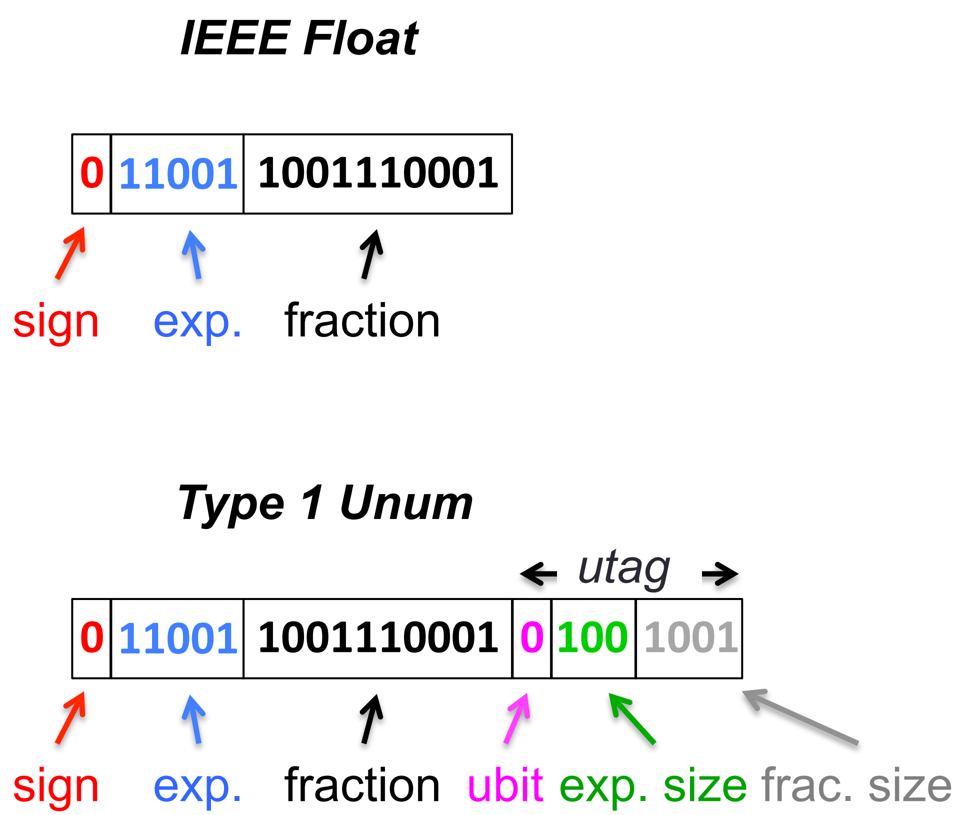
Presence or absence of the "⋯" is the *ubit*, just like a sign bit. It is 0 if exact, 1 if there are more bits after the last fraction bit, not all 0s and not all 1s.

What’s wrong with IEEE 754? (2)

- Exponents usually too large; not adjustable
- Accuracy is flat across a vast range, then falls off a cliff
- Wasted bit patterns; “negative zero,” too many NaN values
- Subnormal numbers are headache
- Divides are hard
- Decimal floats are expensive; no 32-bit version

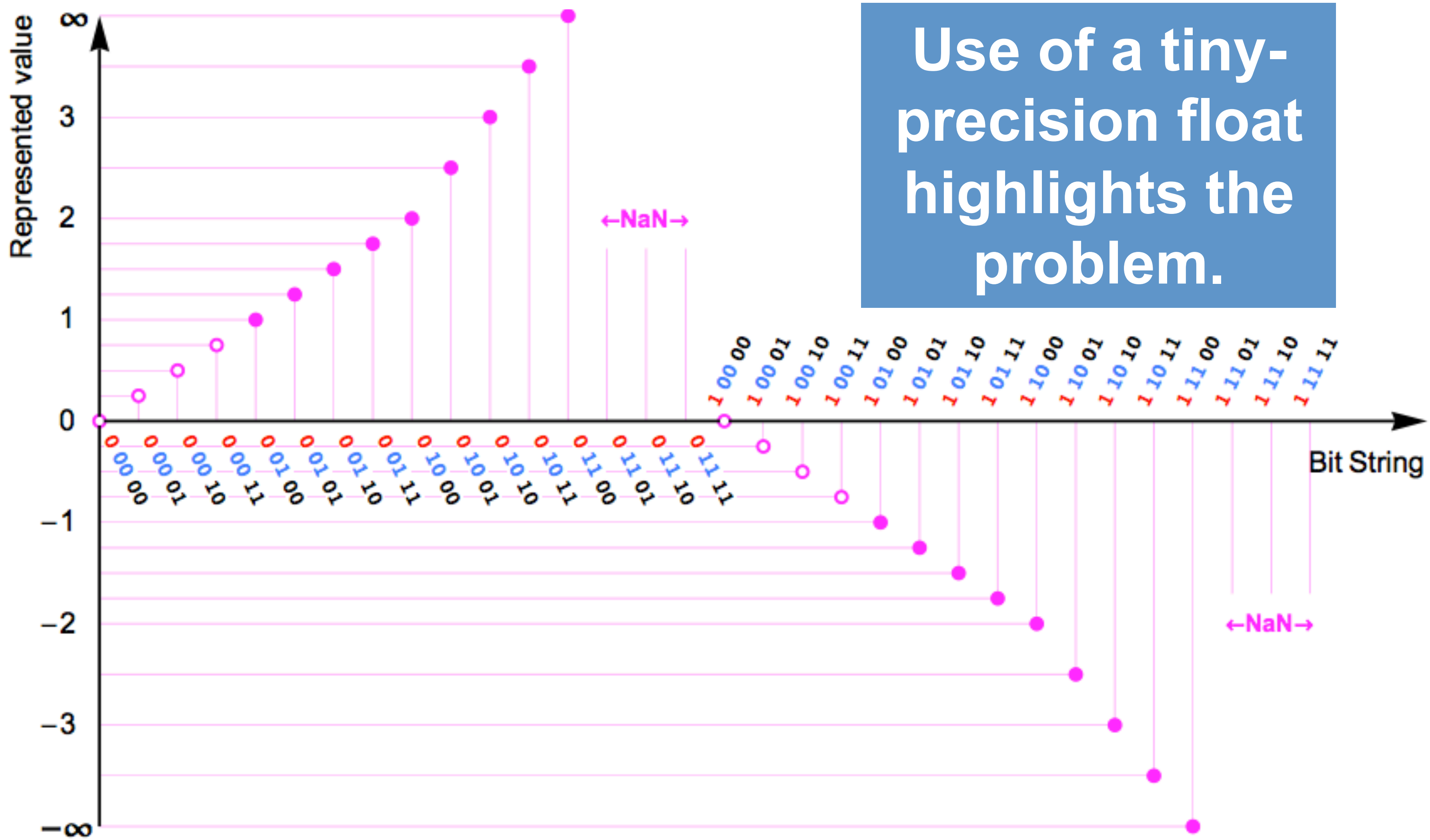
Quick Introduction to Unum (universal number) Format: Type 1

- Type 1 unums extend IEEE floating point with three metadata fields for exactness, exponent size, and fraction size. Upward compatible.
- Fixed size if "unpacked" to maximum size, but can vary in size to save storage, bandwidth.



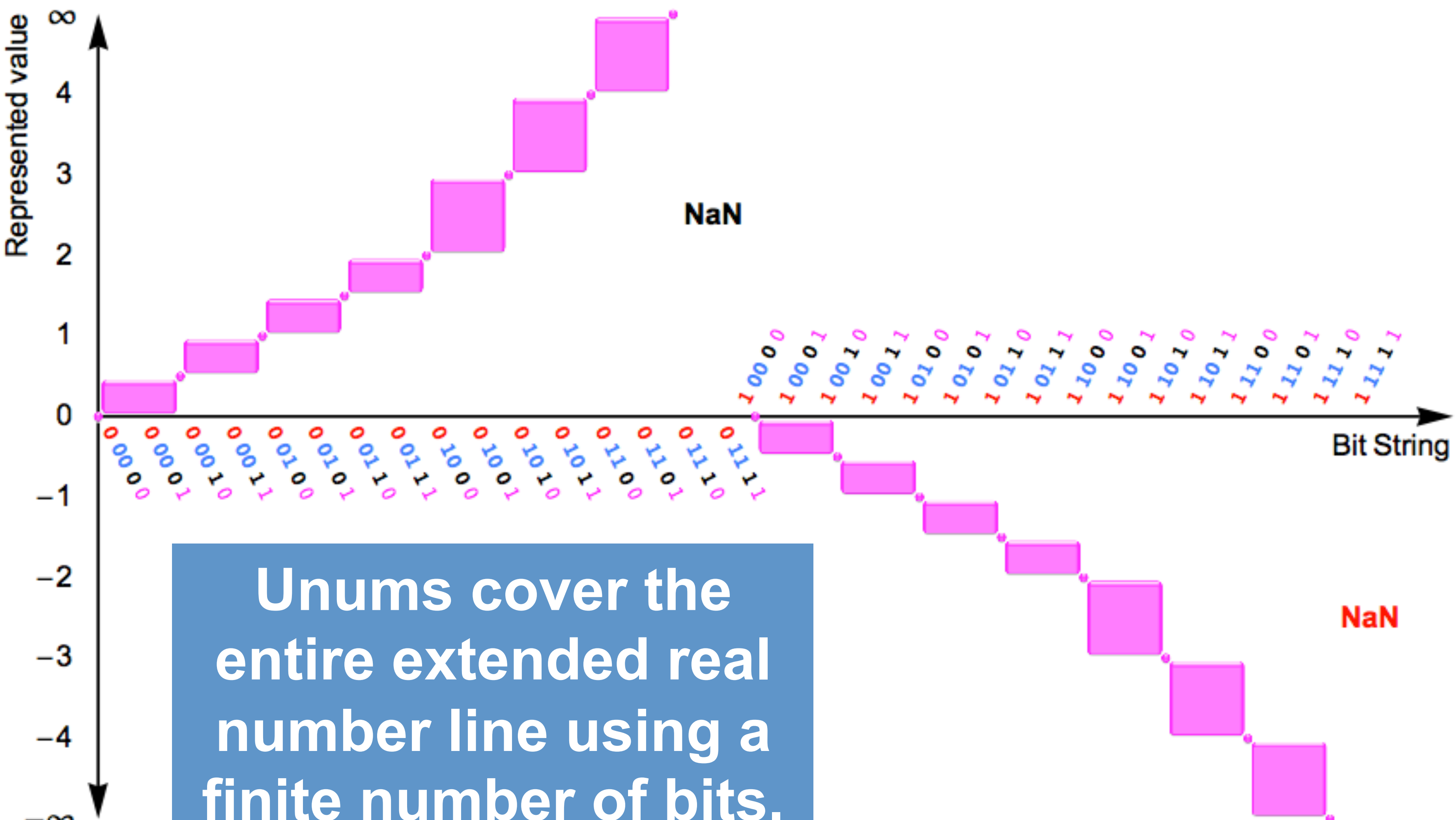
For details see *The End of Error: Unum Arithmetic*, CRC Press, 2015

Floats only express discrete points on the real number line



Use of a tiny-precision float highlights the problem.

