Beating Floating Point at its Own Game: Posit Arithmetic

John L. Gustafson
Visiting Scientist at A*STAR and
Professor at National University of Singapore

The “Memory Wall”

Operation	Energy
64-bit floating-point multiply-add	0.2 nanojoules
Read 64 bits from memory (DRAM)	***12*** nanojoules

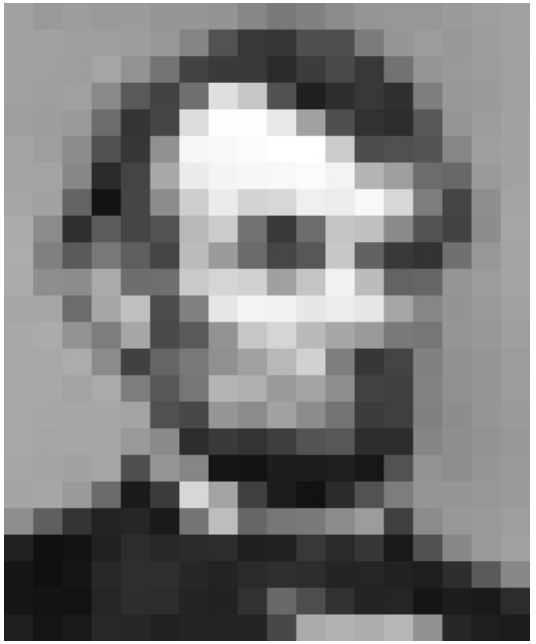
The issue is *communication*, not computation:
Run time, parts cost, electric power.
What if we could cut communication in half
by *doubling information per bit*?

Decades of asking “How do you know your answer is correct?”

- “(Laughter) “What do you mean?”
- “Well, we used double precision.”
- “Oh. We always get that answer.”
- “Other people get that answer.”

Precision versus Accuracy

150,000 pixels

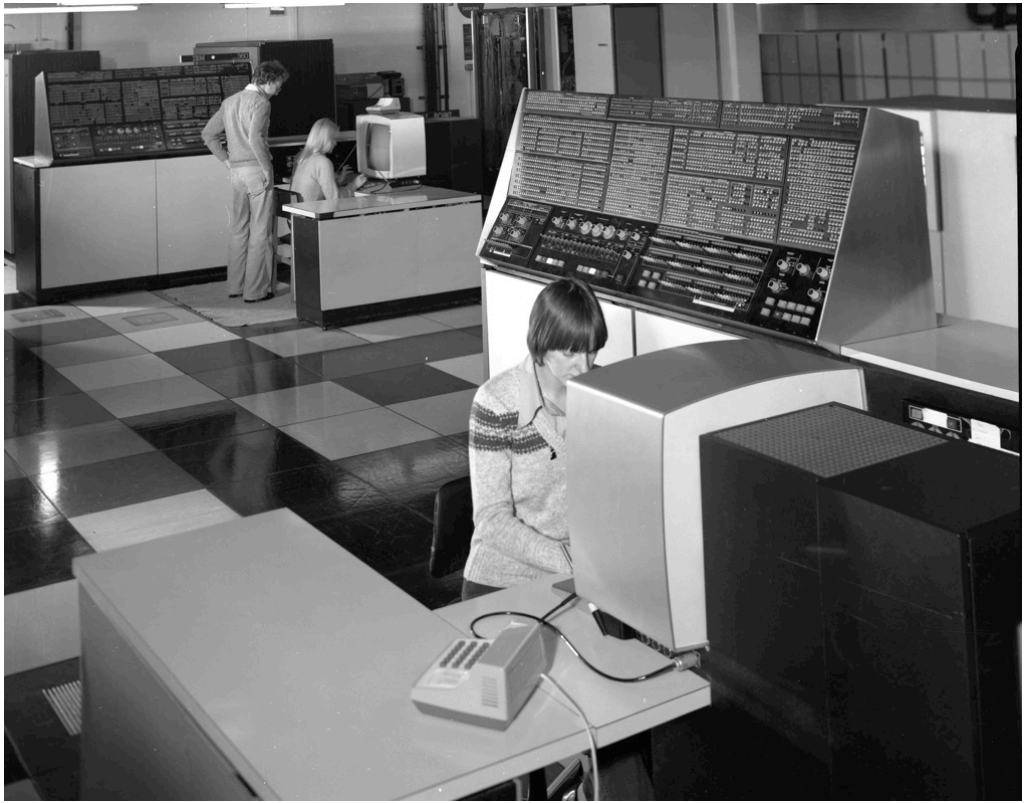


432 pixels

Metrics for Number Systems

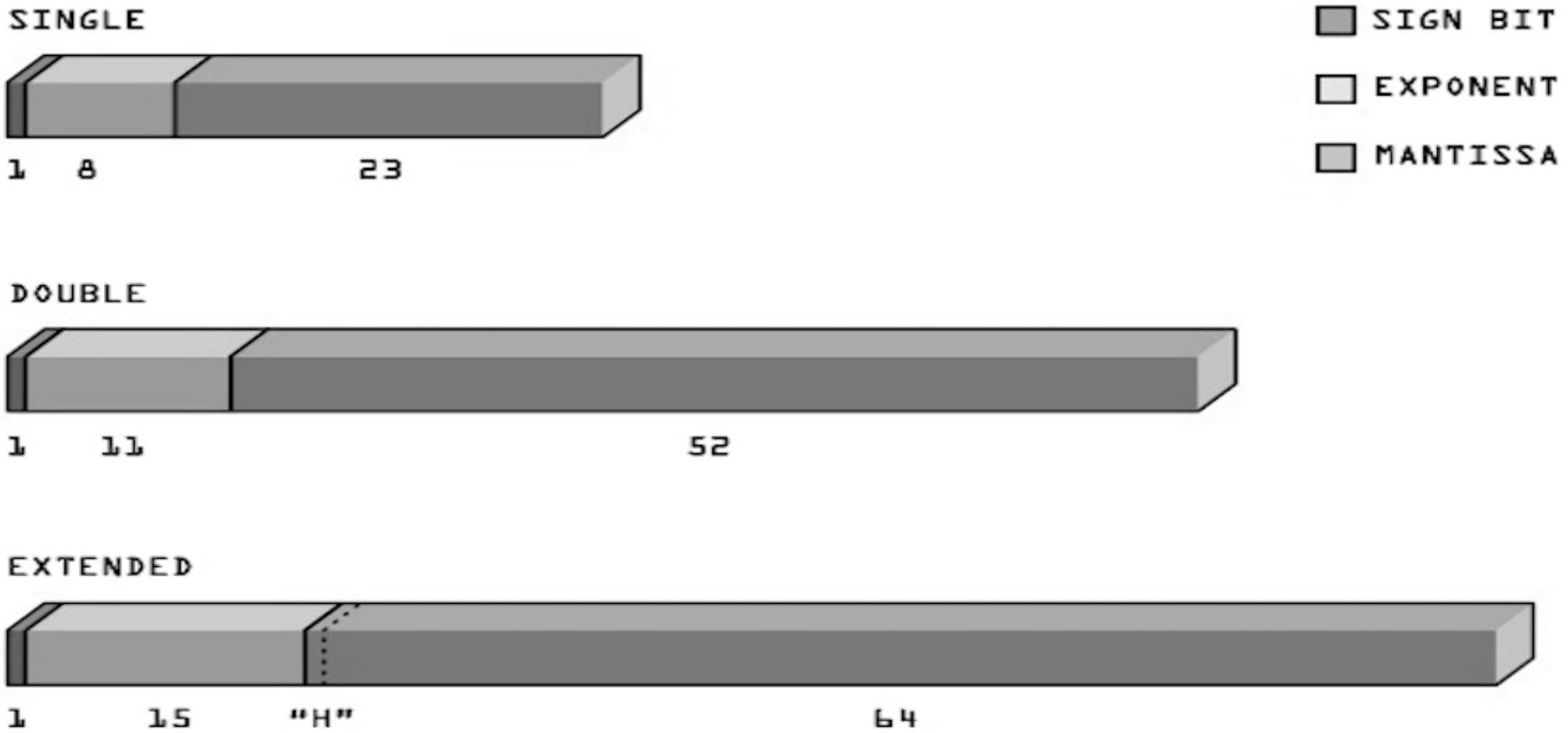
- *Relative Error* = |(correct *x* – computed *x*) / correct *x*|
- *Dynamic range* = $\log_{10}$(*maxreal* / *minreal*) decades
- *Decimal Accuracy* = –$\log_{10}$(*Relative Error*)
- Percentage of operations that are *exact*
  (closure under + – × ÷ √ etc.)
- Average accuracy loss when *inexact*
- Entropy per bit (maximize information)
- Accuracy benchmarks: simple formulas, linear equation solving, math kernels…

COMPUTERS THEN

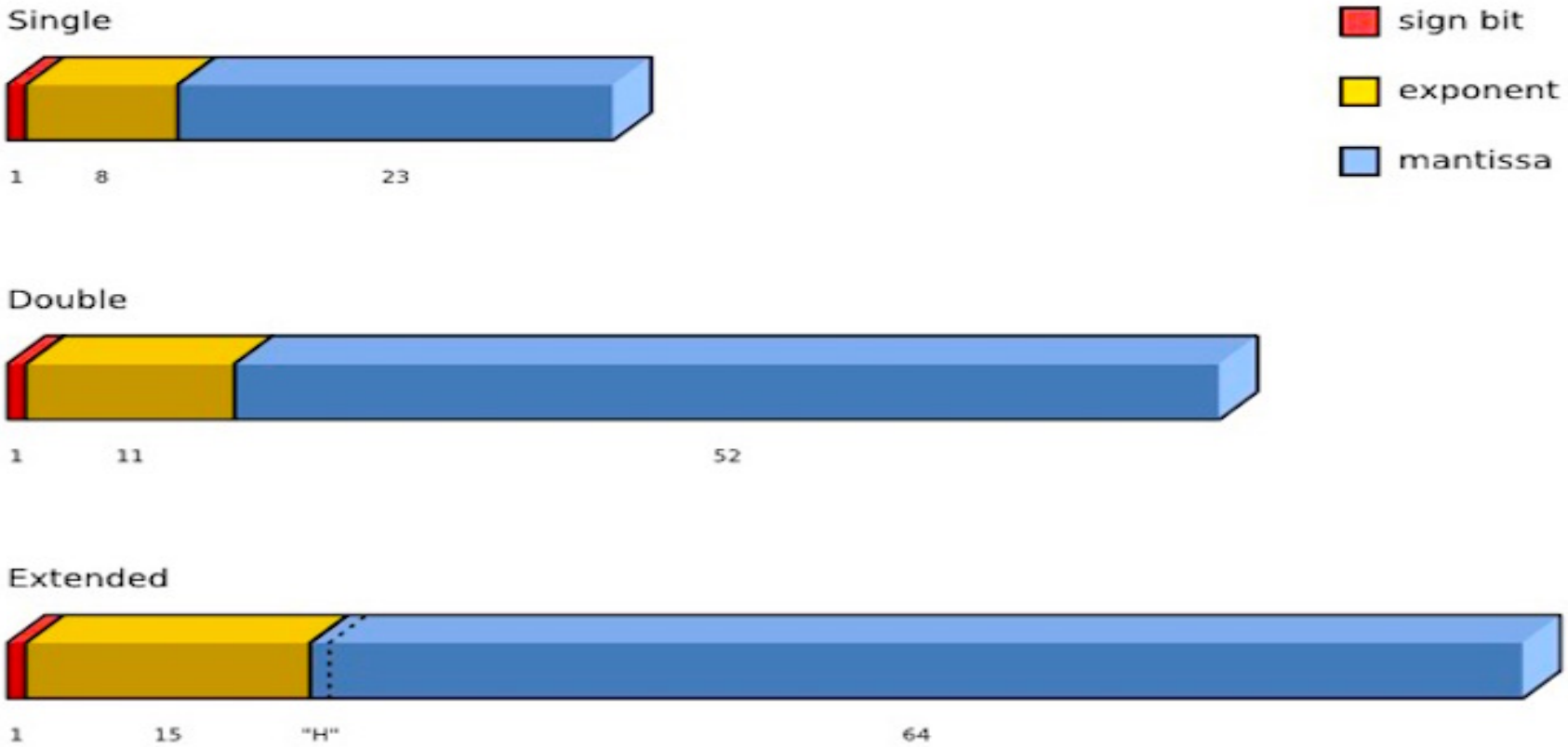


COMPUTERS NOW

ARITHMETIC THEN



ARITHMETIC NOW



Analogy: Printing Technology

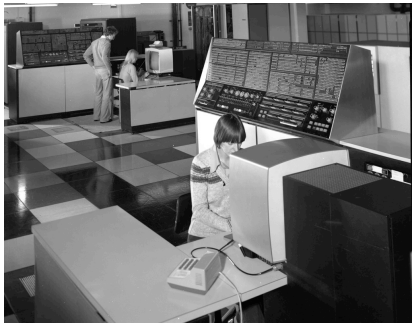
1970: 30 sec

```
      DISK OPERATING SYSTEM/360 FORTRAN   360N-FO-451  CL
C     ROBERT GLASER, RANDALLSTOWN SENIOR, GROUP A, P AND S
C     PRIME NUMBERS
      DO 100 I=1,1000
      J=2
      K=2
    2 L=J*K
      IF (L-I) 10,100,10
   10 M=2+3
      IF (K-I) 20,3,3
   20 K=K+1
      GO TO 2
    3 K=2
      IF (J-I) 5,4,4
    5 J=J+1
      GO TO 2
    4 WRITE (3,6) I
    6 FORMAT (I10)
  100 CONTINUE
      STOP
      END
```

2017: 30 sec

Challenges for the Existing Arithmetic

IEEE Standard Floats are a **storage-inefficient**, **1980s-era** design.



- No guarantee of **repeatable** or **portable** behavior (!)
- Insufficient 32-bit accuracy forces wasteful use of 64-bit types
- Fails to obey laws of algebra (associative, distributive laws)
- Poor handling of overflow, underflow, Not-a-Number results
- Dynamic ranges are too large, stealing accuracy needed for workloads
- Rounding errors are invisible, hazardous, and costly to debug
- Computations are unstable when *parallelized*

Why worry about floating point?

Find the scalar product $a \cdot b$:

a = (3.2e8, 1, –1, 8.0e7)
b = (4.0e7, 1, –1, –1.6e8)

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: $a \cdot b = 0$

Correct answer: $a \cdot b = 2$

Most linear algebra is unstable with floats!

What's wrong with IEEE 754? A start:

- **No guarantee of identical results across systems**
- It's a *guideline*, not a *standard*
- Breaks the laws of algebra:

$$a + (b + c) \neq (a + b) + c \qquad a \cdot (b + c) \neq a \cdot b + a \cdot c$$

- Overflow to infinity creates infinite *relative error.*

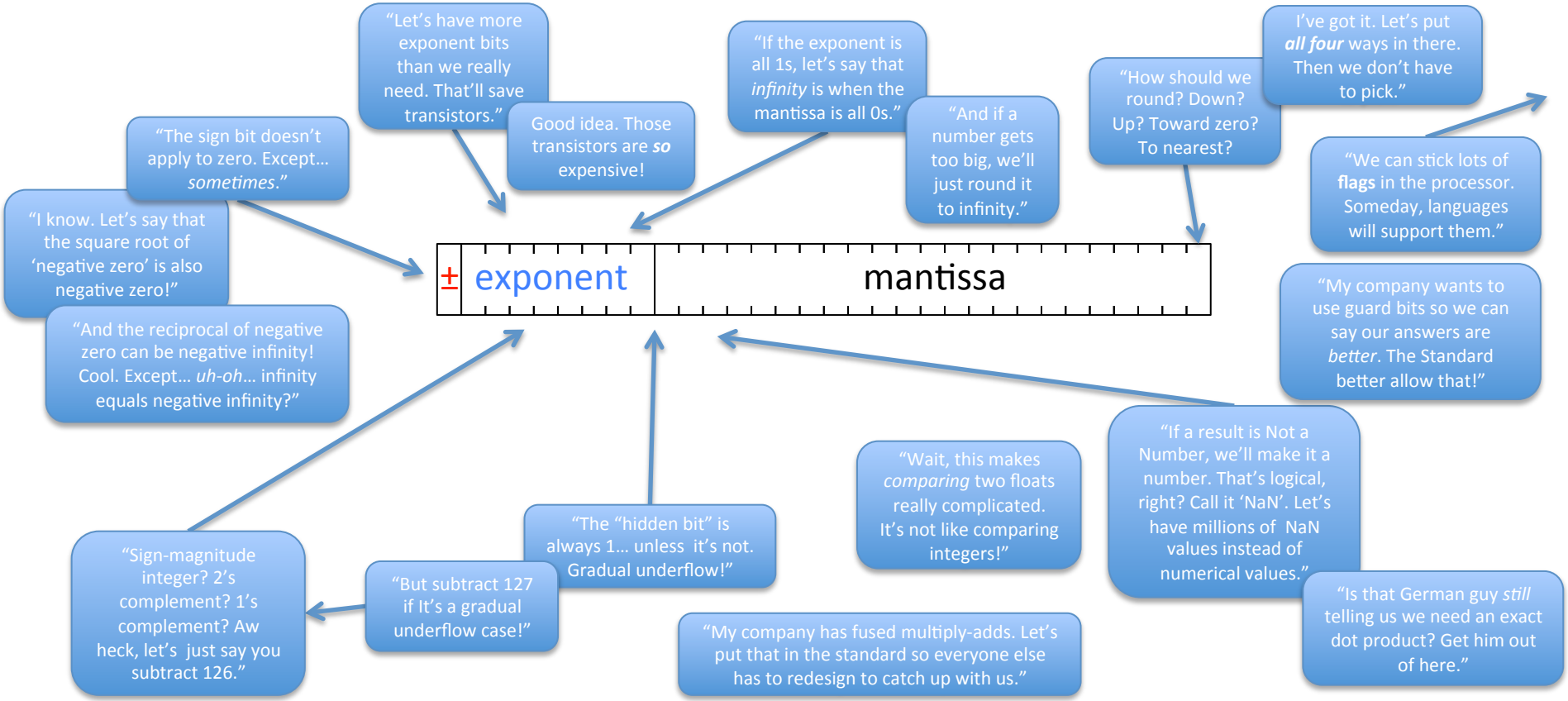
IEEE floats are weapons of math destruction.

What *else* is wrong with IEEE 754?

- Exponents usually take too many bits
- Accuracy is flat across a vast range, then falls off a cliff
- Subnormal numbers are a headache (“gradual underflow”)
- Divides are messy and slow
- Wasted bit patterns: “negative zero,” too many NaN values

Do we really need 9,007,199,254,740,990 numbers to indicate something is *Not a Number*??

Floats: Designed by a 1980s Committee



Contrasting Calculation "Esthetics"

	Rounded: cheap, uncertain, "good enough"	**Rigorous: more work, certain, mathematical**
IEEE Standard (1985)	Floats, $f = n \times 2^m$ m, n are integers	Intervals $[f_1, f_2]$, all x such that $f_1 \le x \le f_2$

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

"I need the hardware to protect me from NaN and overflow in my code."

"Really? And do you keep debugger mode turned on in your production software?"

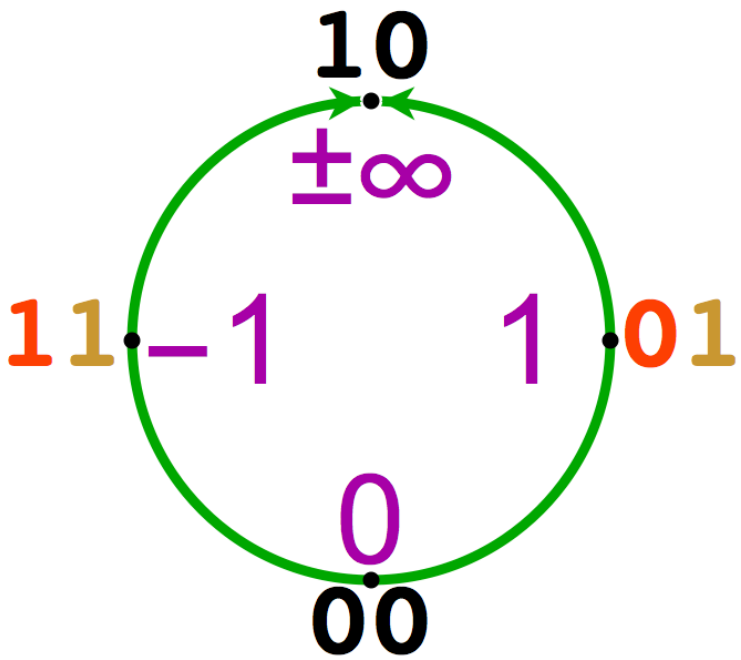
posit | ˈpäzət |

noun *Philosophy*

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

Posits use the *Projective* Reals

- Posits map reals to standard *signed integers*.
- Can be as small as *2 bits* and still be useful!
- This eliminates “negative zero” and other IEEE float issues



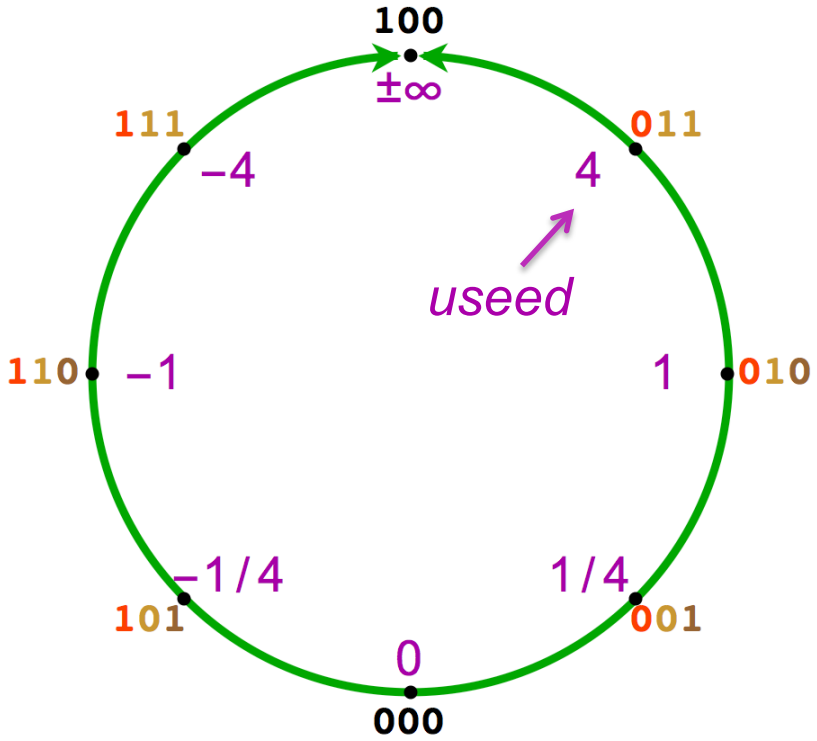
Mapping to the Projective Reals

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.