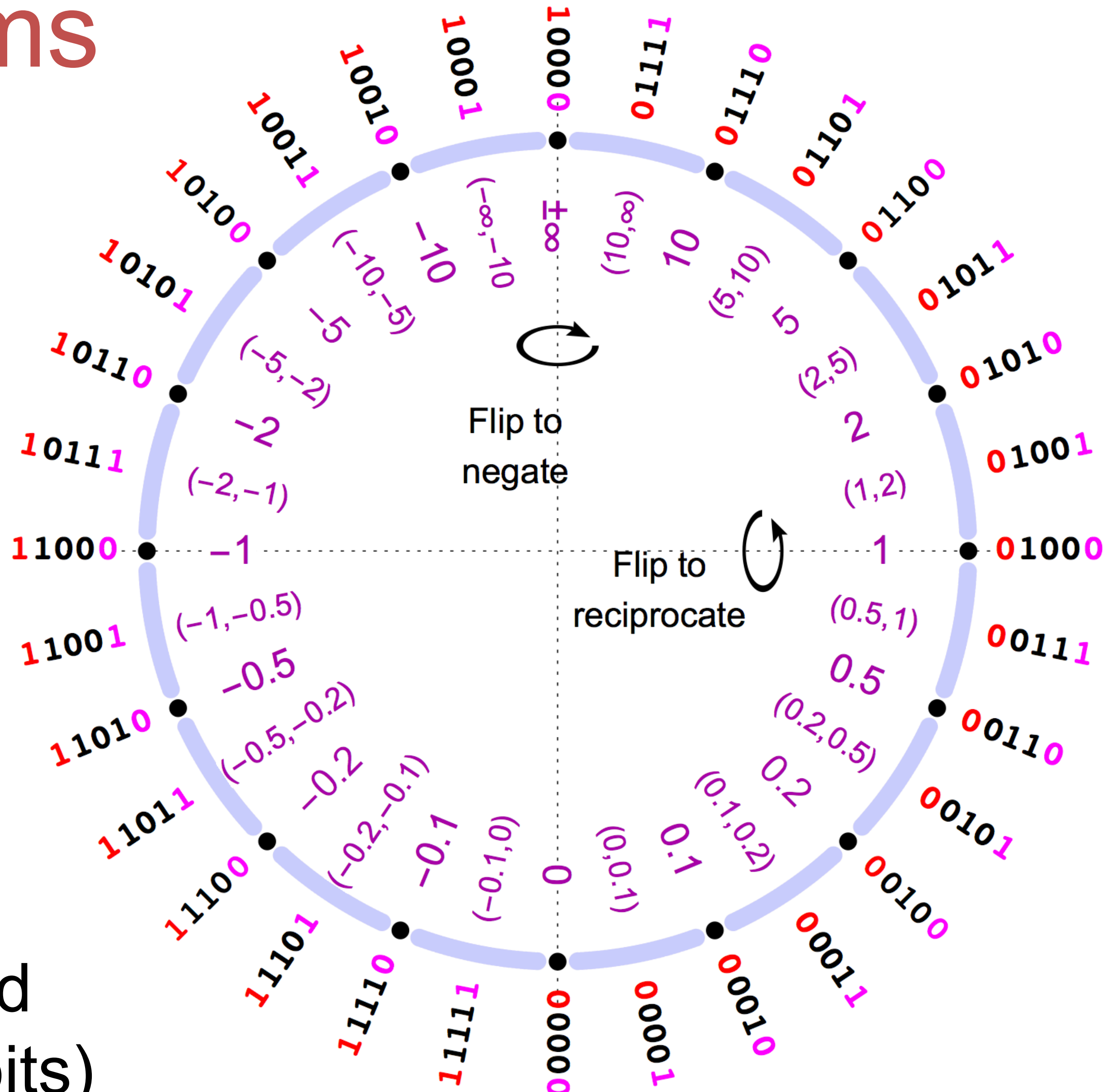
The ubit can represent exact values or the range *between exacts*



Unums cover the entire extended real number line using a finite number of bits.

Type 2 unums

- *Projective* reals
- Custom lattice
- No penalty for decimal
- Table look-up
- Perfect reciprocals
- No redundancy
- Incredibly fast (ROM) but limited precision (< 20 bits)



For details see http://superfri.org/superfri/article/view/94/78

Contrasting Calculation "Esthetics"

	Rounded: cheap, uncertain, but "good enough"	**Rigorous: certain, more work, mathematical**
IEEE Standard (1985)	Floats, $f = n \times 2^m$ m, n are integers	Intervals $[f_1, f_2]$, all x such that $f_1 \leq x \leq f_2$
Type 1 Unums (2013)	"Guess" mode, flexible precision	Unums, ubounds, sets of uboxes
Type 2 Unums (2016)	"Guess" mode, fixed precision	Sets of Real Numbers (SORNs)
Sigmoid Unums (2017)	***Posits***	***Valids***

If you mix the two esthetics, you wind up satisfying *neither*.

posit | ˈpäzət |

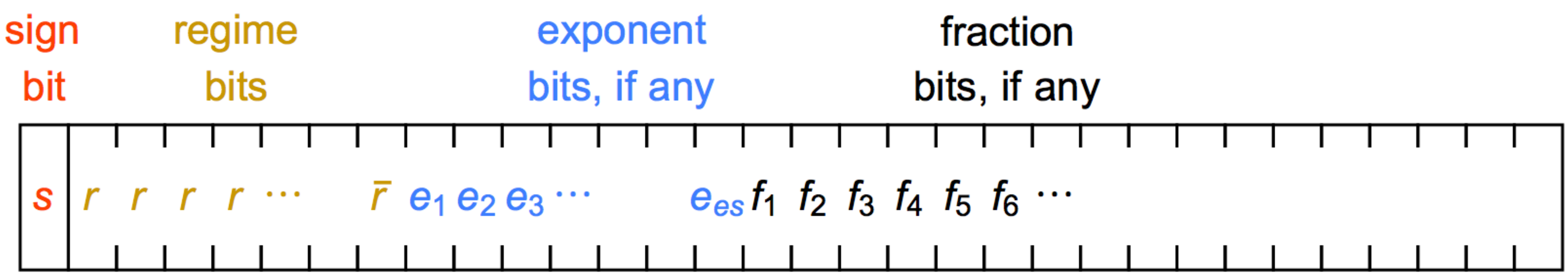
noun *Philosophy*

a statement that is made on the assumption that it will prove to be true.

Metrics for Number Systems

- Accuracy $-\log_{10}(\log_{10}(x_j / x_{j+1}))$
- Dynamic range $\log_{10}(\mathit{maxreal} / \mathit{minreal})$
- Percentage of operations that are exact (closure under + – × ÷ √ etc.)
- Average accuracy loss when they aren't
- Entropy per bit (maximize information)
- Accuracy benchmarks: simple formulas, linear equation solving, math library kernels…

Posit Arithmetic:
Beating floats at their own game



Fixed size, *nbits*.
No ubit.
Rounds after every operation.
es = exponent size = 0, 1, 2,… bits.

Posit Arithmetic Example

sign bit	regime bits				exponent bits			fraction bits							
0	0	0	0	1	1	0	1	1	1	0	1	1	1	0	1

$$+ \quad 256^{-3} \times 2^{5} \times \left(1 + 221/256\right) = 3.55\cdots\times 10^{-6}$$

Here, *es* = 3. Float-like circuitry is all that is needed (integer add, integer multiply, shifts to scale by 2^k)

Posits **do not underflow or overflow**. There is no NaN.

Simpler, smaller, faster circuits than IEEE 754

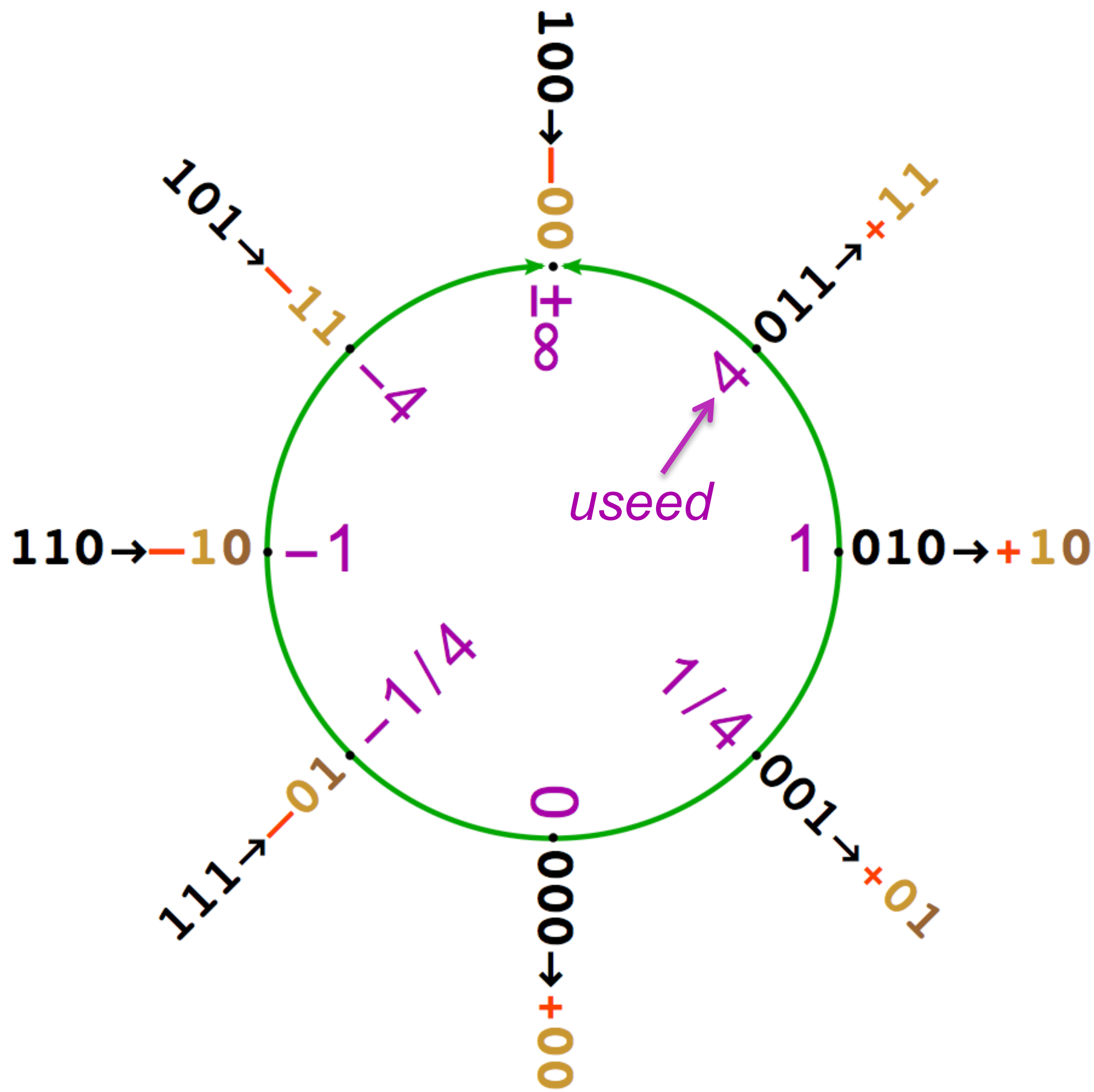
Mapping to the Projective Reals

Example with *nbits* = 3, *es* = 1.

Value at 45° is always

$$useed = 2^{2^{es}}$$

If bit string < 0, set sign to – and negate integer.

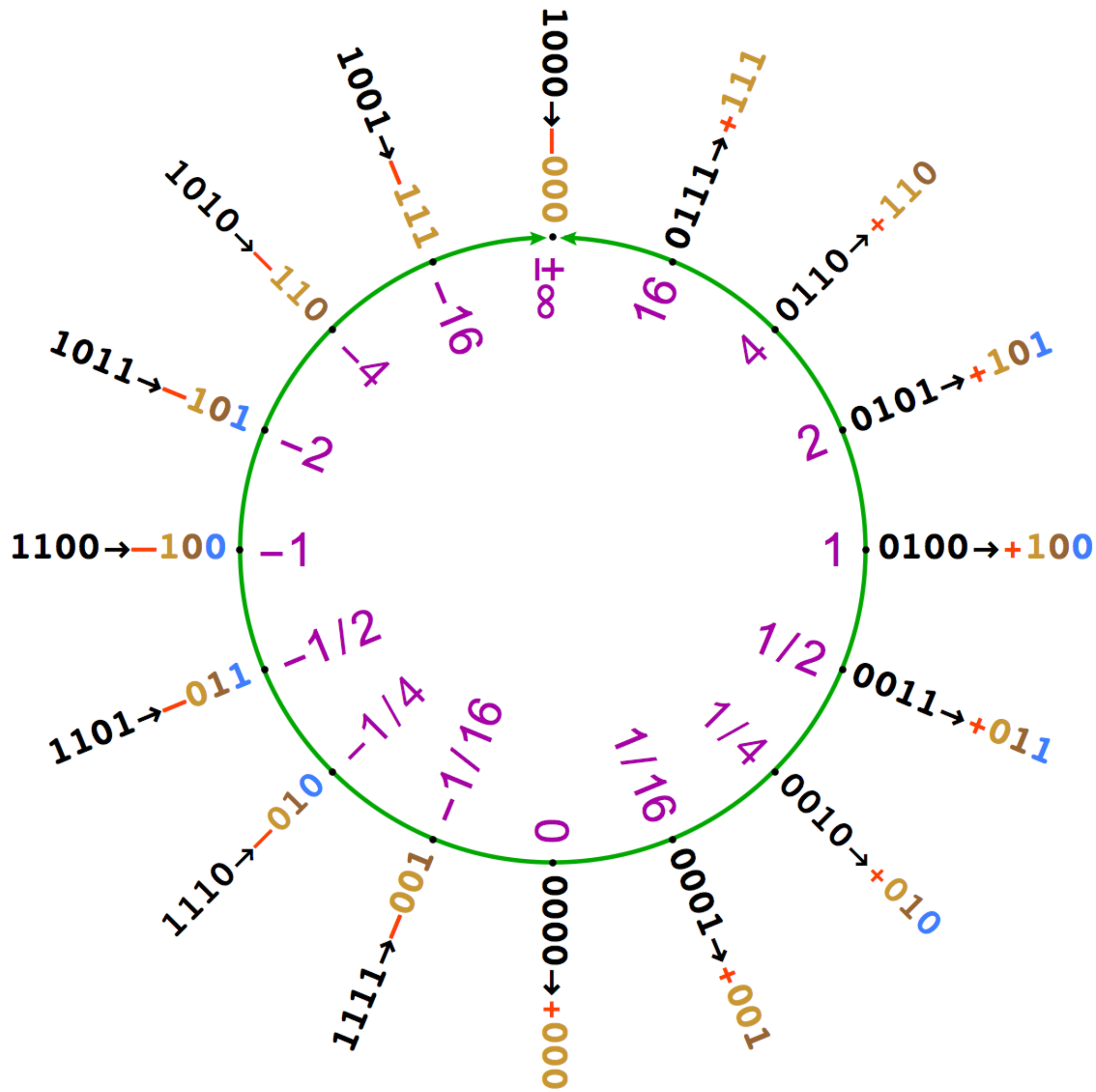


Rules for inserting new points

Between ±*maxpos* and ±∞, scale up by *useed*.
(New **regime** bit)

Between 0 and ±*minpos*, scale down by *useed.*
(New **regime** bit)

Between 2^m and 2^n where $n - m > 2$, insert $2^{(m+n)/2}$.
(New **exponent** bit)

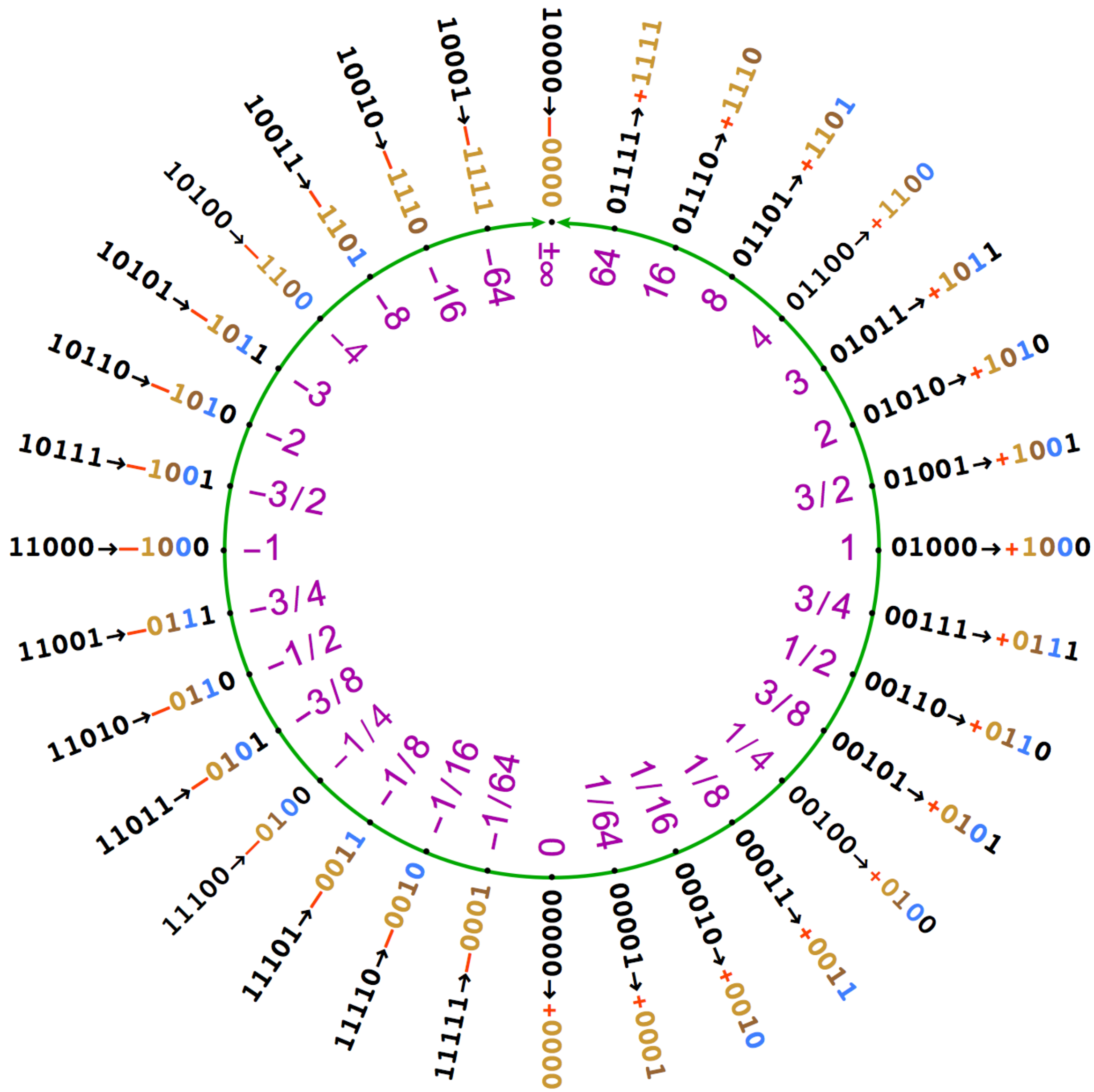


At *nbits* = 5, fraction bits appear.

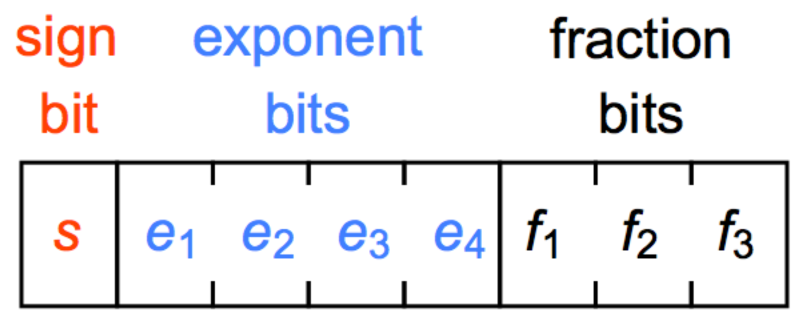
Between x and y where $y \leq 2x$, insert $(x + y)/2$.

Notice existing values stay in place.

Appending bits increases ***accuracy*** east and west, ***dynamic range*** north and south!



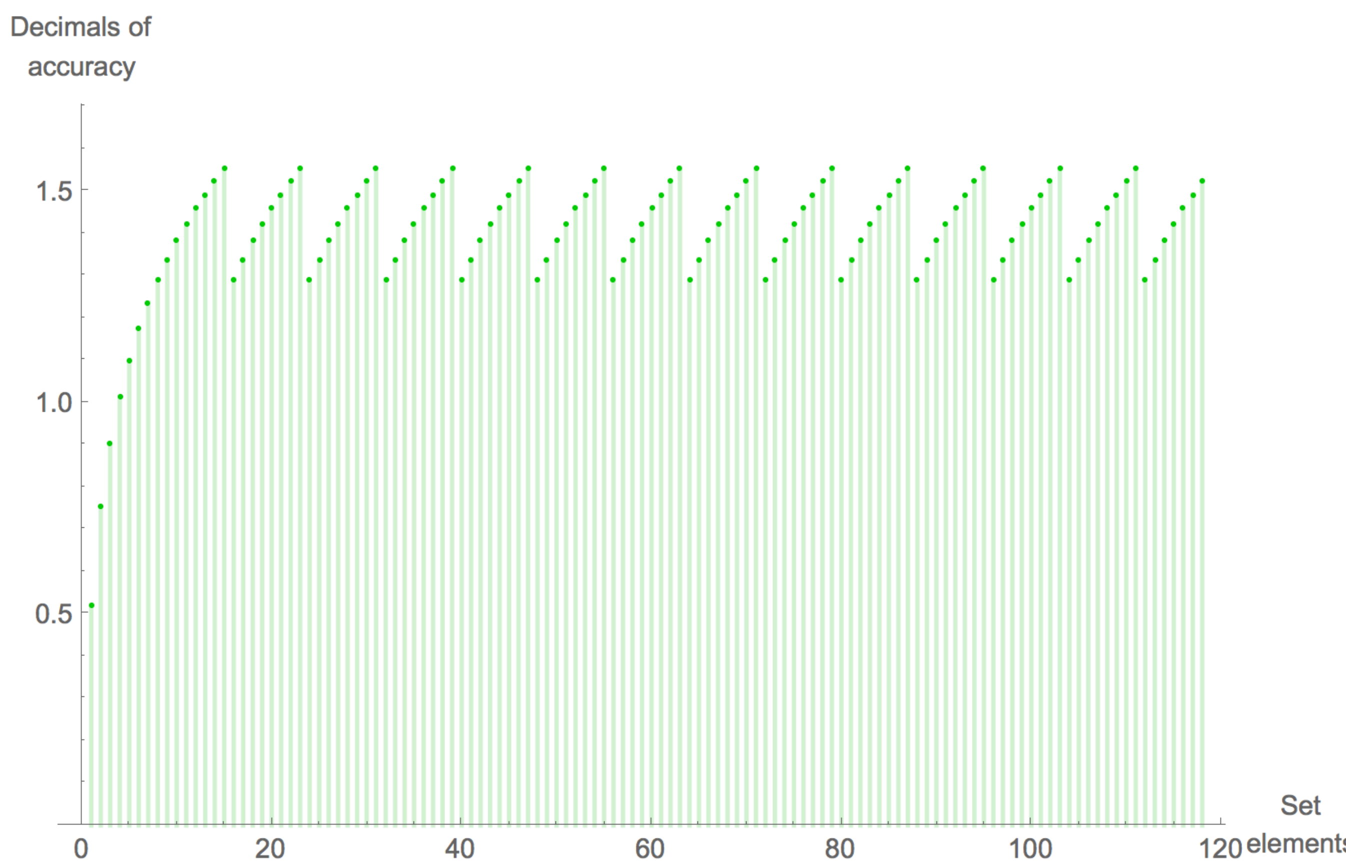
Posits vs. Floats: a metrics-based study



- Use *quarter-precision* IEEE-style floats
- Sign bit, 4 exponent bits, 3 fraction bits
- *smallsubnormal* = 1/512; *maxfloat* = 240.
- Dynamic range of five orders of magnitude
- Two representations of zero
- Fourteen representations of “Not a Number” (NaN)