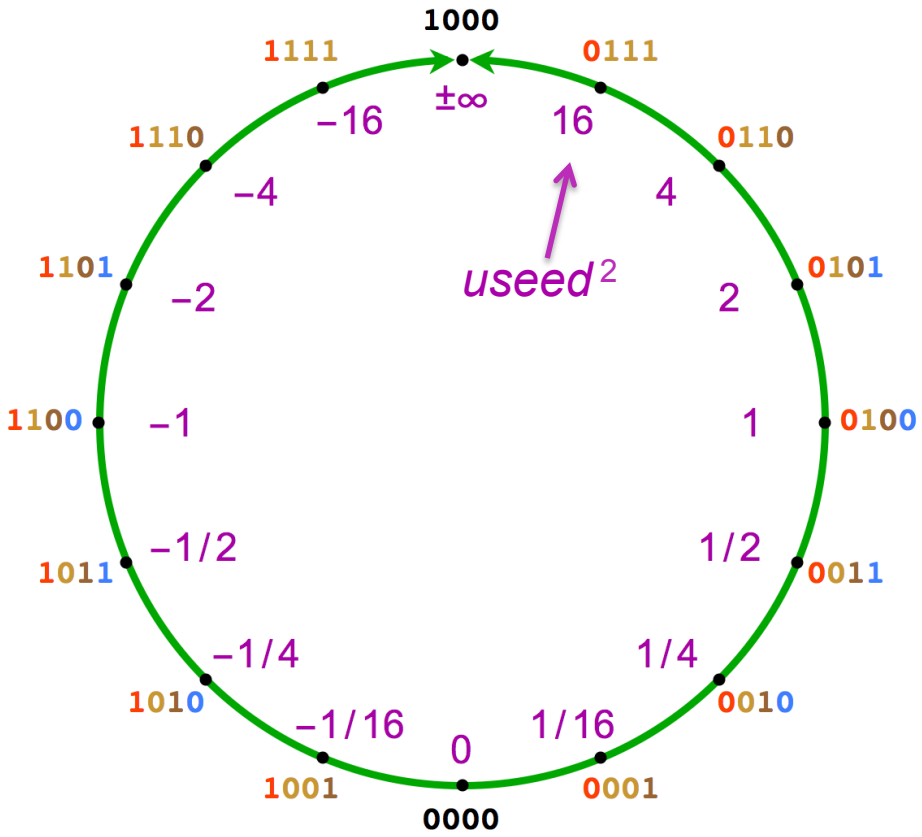


Rules for inserting new points

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 $n - m \geq 2$, insert $2^{(m+n)/2}$.
(New **exponent** bit)

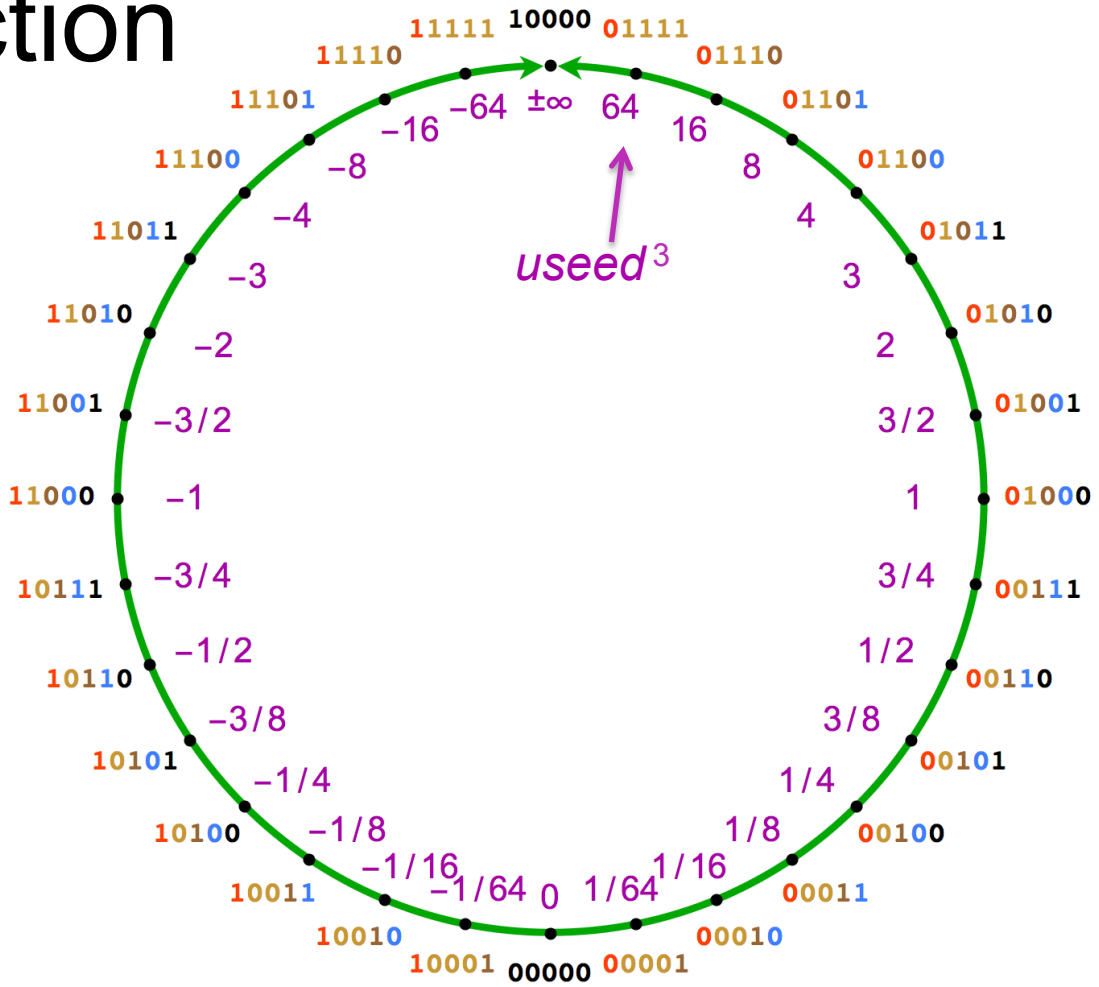


At *nbits* = 5, fraction bits appear.

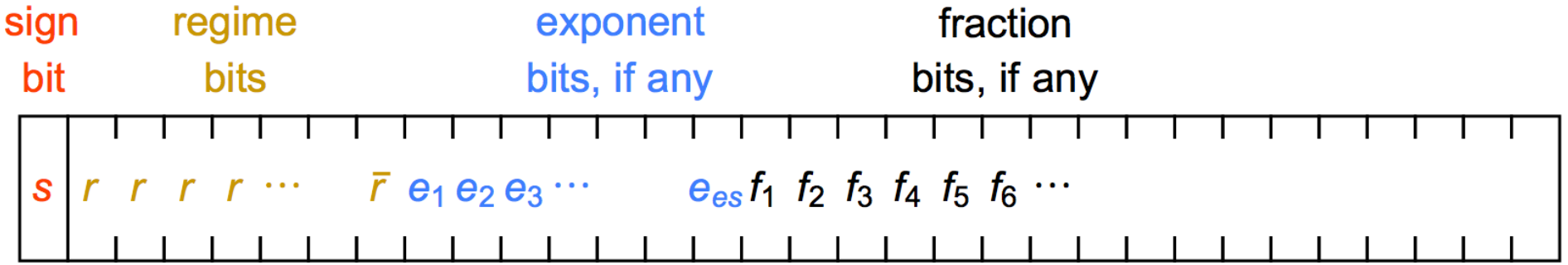
Between *x* and *y* where $y \le 2x$, insert $(x + y)/2$.

Existing values stay put as *trailing* bits are added.

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



Posit Arithmetic:
Beating floats at their own game

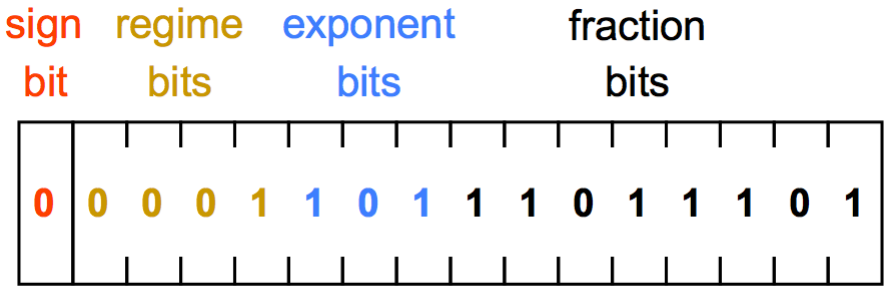


Fixed size, *nbits*.

es = exponent size = 0, 1, 2,… bits.

es is also the number of times you square 2 to get *useed*: 2, 4, 16, 256, 65536,…

Posit Format Example



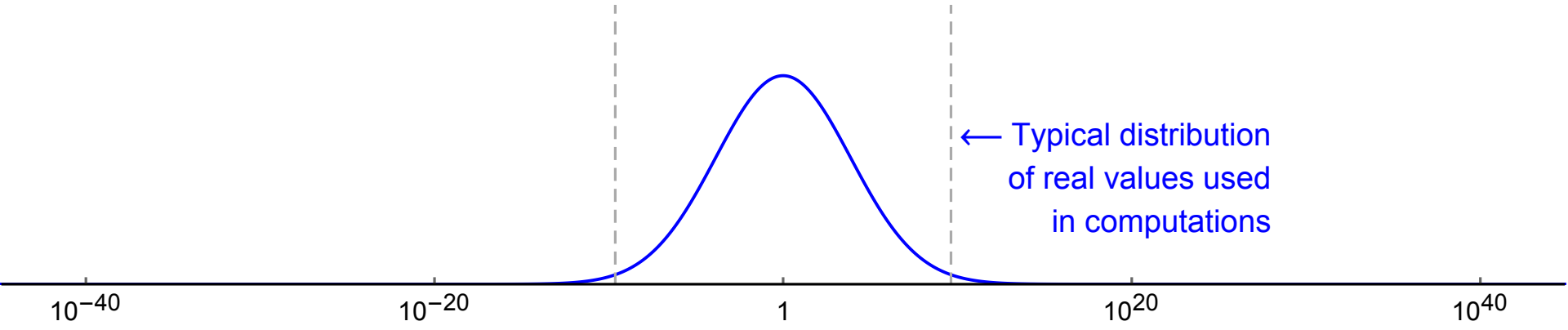
$= 3.55\cdots\times10^{-6}$

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

Posits **do not overflow**. There is no NaN. Relative error ≤ 1.

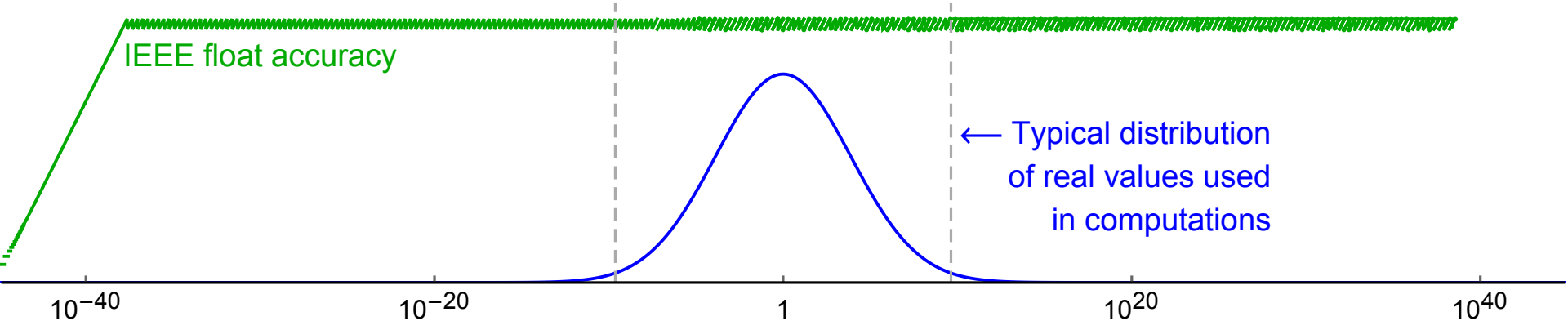
Simpler, faster circuits than IEEE 754

What reals should we seek to represent?



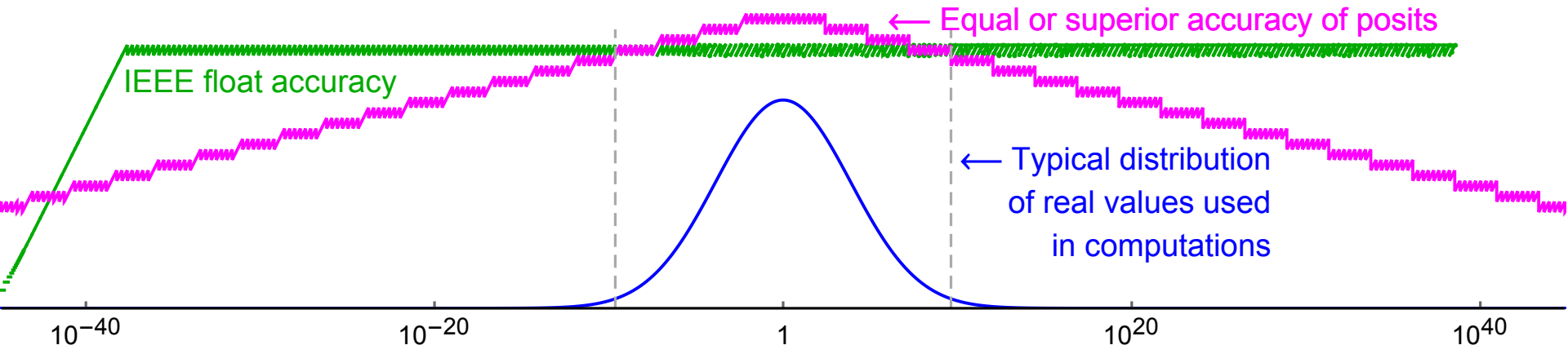
Studies show *very* rare use of values outside 10^{-13} to 10^{13}.
Central Limit Theorem says exponents distribute as a bell curve.

IEEE floats have about 7 decimals accuracy, flat except on the left



This shows 32-bit float accuracy. Dynamic range of 83 decades.
For 64-bit floats, exponent range is even sillier: *631* decades.
Is *flat* accuracy over a huge range really what we need?

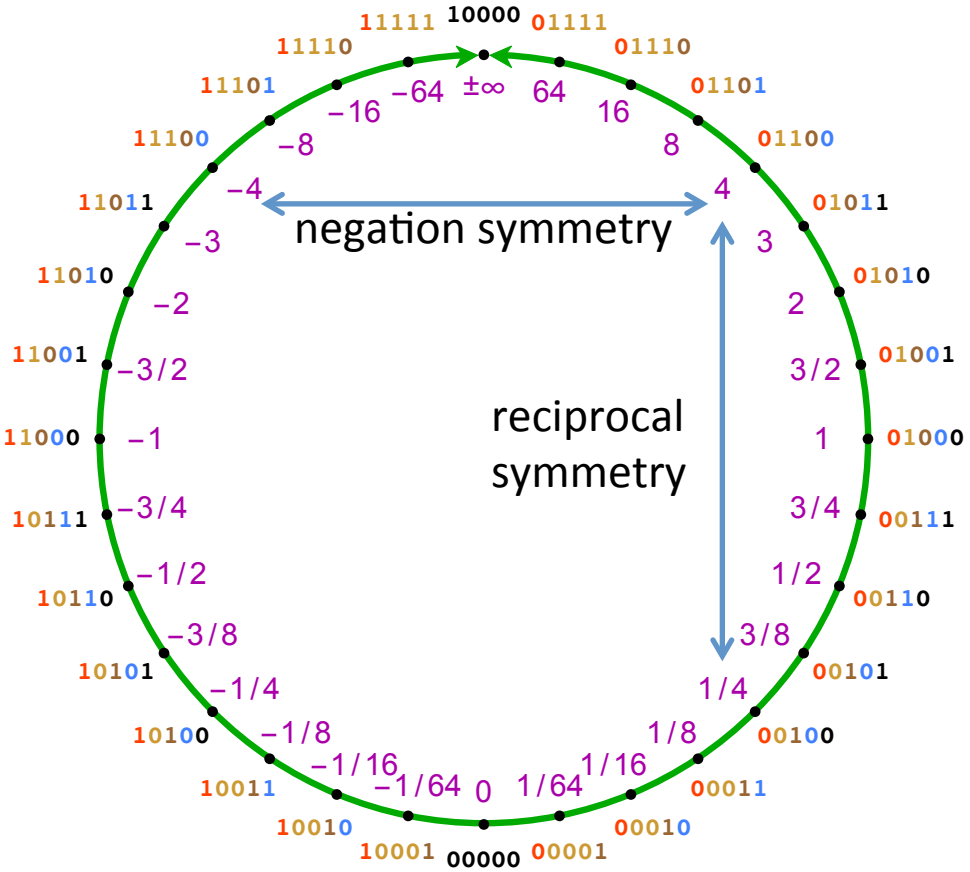
Posits provide concise *tapered accuracy*



Posits have same or better accuracy on the vast majority of calculations, yet have *greater* dynamic range.

This is only *one* major advantage of posits over floats.

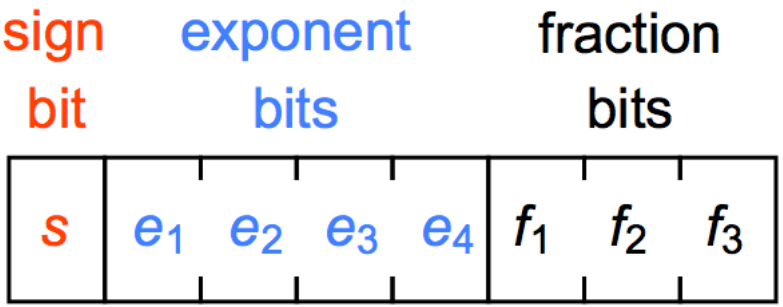
Posits: Designed by *Mathematics*



- **1-to-1 map of *binary integers* to *ordered real numbers***
- Appending bits gives isomorphic increase in precision *and* dynamic range, automatically
- No “negative zero”
- No bit patterns wasted on “NaN”
- Simpler circuitry, less chip area
- No hidden and unused flags
- More information per bit
- As reproducible as integer math; no “hidden scratchpad” work
- Obeys mathematical laws

Posits v. Floats: a *metrics-based* study

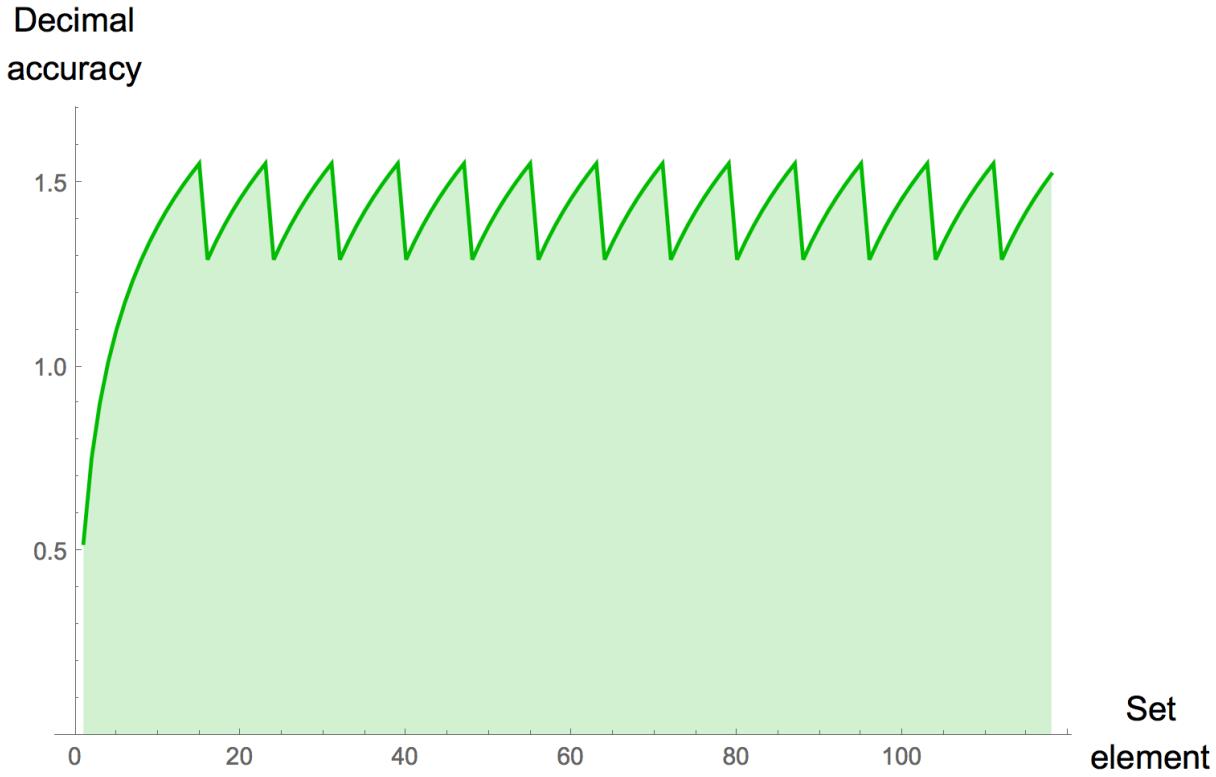
- Compare *quarter-precision* IEEE-style floats
- Sign bit, 4 exponent bits, 3 fraction bits



- *smallsubnormal* = 1/512; *maxfloat* = 240.
- Dynamic range of five orders of magnitude
- Two bit patterns that mean zero
- Fourteen bit patterns that mean “Not a Number” (NaN)

Float accuracy tapers only on left

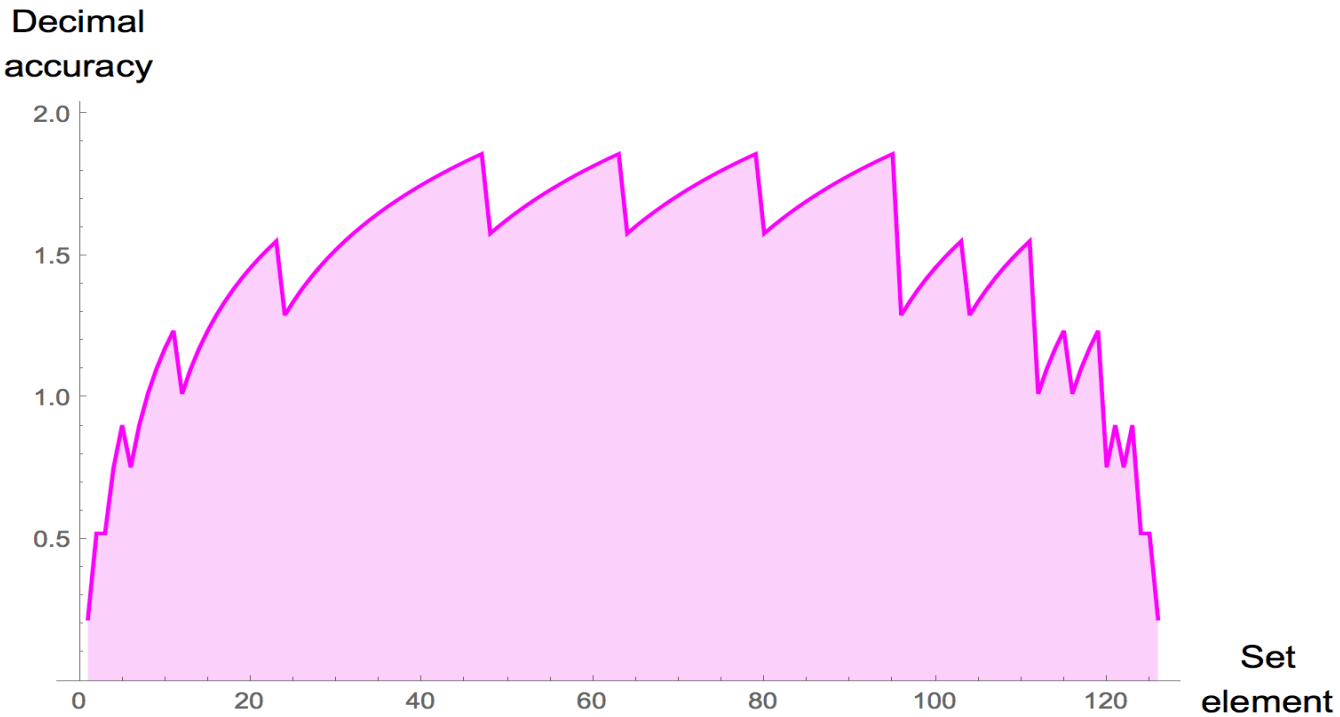
- Min: **0.52** decimals
- Avg: **1.40** decimals
- Max: **1.55** decimals