Float accuracy tapers only on left

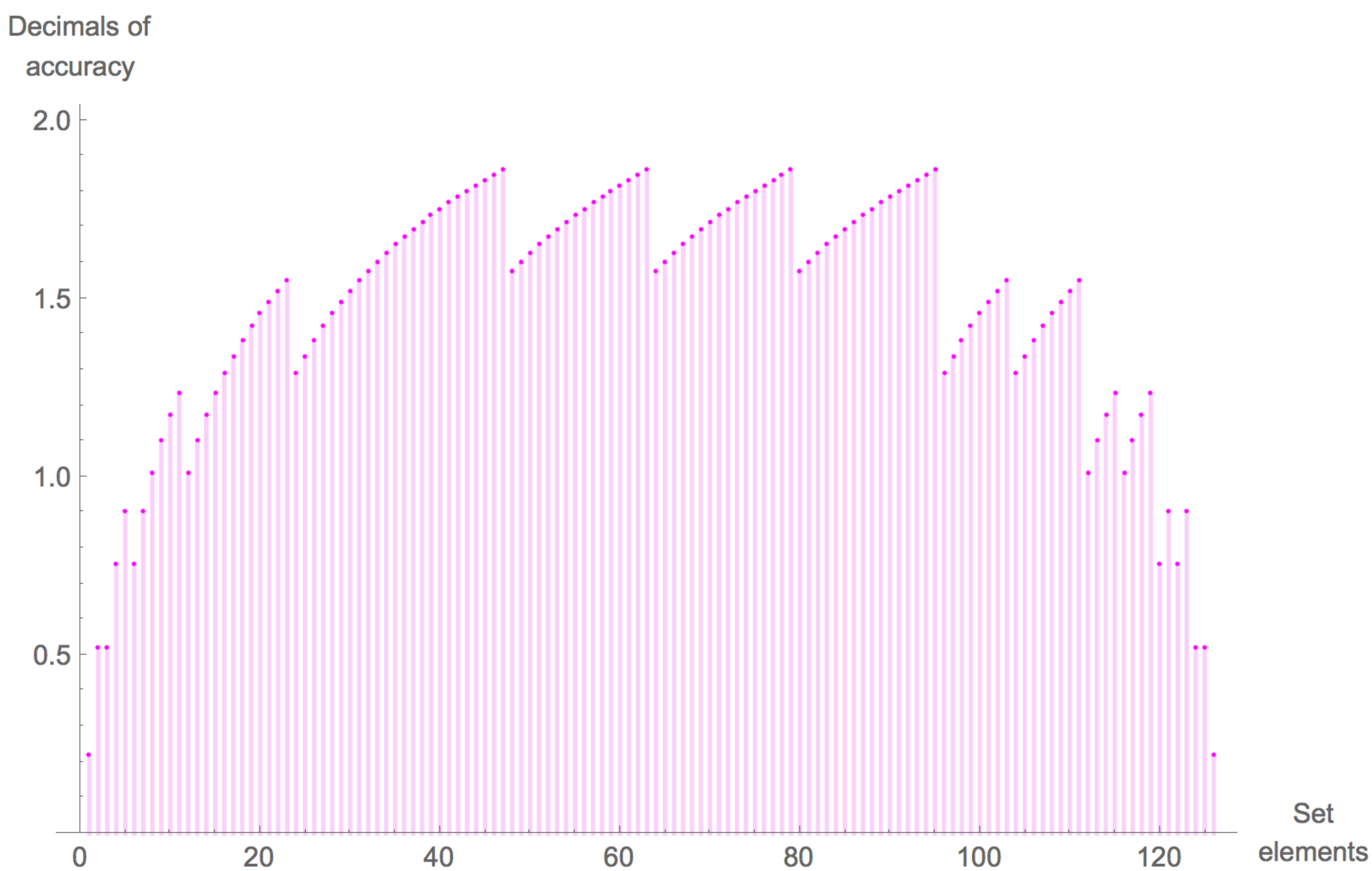
- Min: **0.52** decimals
- Avg: **1.40** decimals
- Max: **1.55** decimals



Graph shows decimals of accuracy from *smallsubnormal* to *maxfloat*.

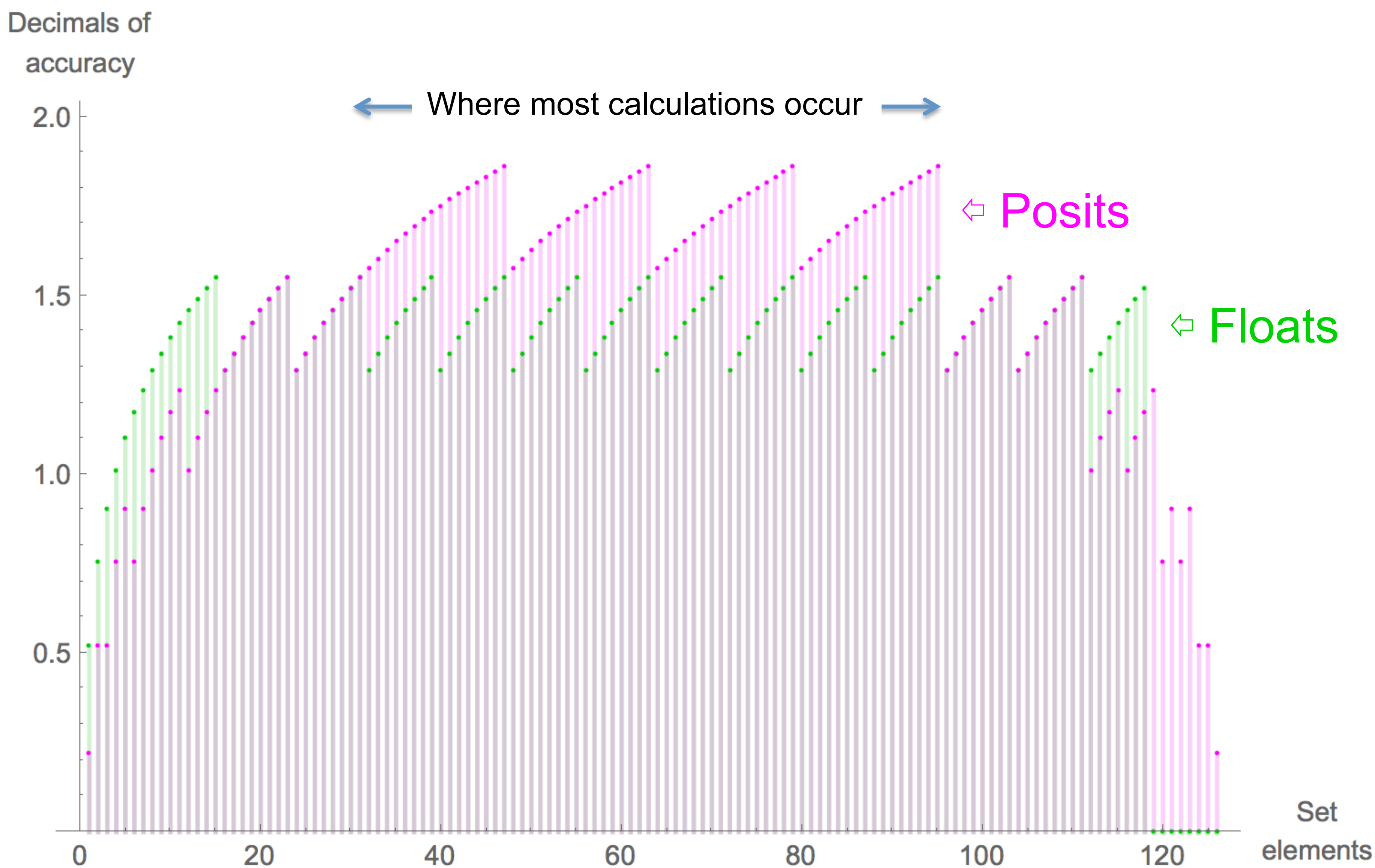
Posit accuracy tapers on both sides

- Min: **0.22** decimals
- Avg: **1.46** decimals
- Max: **1.86** decimals



Graph shows decimals of accuracy from *minpos* to *maxpos*.
But posits cover *seven* orders of magnitude, not five.