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

Posit accuracy tapers on both sides

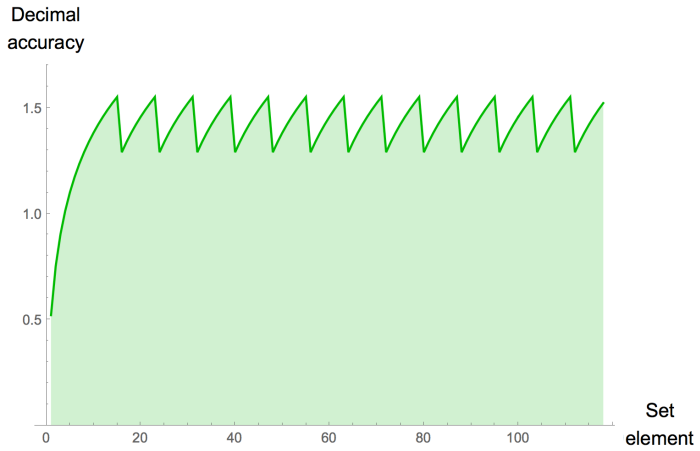
- Min: **0.22** decimals
- Avg: **1.46** decimals
- Max: **1.86** decimals

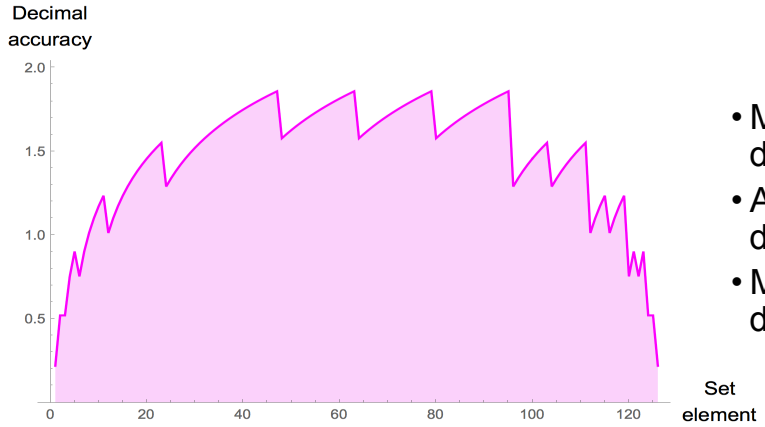


Graph shows decimals of accuracy from *minpos* to *maxpos*.
But posits cover *seven* orders of magnitude, not five.

What do these remind you of?

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

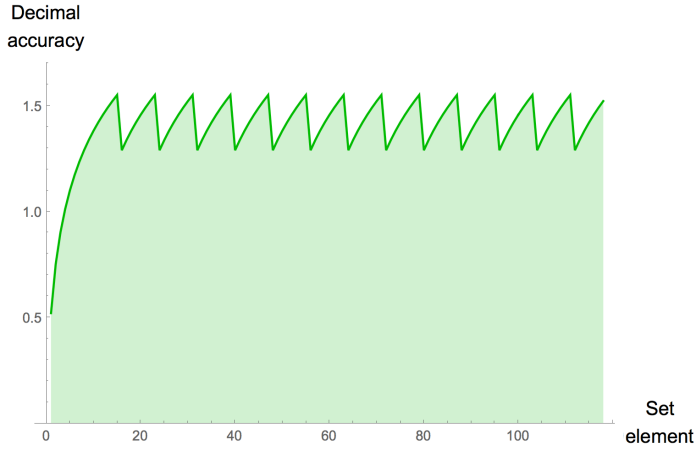


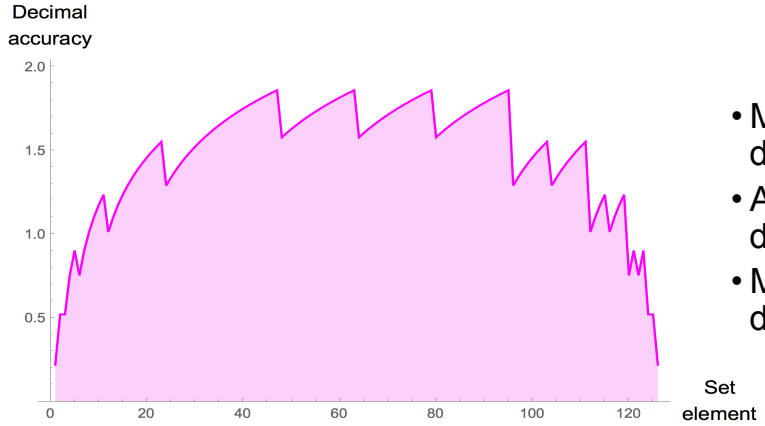


- Min: **0.22** decimals
- Avg: **1.46** decimals
- Max: **1.86** decimals

What do these remind you of?

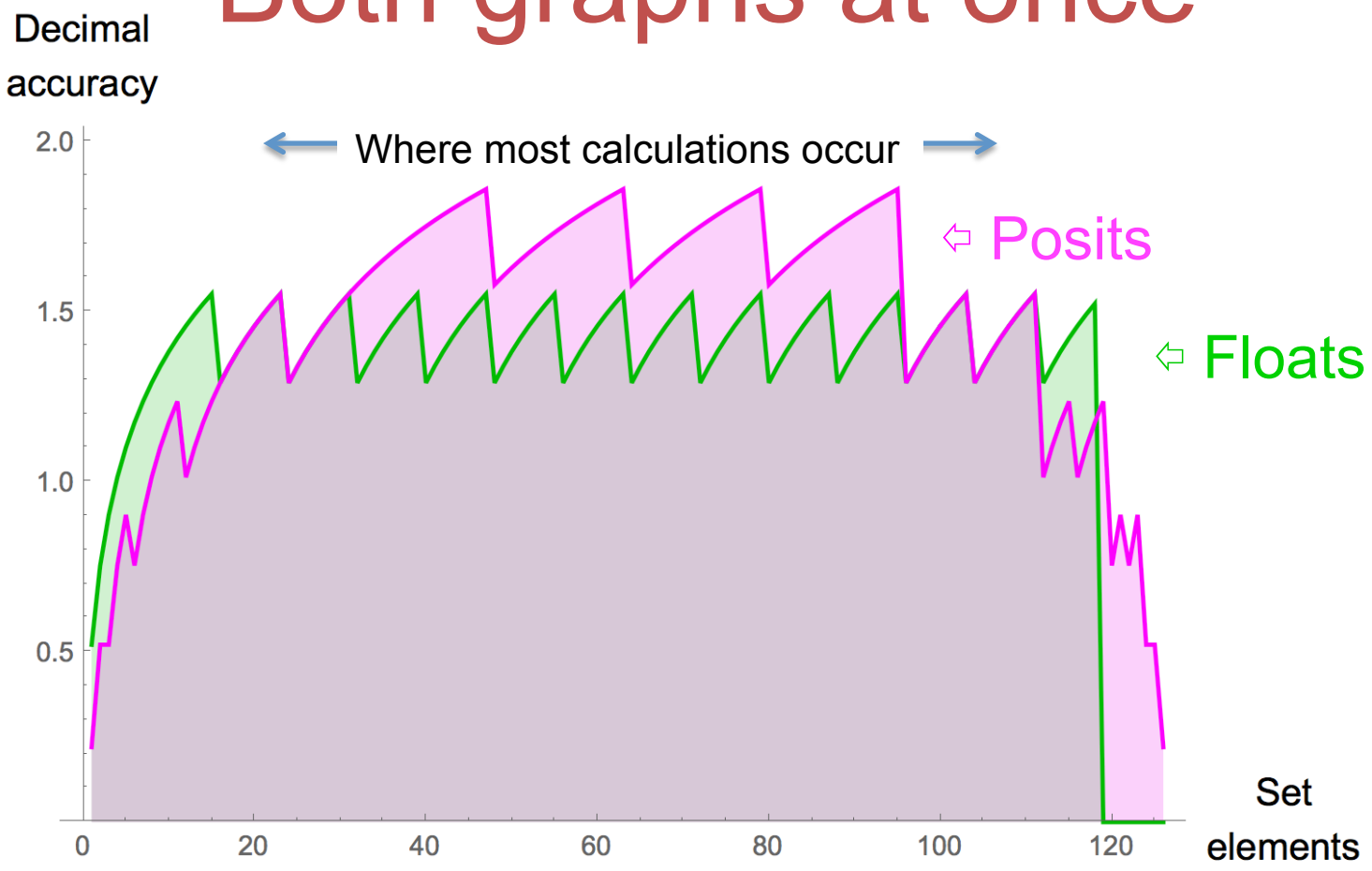
- Min: **0.52** decimals
- Avg: **1.40** decimals
- Max: **1.55** decimals





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

Both graphs at once



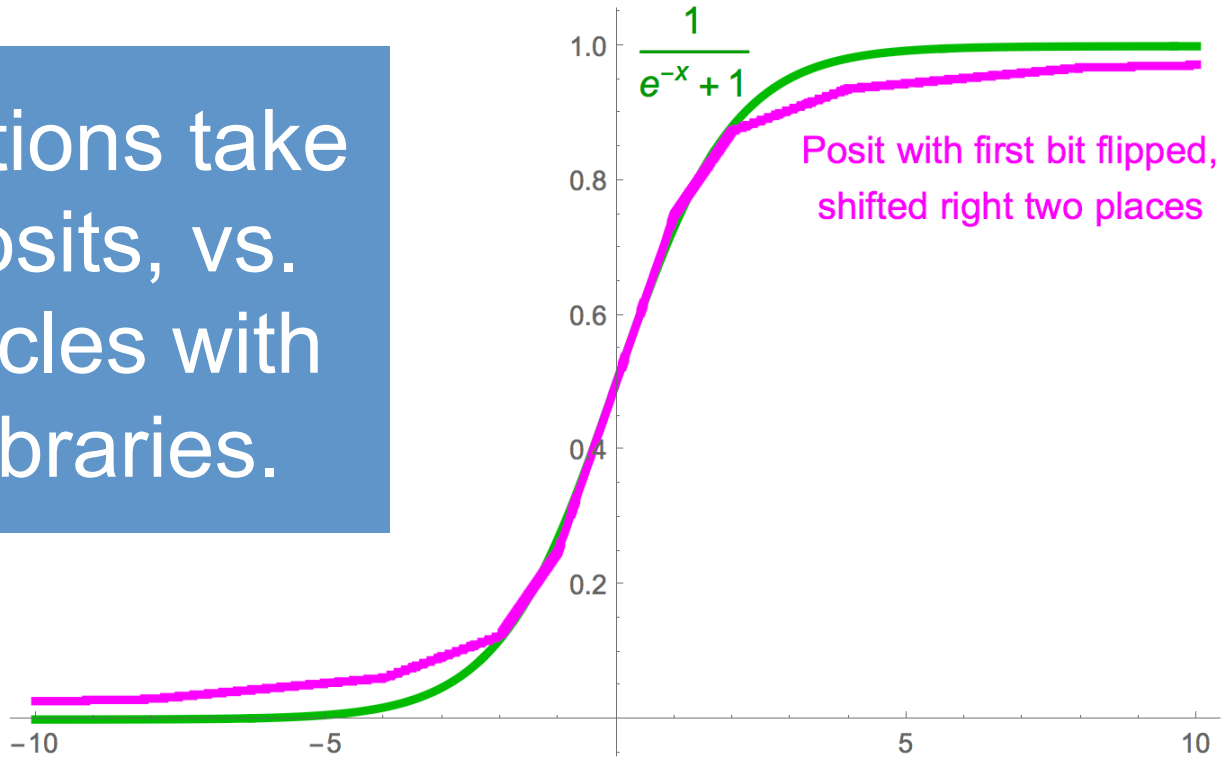
Matching float dynamic ranges

Size, bits	Float exponent size	Float dynamic range	Posit *es* value	Posit dynamic range
16	5	6×10^{-8} to 7×10^{4}	1	4×10^{-9} to 3×10^{8}
32	8	1×10^{-45} to 3×10^{38}	3	6×10^{-73} to 2×10^{72}
64	11	5×10^{-324} to 2×10^{308}	4	2×10^{-299} to 4×10^{298}
128	15	6×10^{-4966} to 1×10^{4932}	7	1×10^{-4855} to 1×10^{4855}
256	19	2×10^{-78984} to 2×10^{78913}	10	2×10^{-78297} to 5×10^{78296}

Note: Isaac Yonemoto has shown that 8-bit posits suffice for neural network training, with *es* = 0

8-bit posits for fast *neural nets*

Sigmoid functions take 1 cycle in posits, vs. dozens of cycles with float math libraries.