Both graphs at once

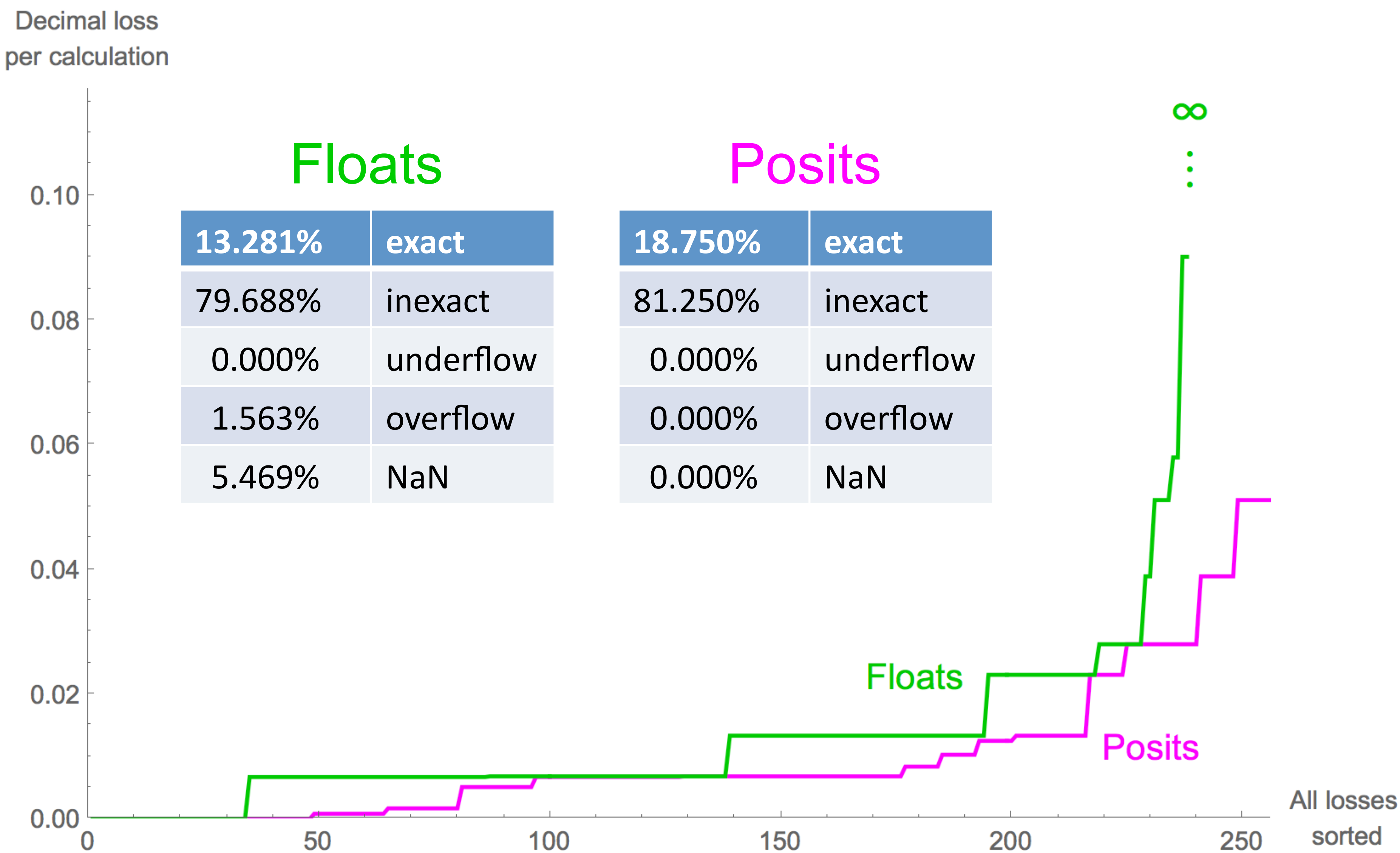


ROUND 1

Unary Operations

$1/x$, $\sqrt{x}$, x^2, $\log_2(x)$, 2^x

Closure under Reciprocation, 1/*x*



Floats

13.281%	exact
79.688%	inexact
0.000%	underflow
1.563%	overflow
5.469%	NaN

Posits

18.750%	exact
81.250%	inexact
0.000%	underflow
0.000%	overflow
0.000%	NaN

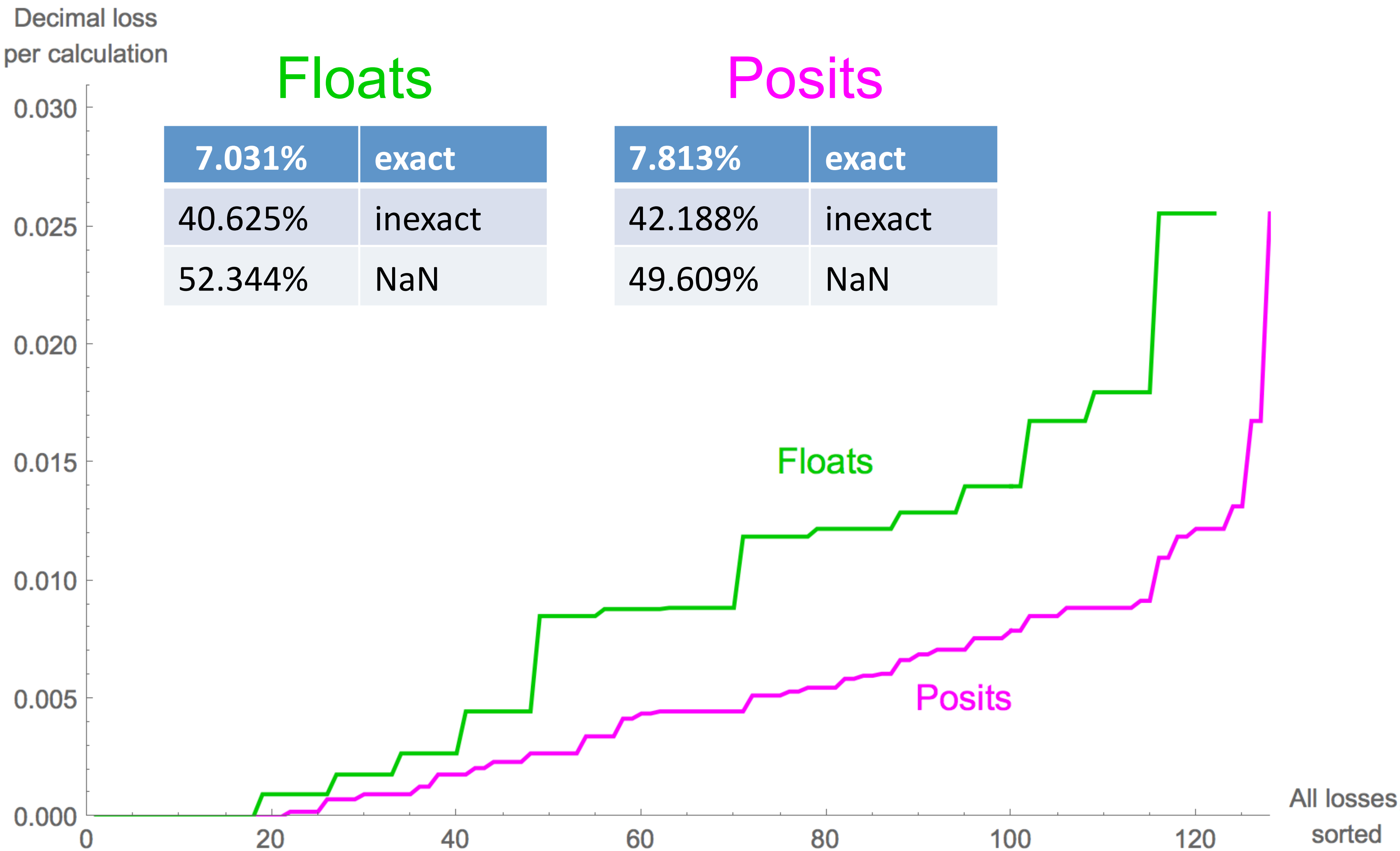
Closure under Square Root, $\sqrt{x}$

Floats

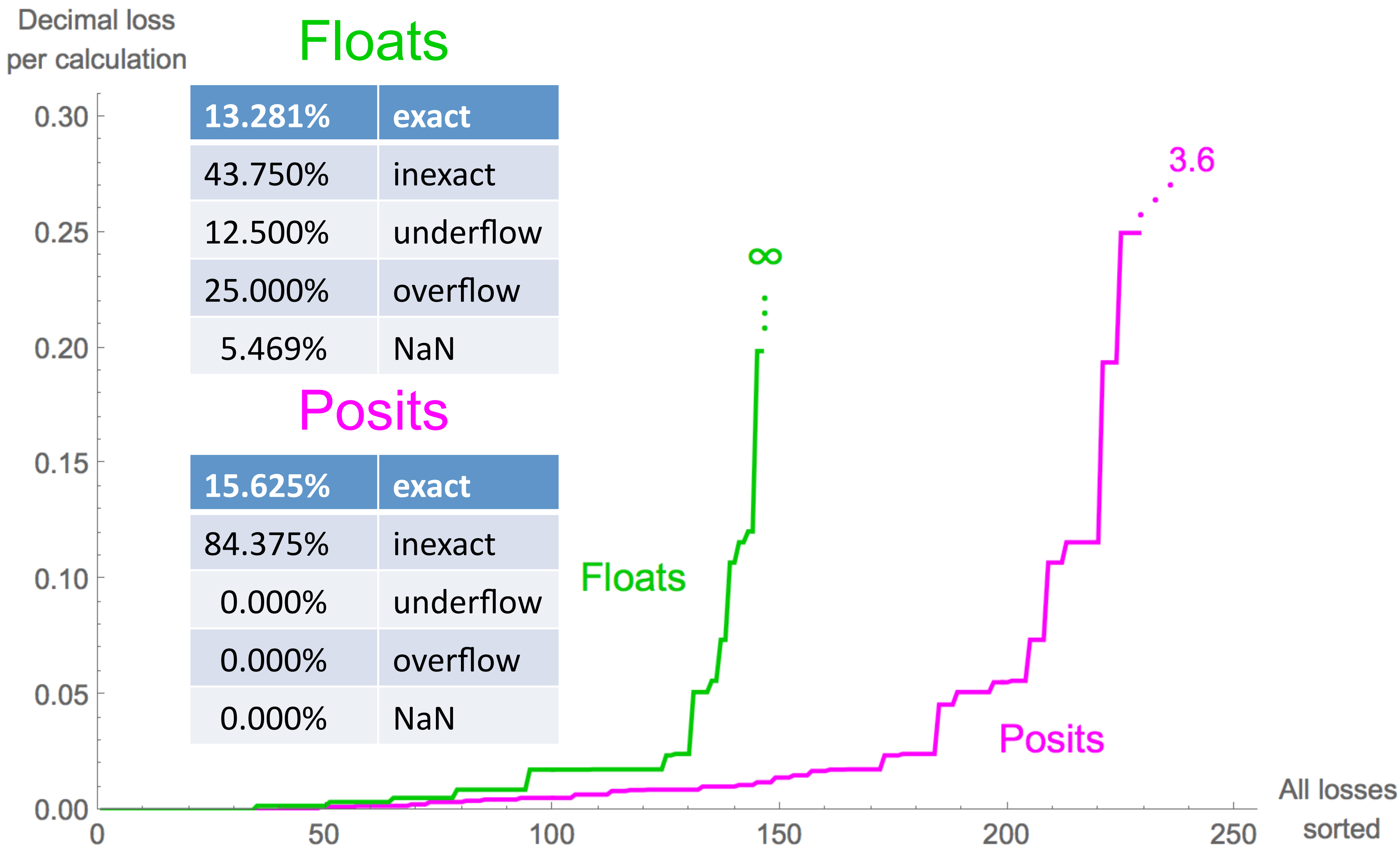
7.031%	exact
40.625%	inexact
52.344%	NaN

Posits

7.813%	exact
42.188%	inexact
49.609%	NaN



Closure under Squaring, x^2



Floats

13.281%	exact
43.750%	inexact
12.500%	underflow
25.000%	overflow
5.469%	NaN

Posits

15.625%	exact
84.375%	inexact
0.000%	underflow
0.000%	overflow
0.000%	NaN

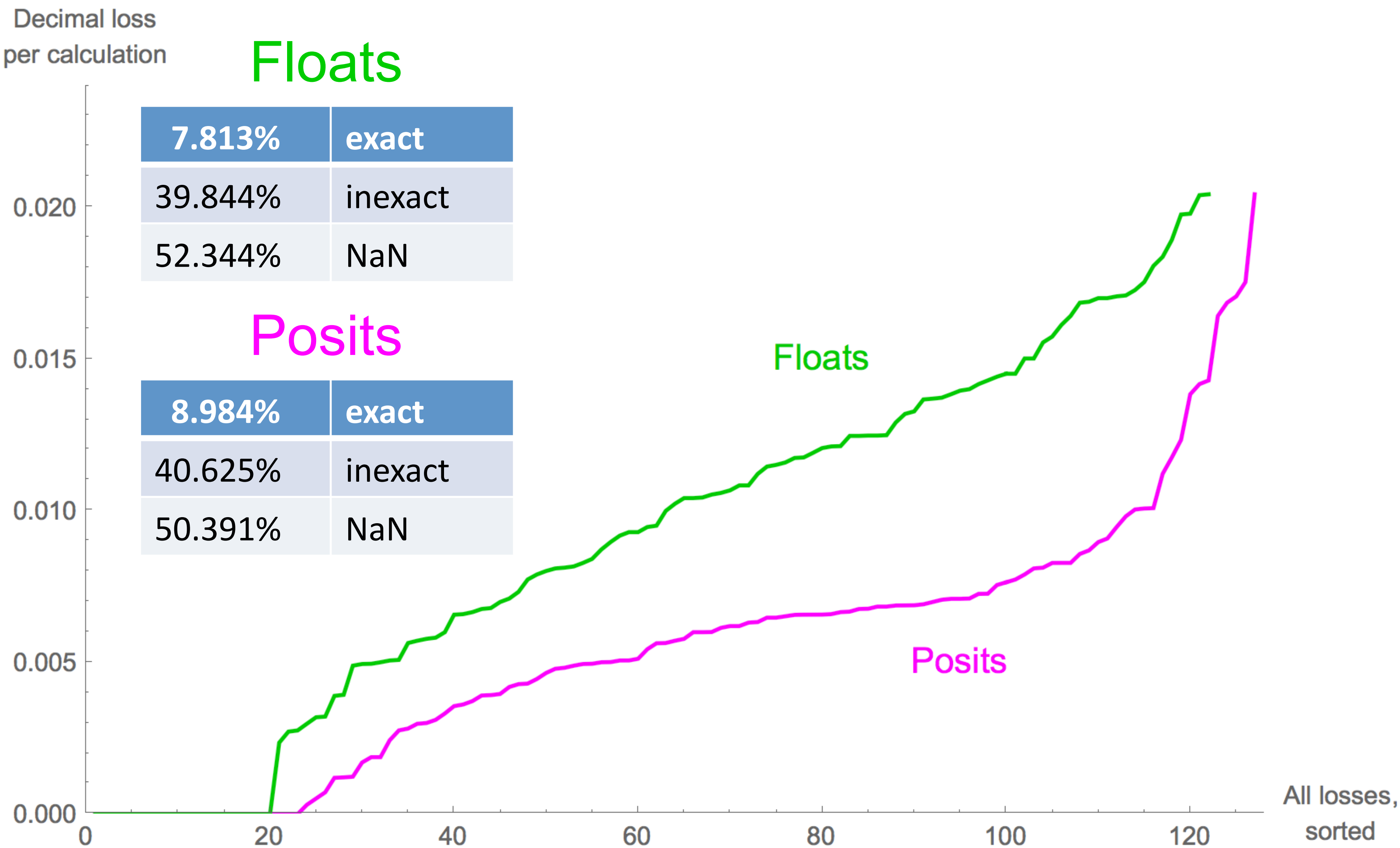
Closure under $\log_2(x)$

Floats

7.813%	exact
39.844%	inexact
52.344%	NaN

Posits

8.984%	exact
40.625%	inexact
50.391%	NaN



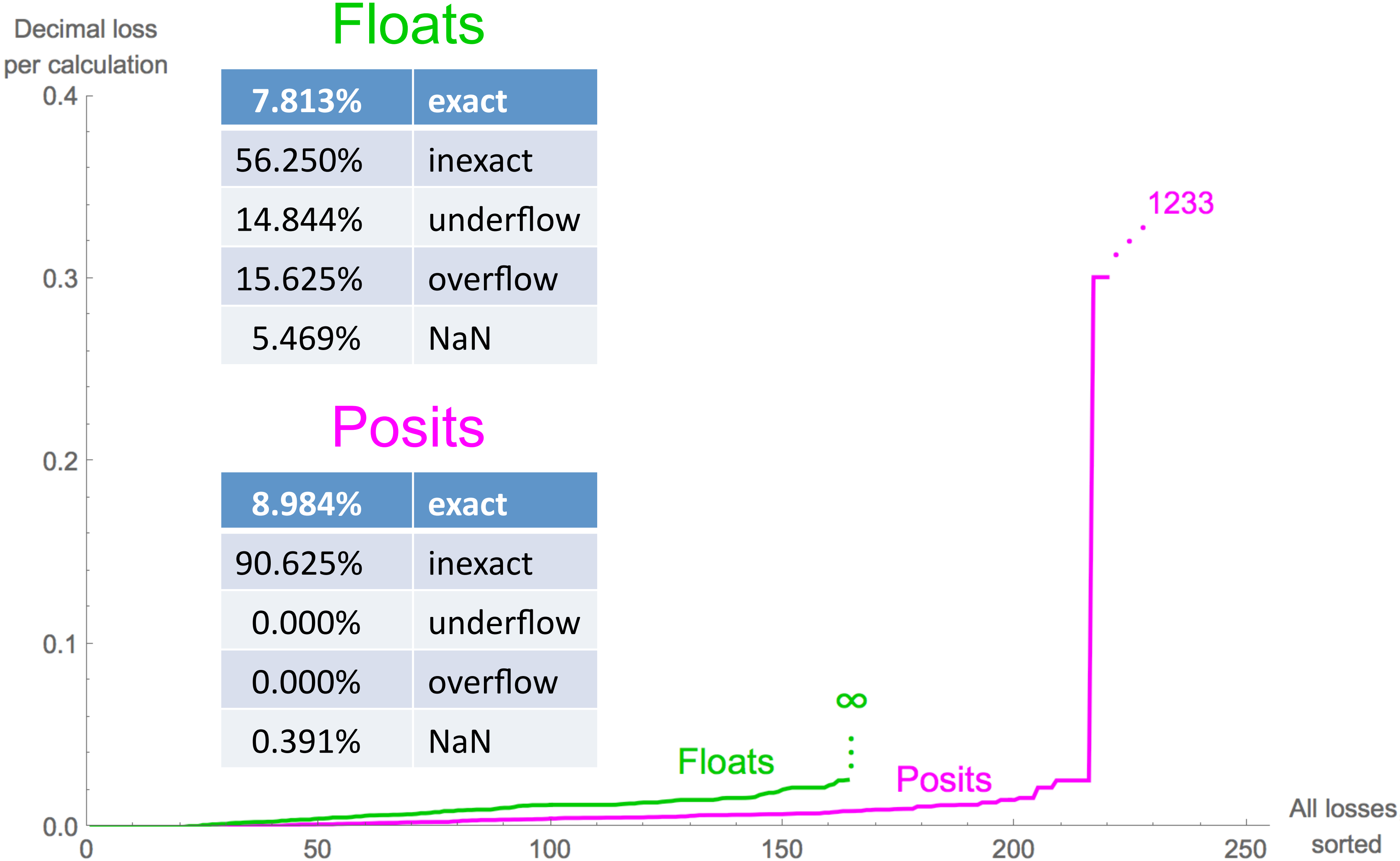
Closure under 2^x

Floats

7.813%	exact
56.250%	inexact
14.844%	underflow
15.625%	overflow
5.469%	NaN

Posits

8.984%	exact
90.625%	inexact
0.000%	underflow
0.000%	overflow
0.391%	NaN



ROUND 2

Two-Argument Operations

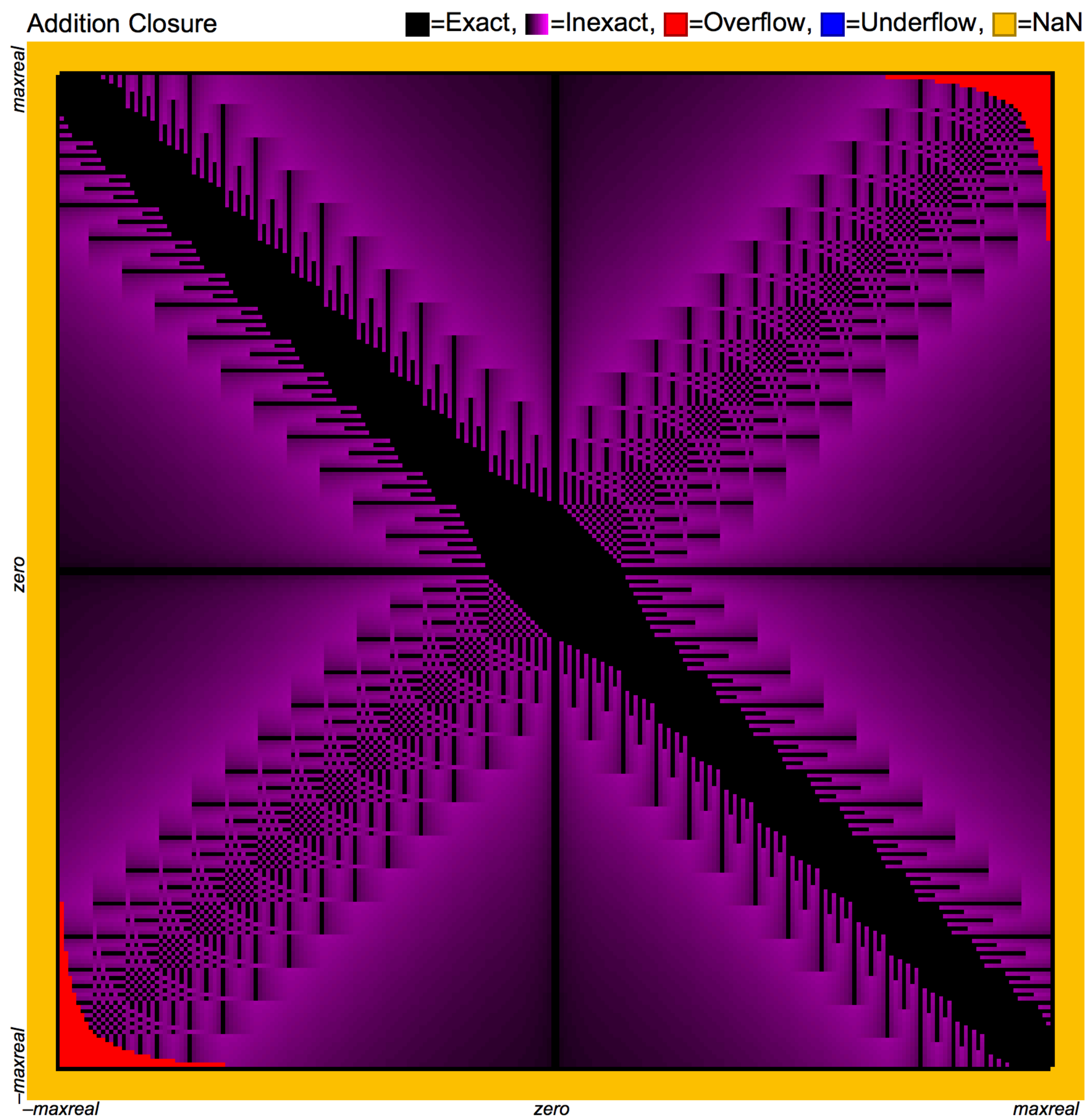
$x + y$, $x \times y$, $x \div y$

Addition Closure Plot: Floats

18.533%	exact
70.190%	inexact
0.000%	underflow
0.635%	overflow
10.641%	NaN

Inexact results are magenta; the larger the error, the brighter the color.

Addition can overflow, but cannot underflow.



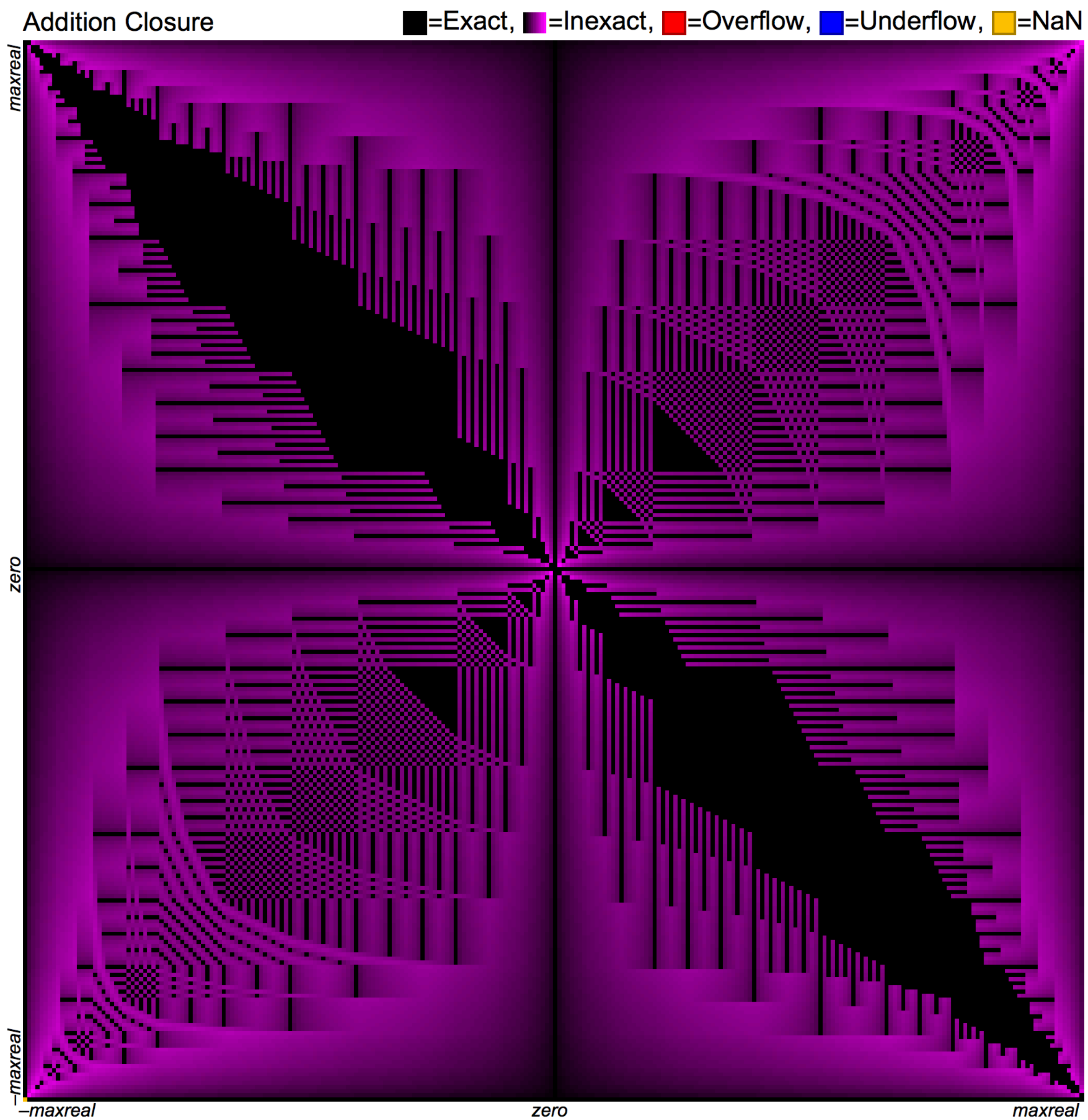
Addition Closure Plot: Posits

25.005%	**exact**
74.994%	inexact
0.000%	underflow
0.000%	overflow
0.002%	NaN

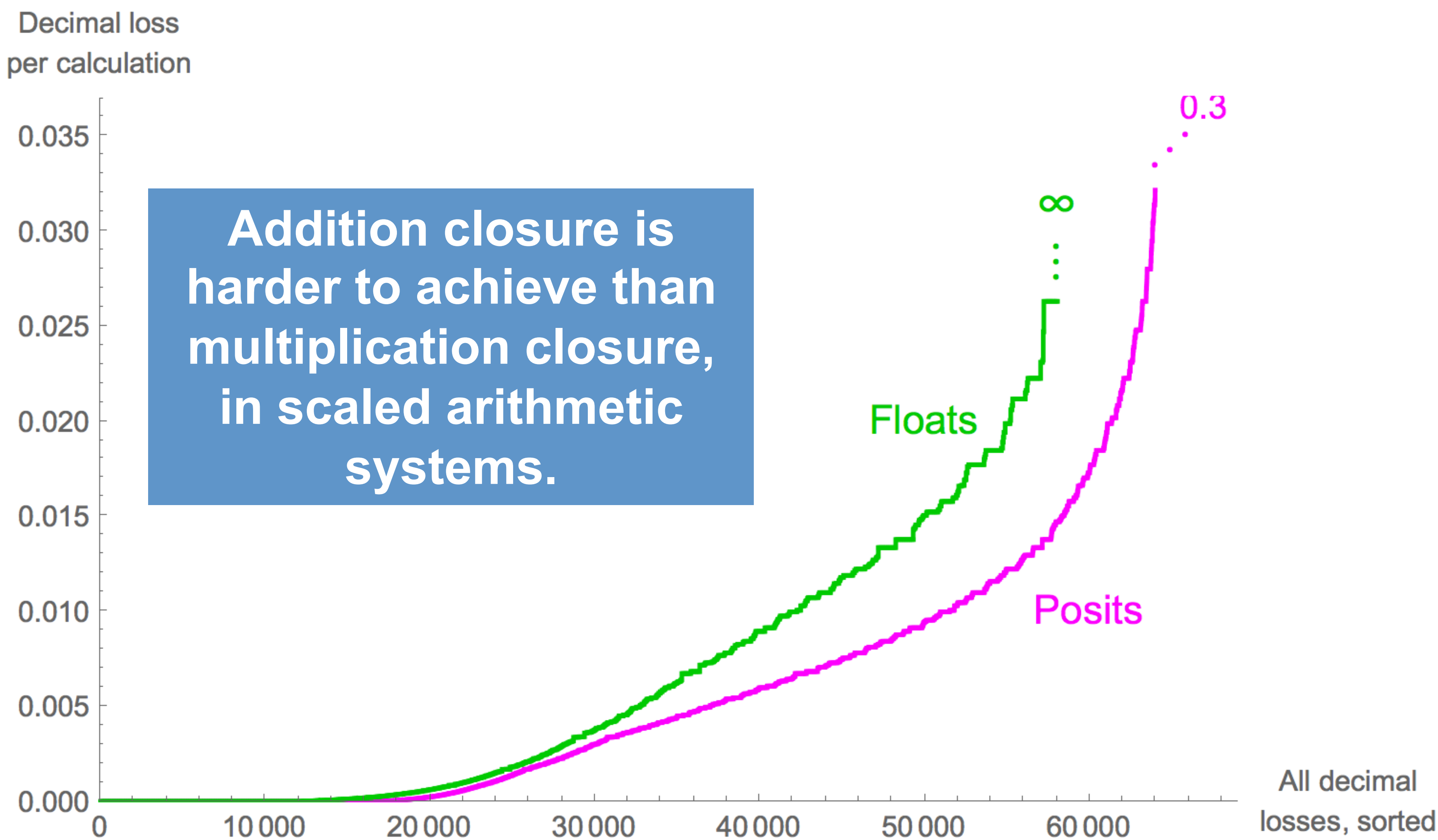
Only one case is a NaN:

$$\pm\infty + \pm\infty$$

With posits, a NaN stops the calculation.