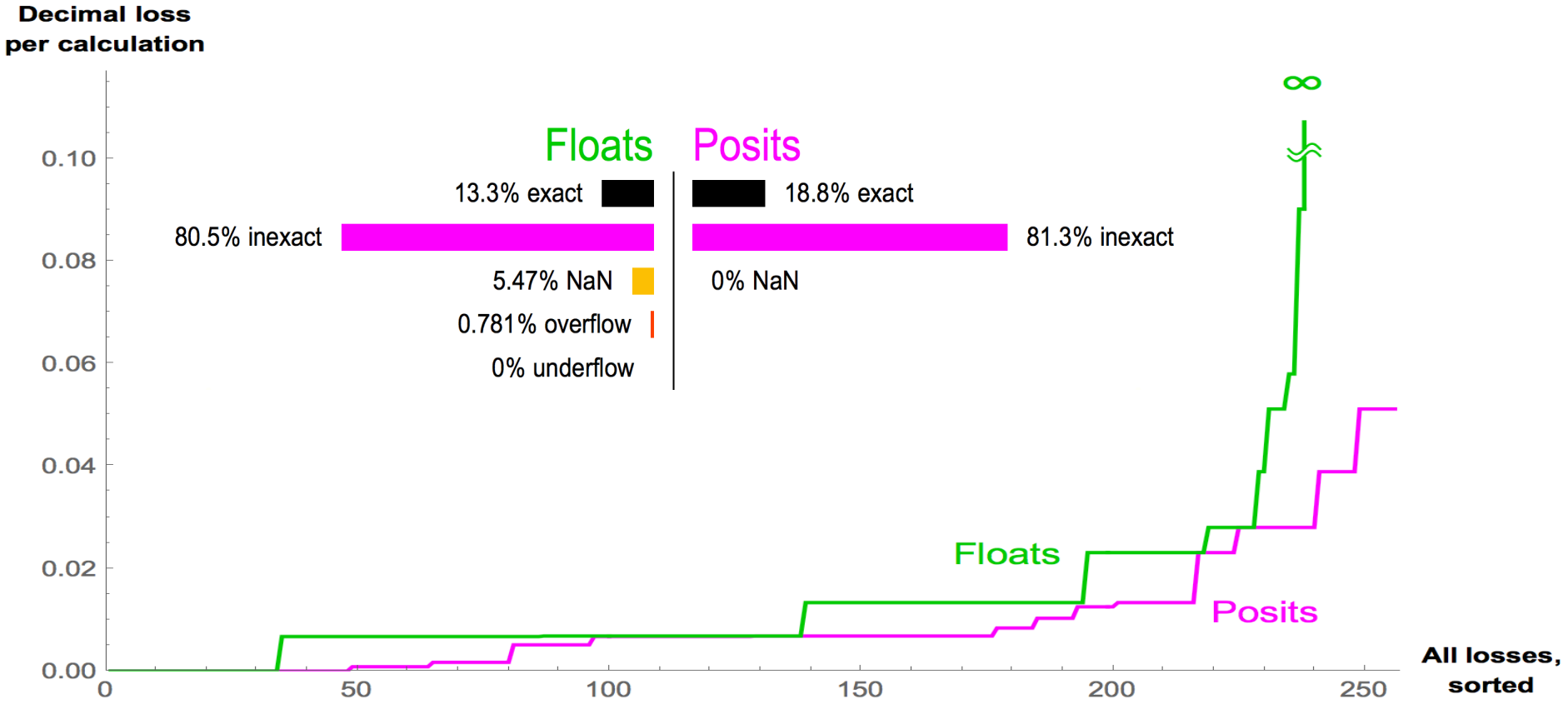
(Observation by I. Yonemoto)