All decimal losses, sorted

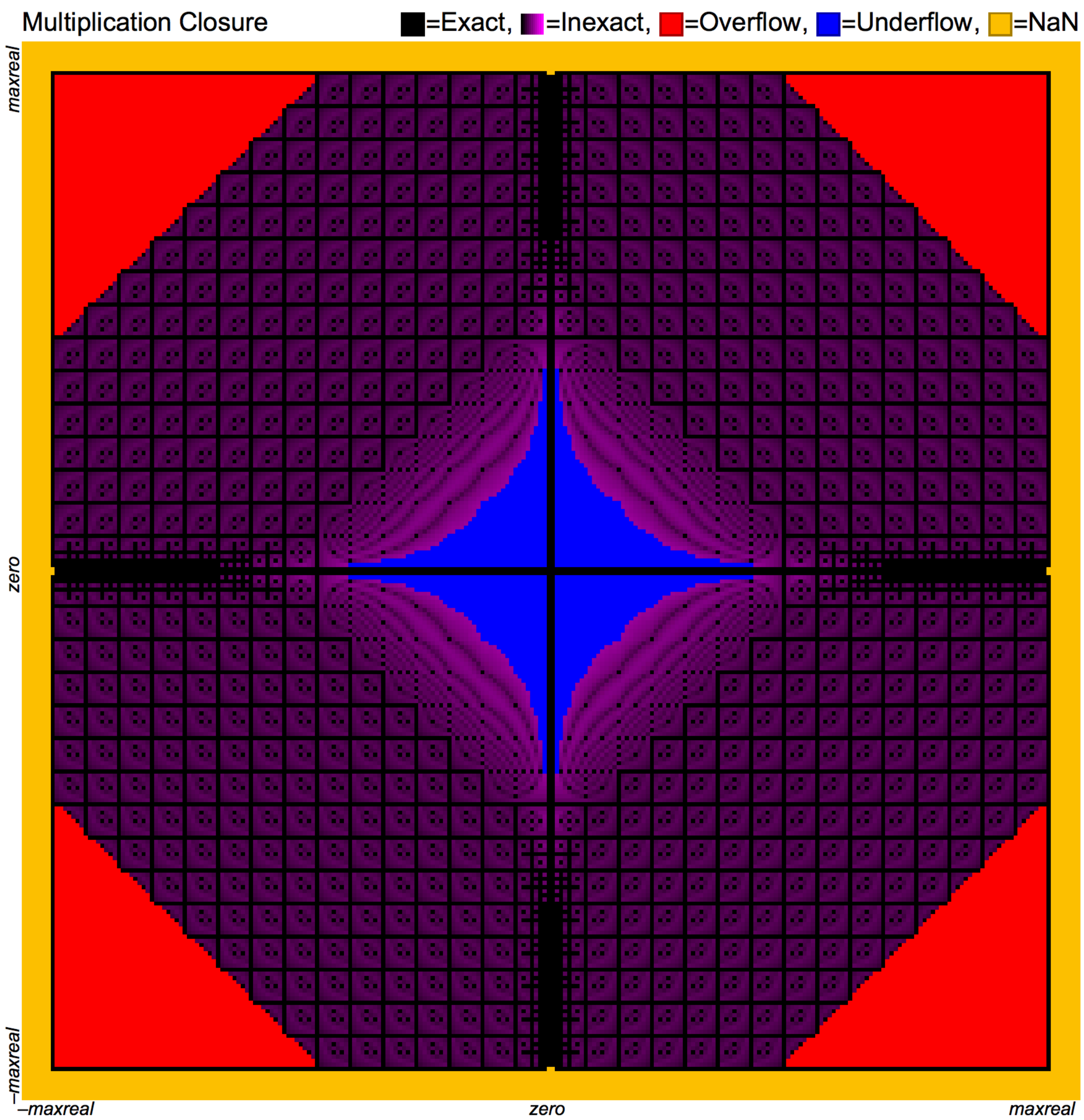


Multiplication Closure Plot: Floats

22.272%	exact
58.279%	inexact
2.475%	underflow
6.323%	overflow
10.651%	NaN

Floats score their first win: more exact products than posits…

but at a terrible cost!

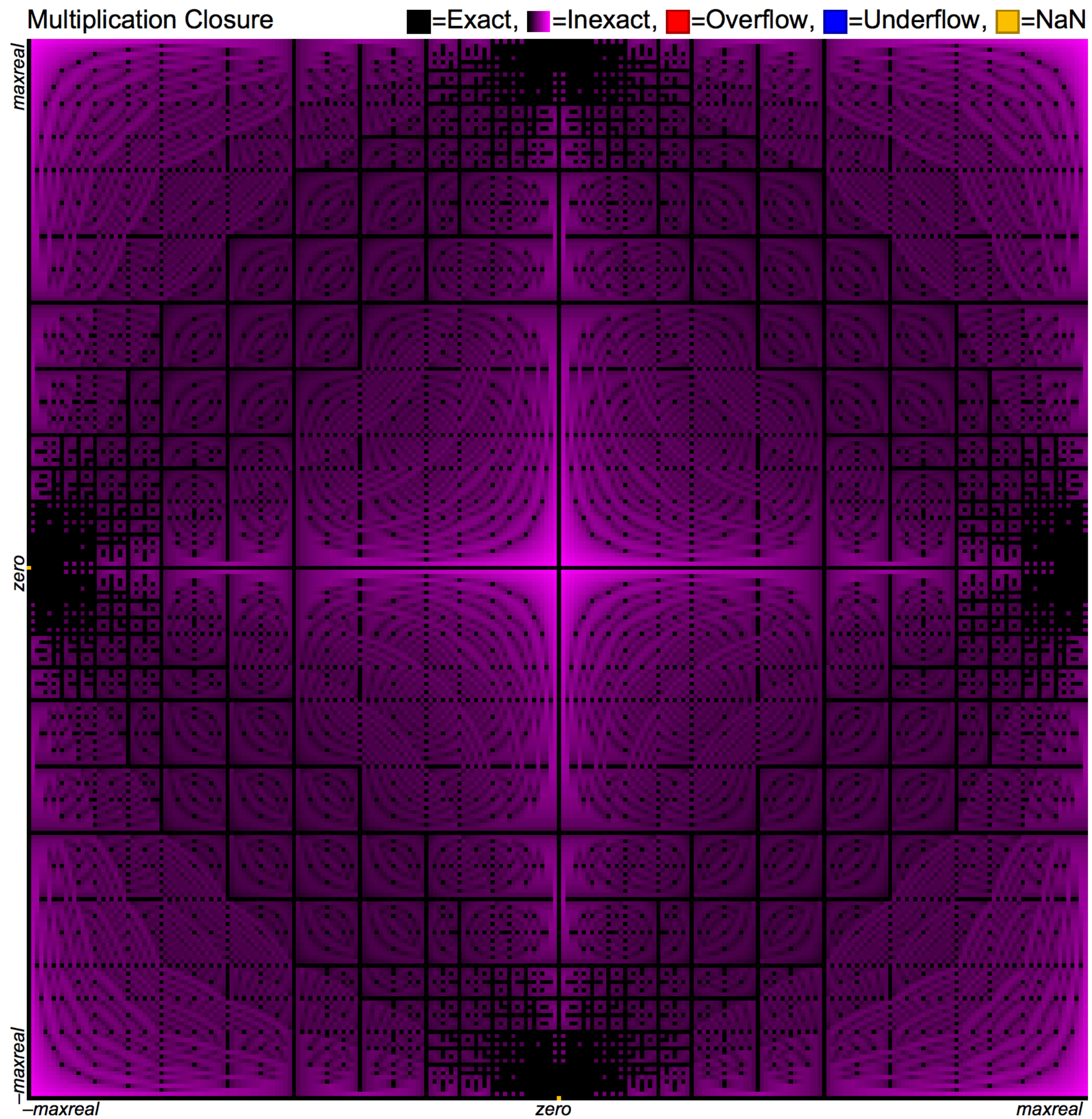


Multiplication Closure Plot: Posits

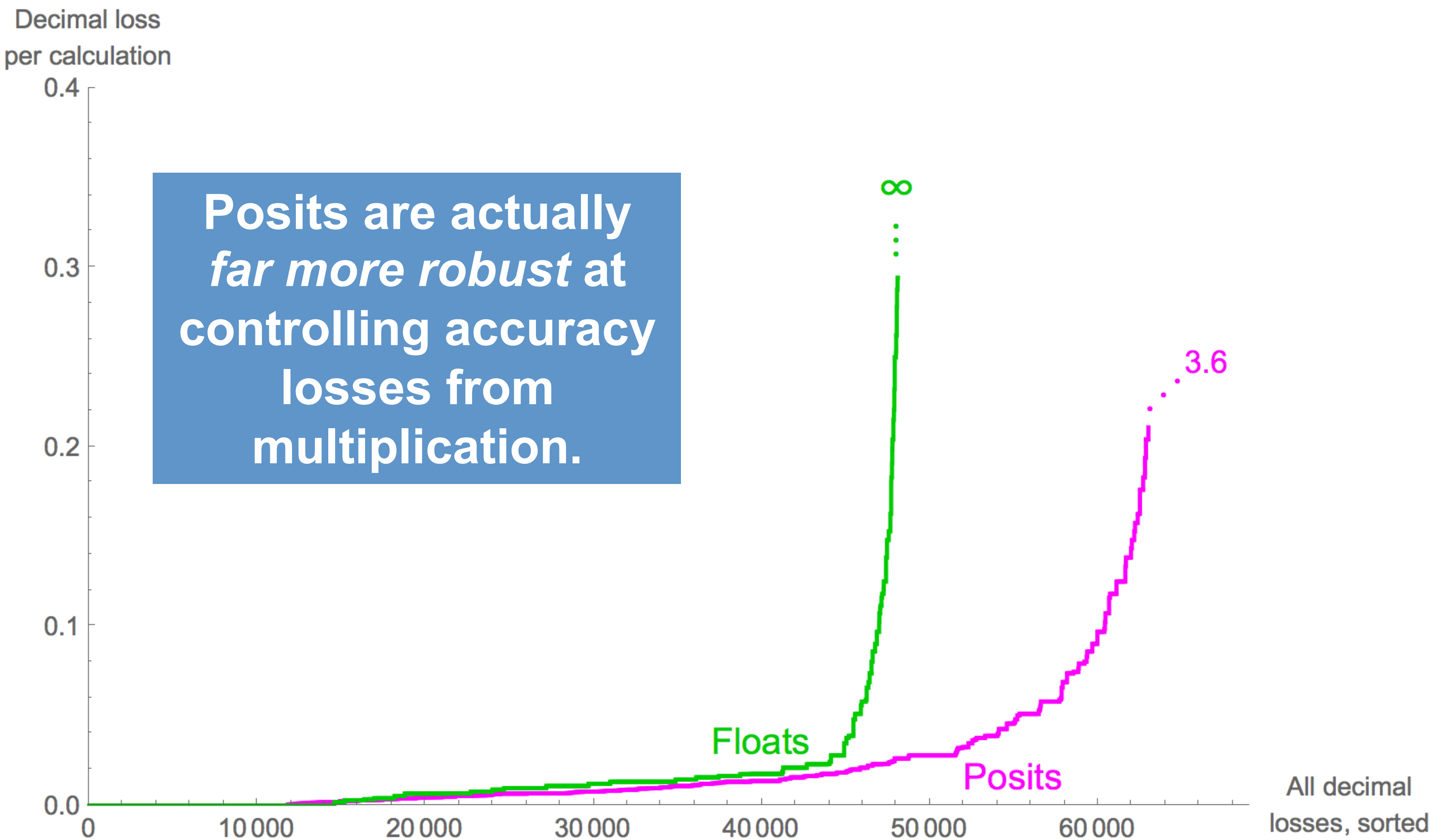
18.002%	exact
81.995%	inexact
0.000%	underflow
0.000%	overflow
0.003%	NaN