ROUND 1

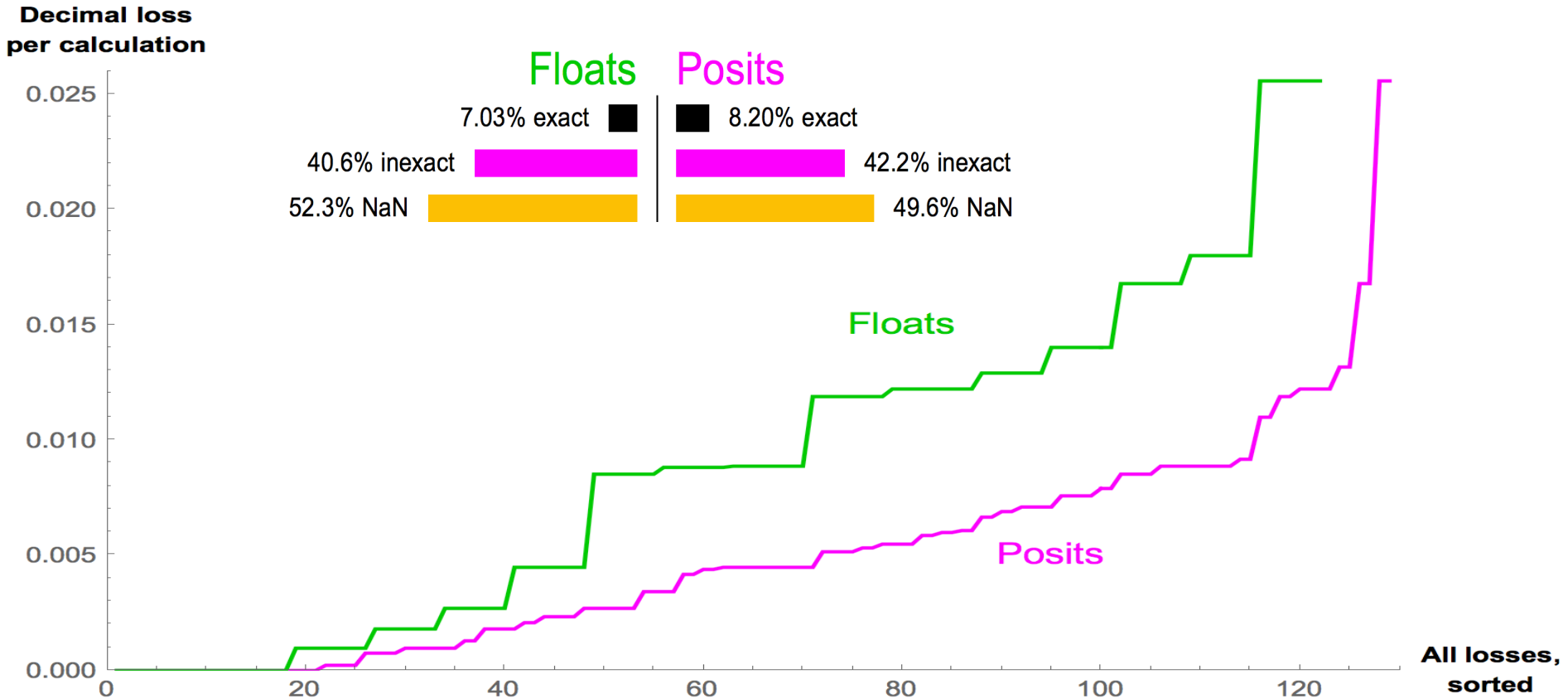
Unary Operations

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

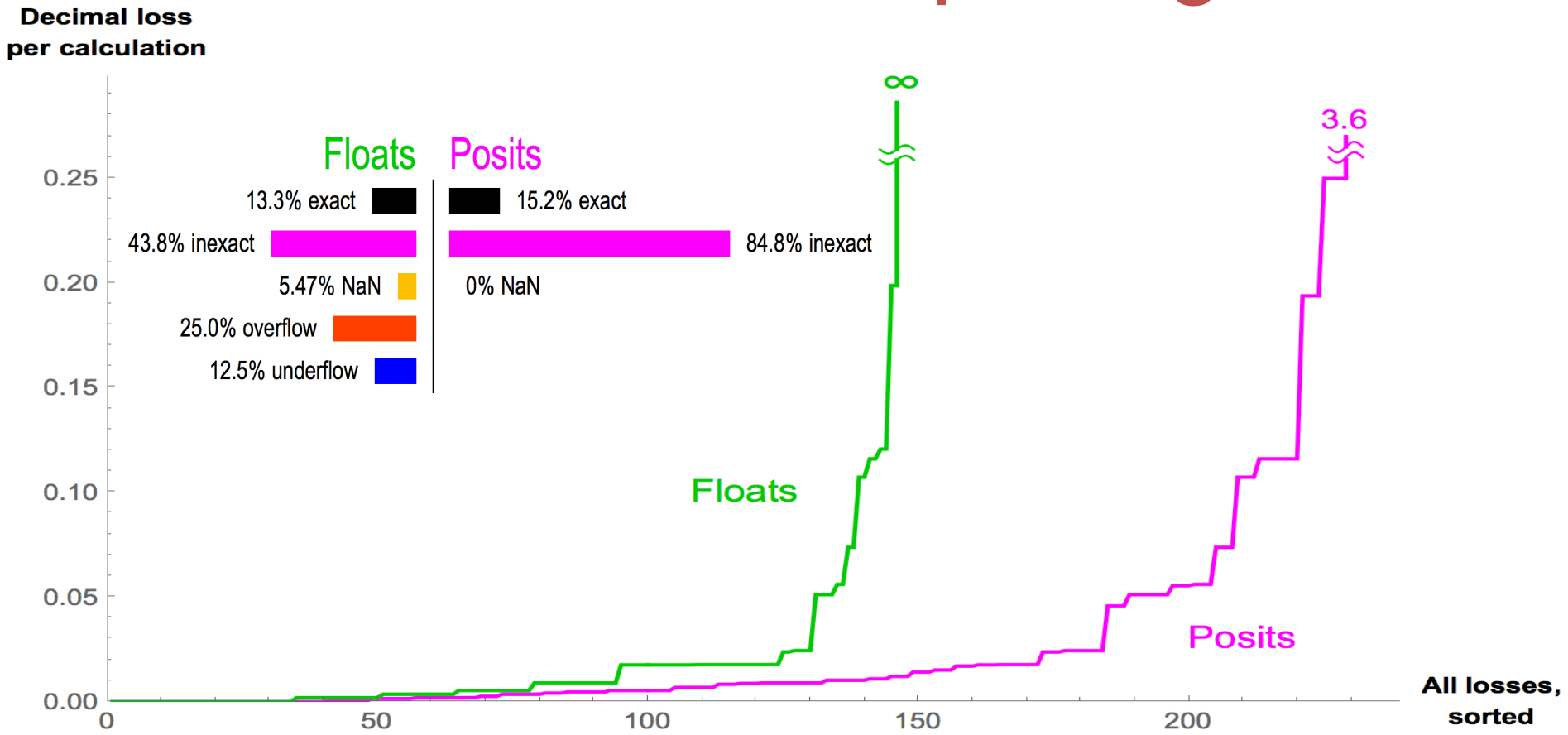
Closure under Reciprocation, 1/*x*



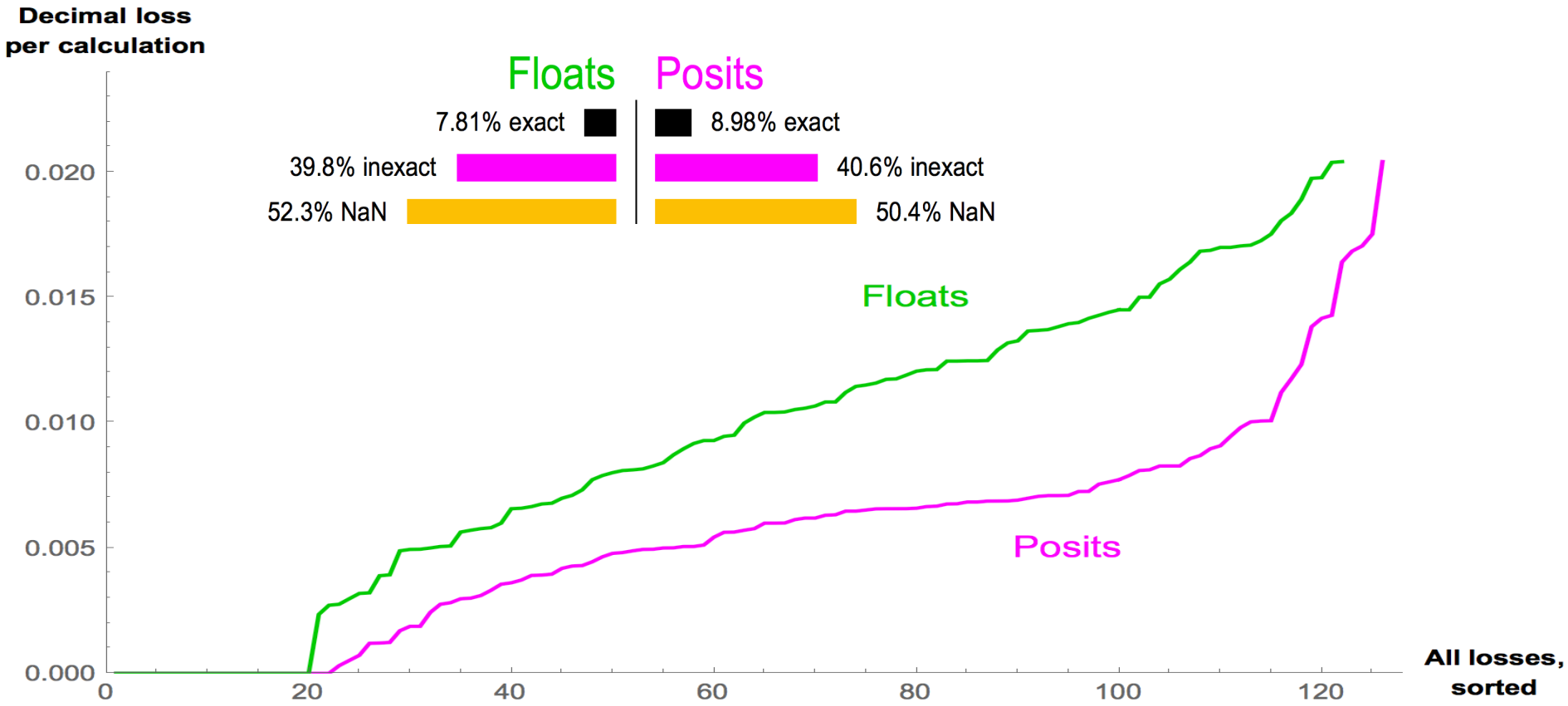
Closure under Square Root, $\sqrt{x}$



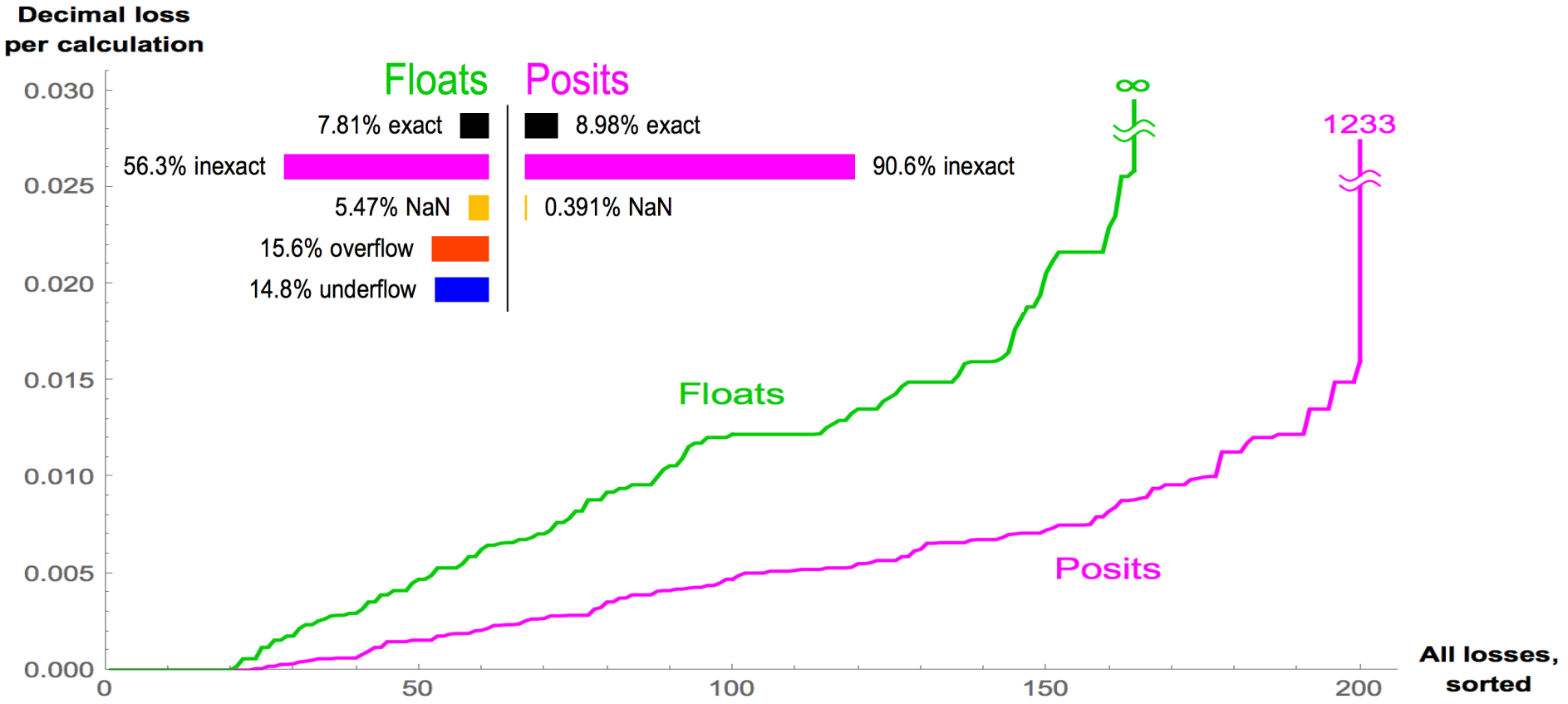
Closure under Squaring, x^2



Closure under $\log_2(x)$



Closure under 2^x



ROUND 2

Two-Argument Operations

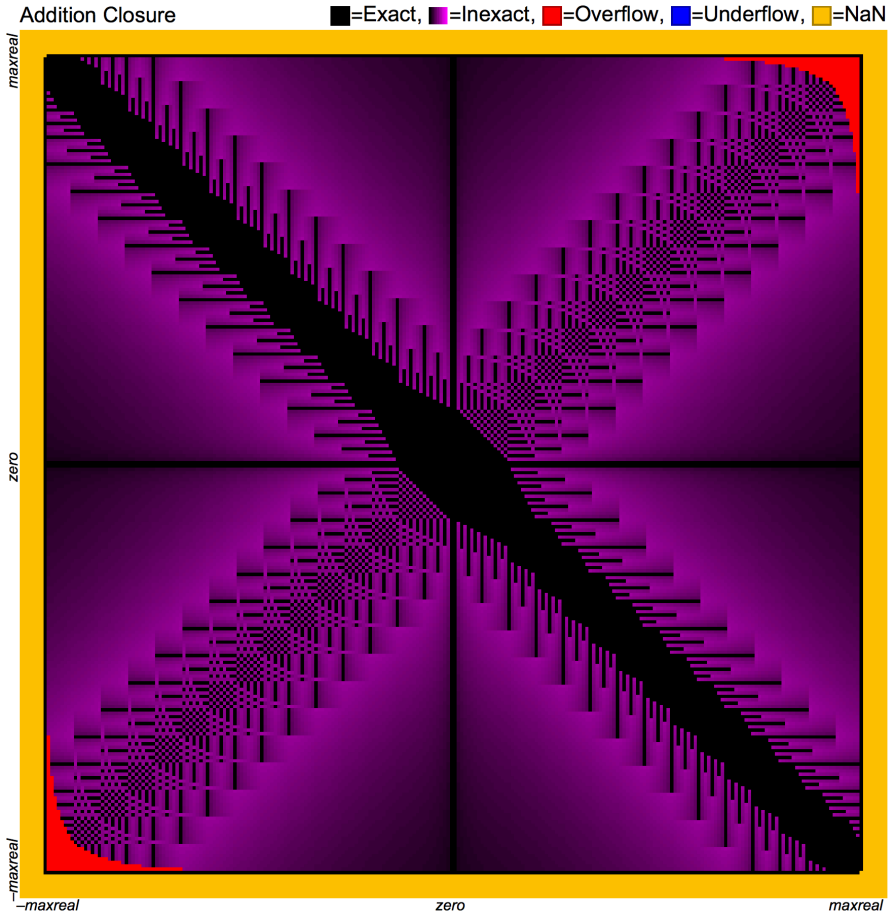
$x + y$, $x \times y$, $x \div y$

Addition Closure Plot: Floats

18.533%	exact
70.190%	inexact
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

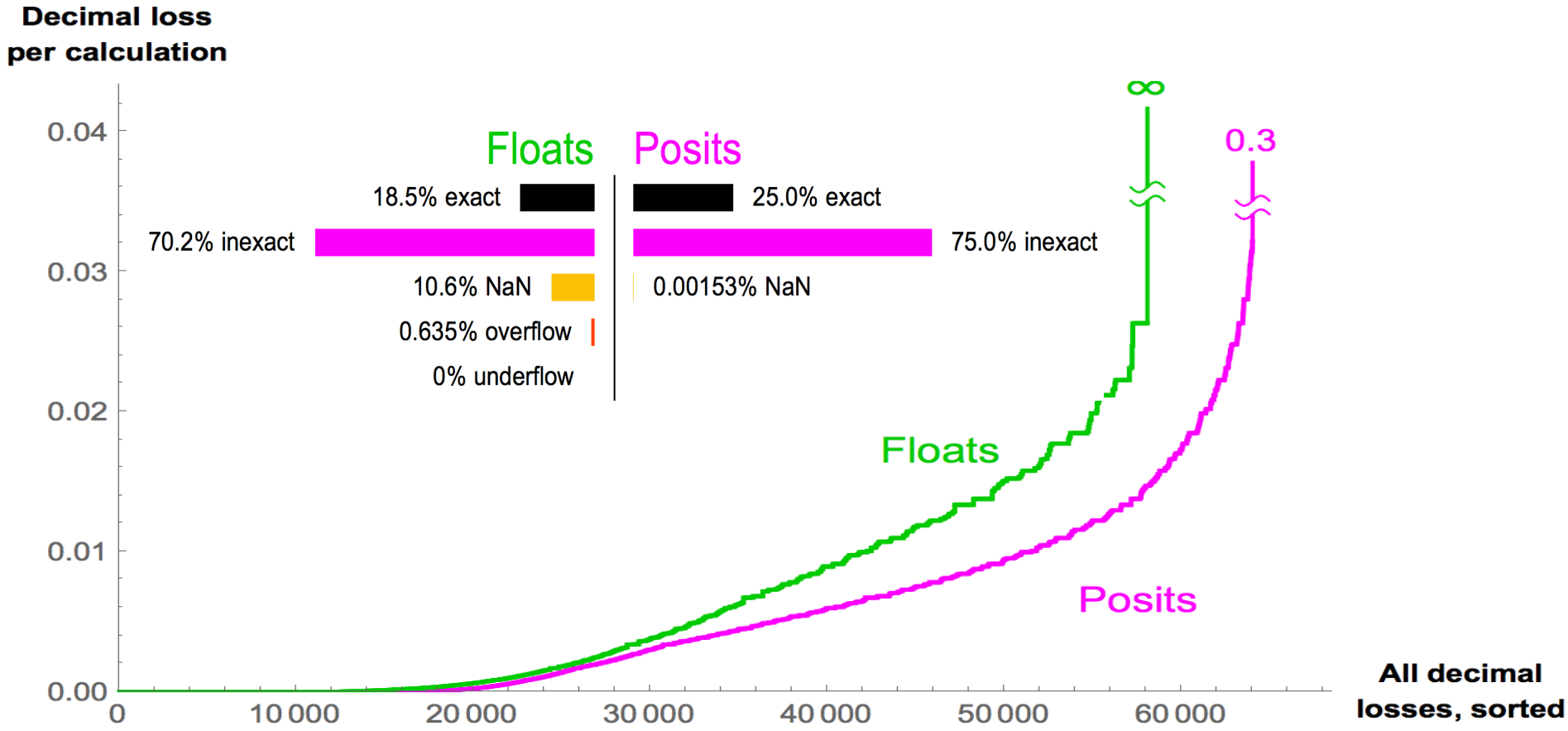
25.005%	exact
74.994%	inexact
0.000%	overflow
0.002%	NaN

Only one case is a NaN:

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

With posits, a NaN *interrupts the calculation*. (Optional mode uses ±∞ as quiet NaN.)