Only two cases produce a NaN:

$$\pm\infty \times 0$$
$$0 \times \pm\infty$$



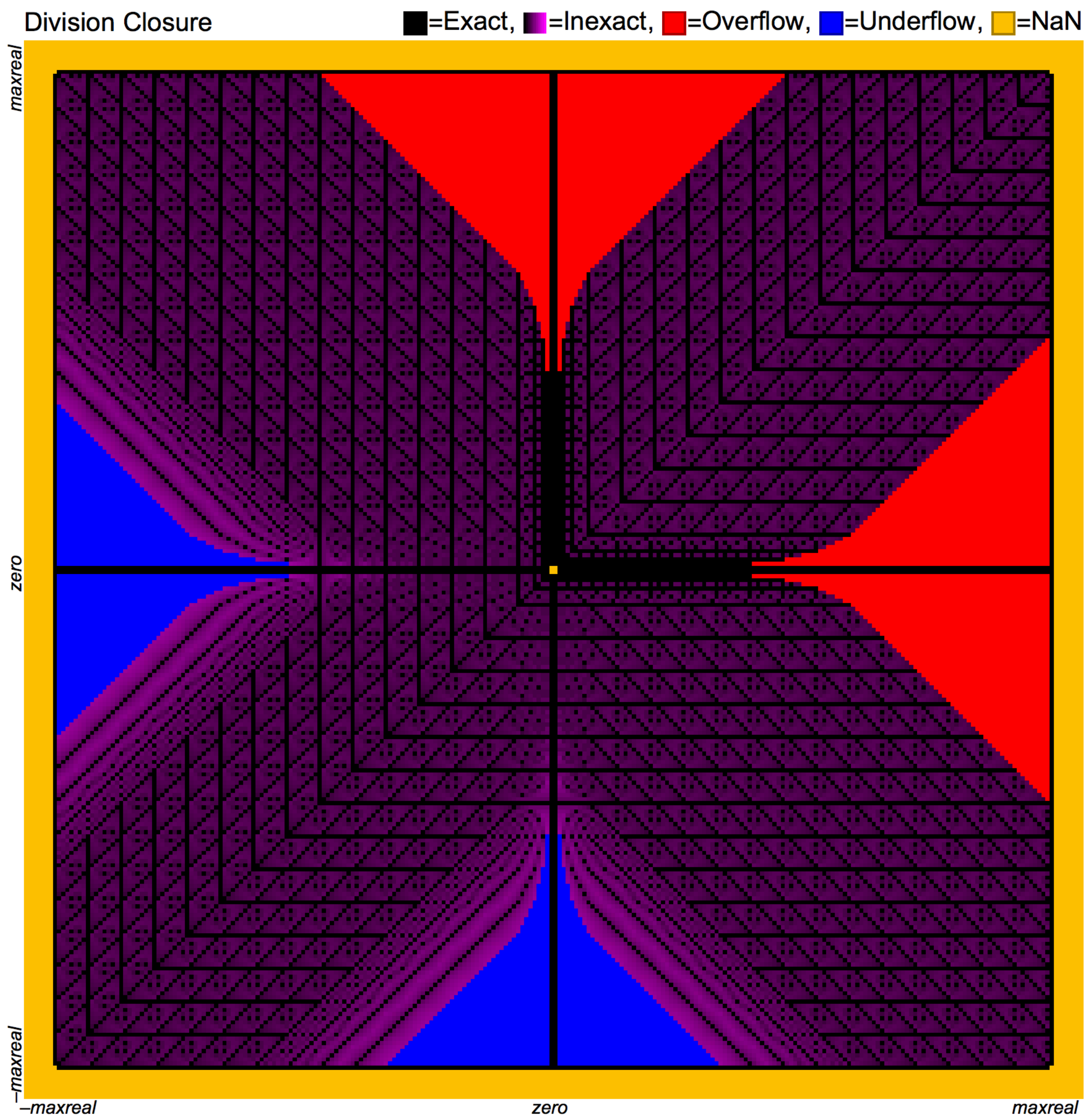
The sorted losses tell the real story



Division Closure Plot: Floats

22.272%	**exact**
58.810%	inexact
3.433%	underflow
4.834%	overflow
10.651%	NaN

Denormalized floats lead to asymmetries.



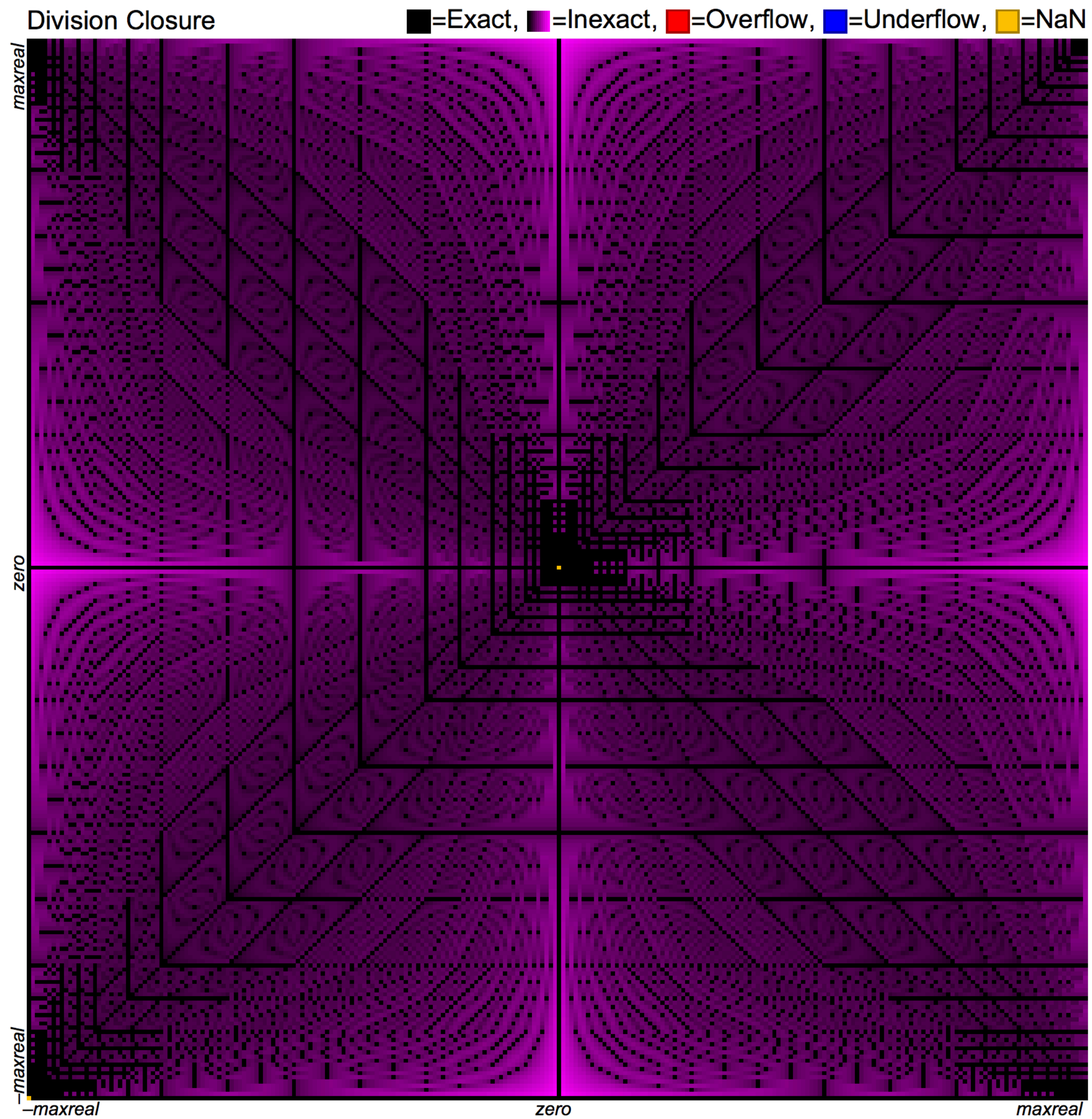
Division Closure Plot: Posits

18.002%	exact
81.995%	inexact
0.000%	underflow
0.000%	overflow
0.003%	NaN

Posits do not have denormalized values. Nor do they need them.

Hidden bit = 1,

always. Simplifies hardware.



ROUND 3

Higher-Precision Operations

32-bit formula evaluation

16-bit linear equation solve

128-bit triangle area calculation

The scalar product, redux

Accuracy on a 32-Bit Budget

Compute: $$\left(\frac{27/10 - e}{\pi - \left(\sqrt{2} + \sqrt{3}\right)}\right)^{67/16} = 302.8827196\ldots$$ with ≤ 32 bits per number.

Number Type	**Dynamic Range**	**Answer**	**Error or Range**
IEEE 32-bit float	2×10^{83}	302.912···	0.0297
Interval arithmetic	10^{12}	[18.21875, 33056.]	3.3×10^{4}
Type 1 unums	4×10^{83}	(302.75, 303.)	0.25
Type 2 unums	10^{99}	302.887···	0.0038
Posits, *es* = 3	3×10^{144}	302.88231···	0.00040
Posits, *es* = 1	10^{36}	302.8827819···	0.000062

Posits beat floats at both dynamic range and accuracy.

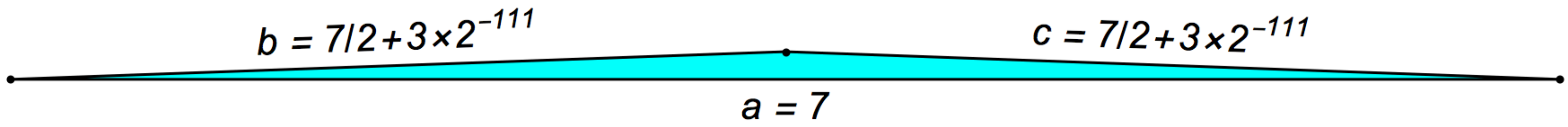
Solving A*x* = *b* with *16-Bit* Numbers

- 10 by 10; random A_{ij} entries in (0, 1)
- *b* chosen so *x* should be all 1s
- Classic LAPACK method: LU factorization with partial pivoting

IEEE 16-bit Floats	16-bit Posits
Dynamic range: 10^{12}	Dynamic range: 10^{16}
RMS error: 0.011	RMS error: 0.0026
Decimals accuracy: 1.96	Decimals accuracy: 2.58

Thin Triangle Area

Find the area of this thin triangle



using the formula $s = \frac{a+b+c}{2}$; $A = \sqrt{s(s-a)(s-b)(s-c)}$

and 128-bit IEEE floats, then 128-bit posits.

Answer, correct to 36 decimals:
$3.14784204874900425235885265494550774\cdots\times10^{-16}$

From “What Every Computer Scientist Should Know About Floating-Point Arithmetic,” David Goldberg, published in the March, 1991 issue of *Computing Surveys*

A Grossly Unfair Contest

IEEE quad-precision floats get only *one decimal digit* right:

$3.634814908423321347259205161580576 83\cdots \times 10^{-16}$

A Grossly Unfair Contest

IEEE quad-precision floats get only *one digit* right:

$3.634814908423321347259205161580576 83\cdots \times 10^{-16}$

128-bit posits get 36 digits right:

$3.147842048749004252358852654945507 74\cdots \times 10^{-16}$

To get this accurate an answer with IEEE floats, you need *octuple* precision (256-bit) representation.

Posits don't even need 128 bits. They can get a very accurate answer with only *119* bits.

Remember this from the beginning?

Find the scalar product $a \cdot b$:

$$a = (3.2\text{e}7,\ 1,\ -1,\ \ 8.0\text{e}7)$$
$$b = (4.0\text{e}7,\ 1,\ -1,\ -1.6\text{e}7)$$

Correct answer: $a \cdot b = 2$

IEEE floats require 80-bit precision to get it right.
Posits ($es = 3$) need only 25-bit precision to get it right.
The ***fused dot product*** is 3 to 6 times **faster** than the float method.*

*Source: "Hardware Accelerator for Exact Dot Product," David Biancolin and Jack Koenig, ASPIRE Laboratory, UC Berkeley

Summary

- Posits beat floats at their own game: superior accuracy, dynamic range, closure
- Bitwise-reproducible answers (at last!)
- Demonstrated *better answers* with same number of bits
- …or, equally good answers with *fewer* bits
- Simpler, more elegant design should reduce silicon cost, energy, and latency.

Who will be the first to produce a chip with posit arithmetic?