All decimal losses, sorted

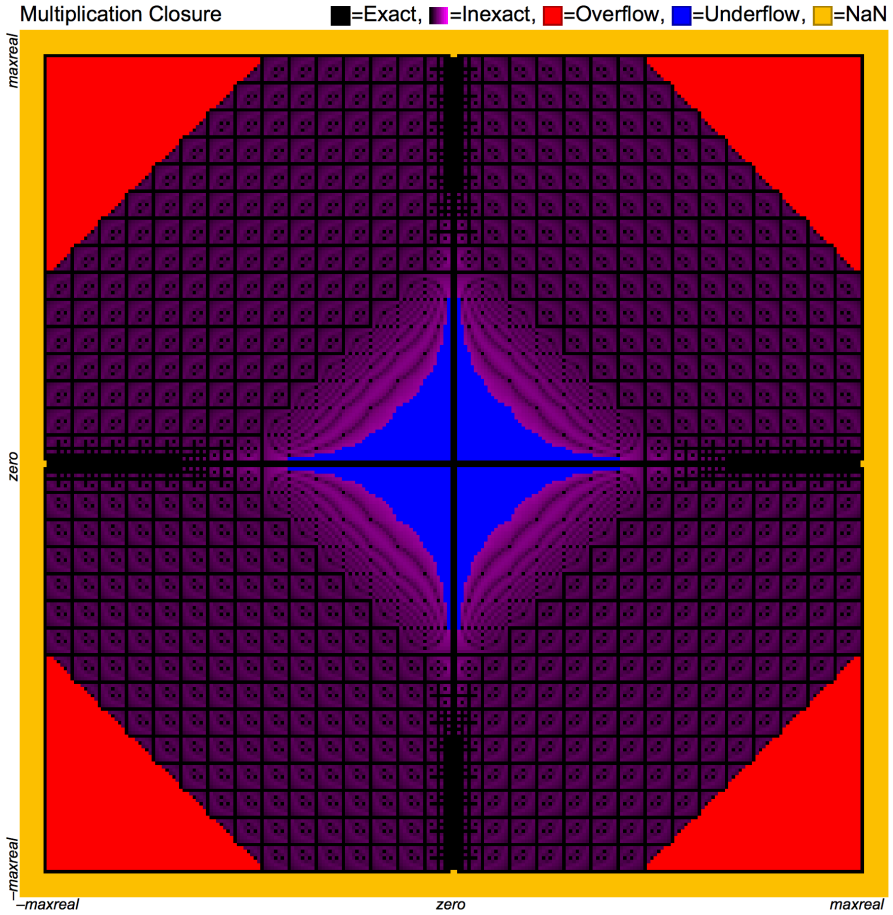


Multiplication Closure Plot: Floats

22.272%	exact
51.233%	inexact
3.345%	underflow
12.500%	overflow
10.651%	NaN

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

but at a **terrible cost**!

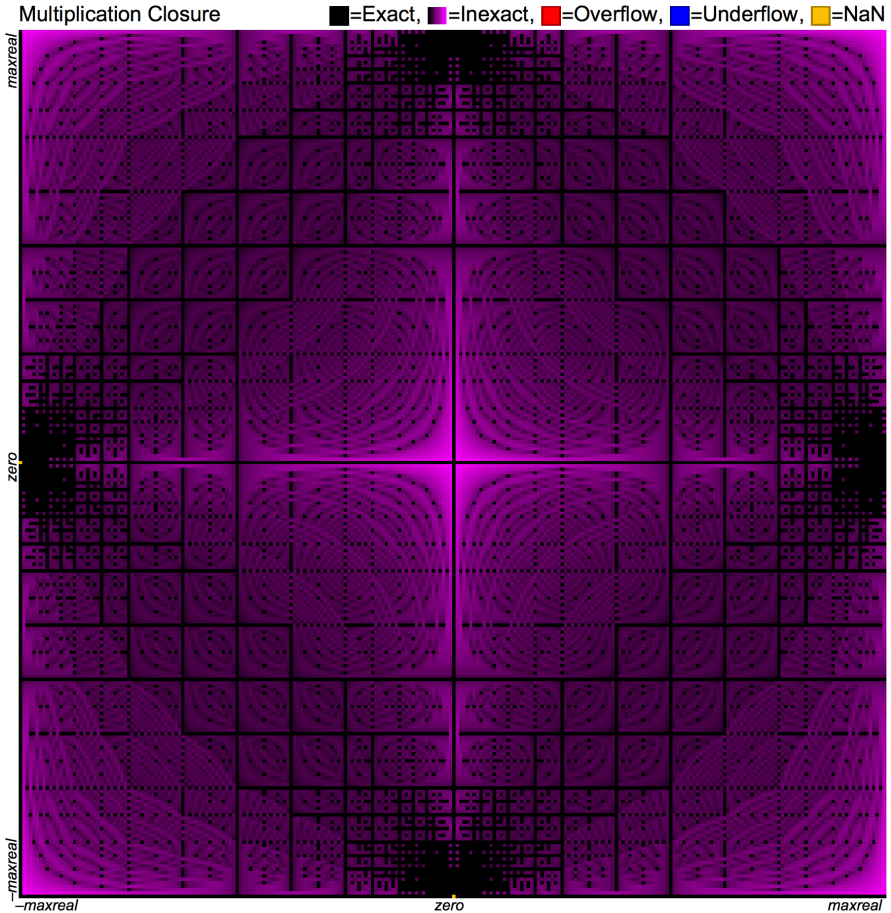


Multiplication Closure Plot: Posits

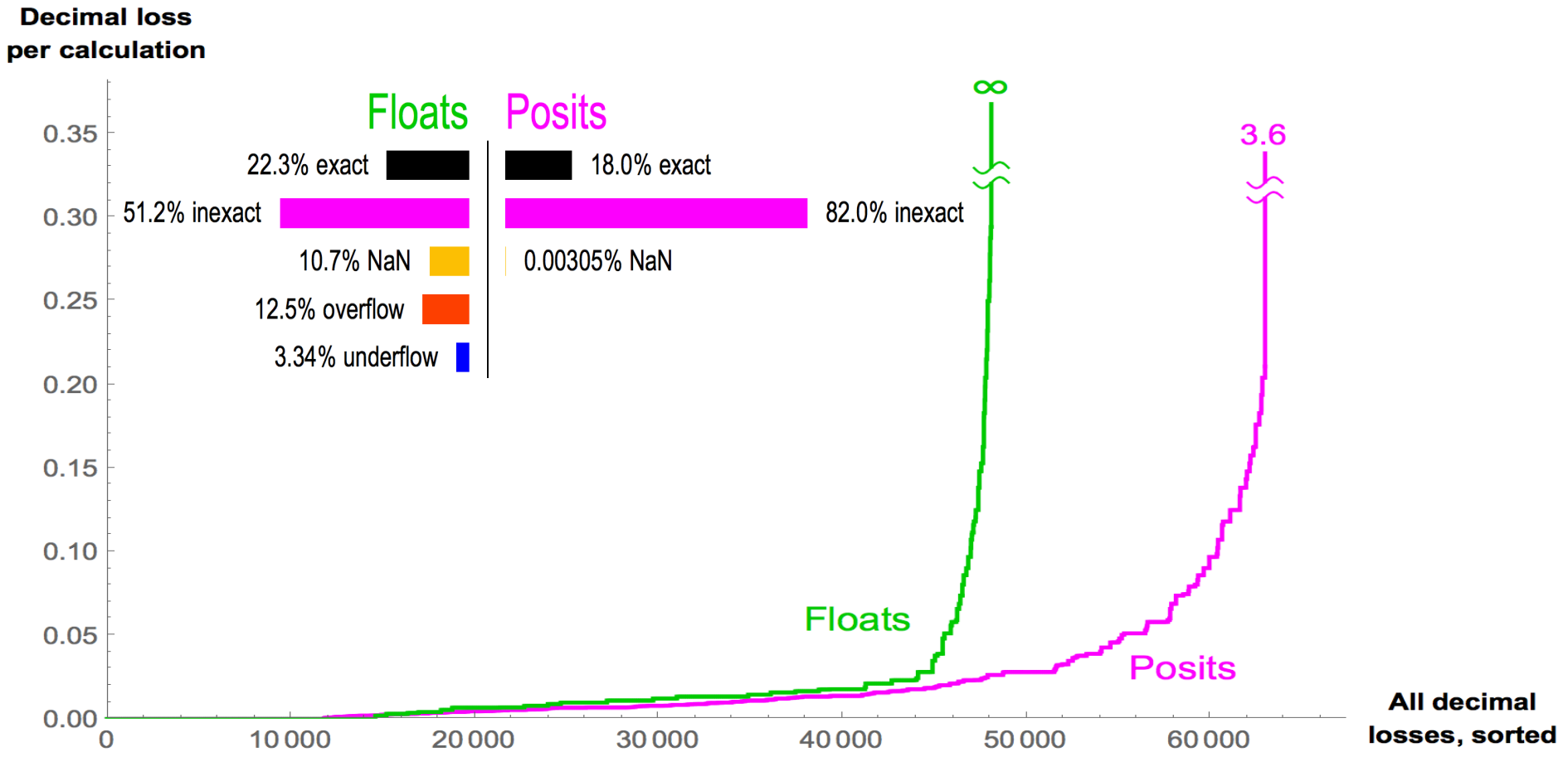
18.002%	exact
81.995%	inexact
0.000%	underflow
0.000%	overflow
0.003%	NaN

Only *two cases* produce a NaN:

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



The sorted losses tell the real story



ROUND 3

Higher-Precision Operations

32-bit formula evaluation
LINPACK solved perfectly with… 16 bits!

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
IEEE 32-bit float	83 decades	302.912···	0.0297···
32-bit posits, no fusing	144 decades	302.8823···	0.00040···
32-bit posits, fused ops	144 decades	302.882713···	0.0000063···

Posits beat floats at both dynamic range and accuracy.

LINPACK: Solve $\mathbf{A}x = b$ *16-bit* posits versus *16-bit* floats

- **A** is a 100 by 100 dense matrix; random $\mathbf{A}_{ij}$ entries in (0, 1)
- *b* chosen so *x* should be all 1s exactly
- Use classic LINPACK method: LU factorization with partial pivoting. Allow refinement using residual.

IEEE 16-bit Floats
Dynamic range: 12 decades
Maximum error: 0.011
Decimal accuracy: 1.96

16-bit Posits
Dynamic range: 16 decades
Maximum error: NONE
Decimal accuracy: ∞

Work funded in part by DARPA under contract BAA 16-39

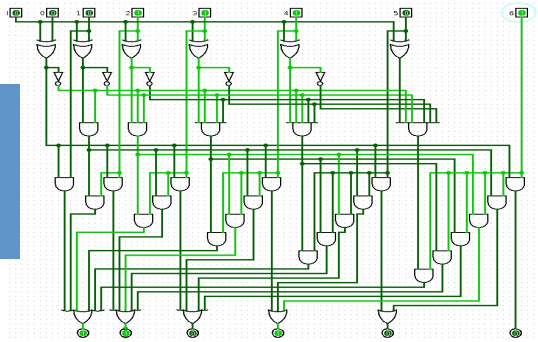
LINPACK: *64-bit* float versus 16-bit posits

64-bit IEEE Floats	16-bit Posits
1.0000000000000124344978758017532527446746826171875	1
0.9999999999999837907438404727145098149776458740234375	1
1.0000000000000193178806284777238033711910247802734375	1
0.99999999999998501198916756038670428097248077392578125	1
0.999999999999991118215802998747676610946655273437 5	1
0.9999999999999990007992778373591136187314987182617187 5	1
⋮	⋮

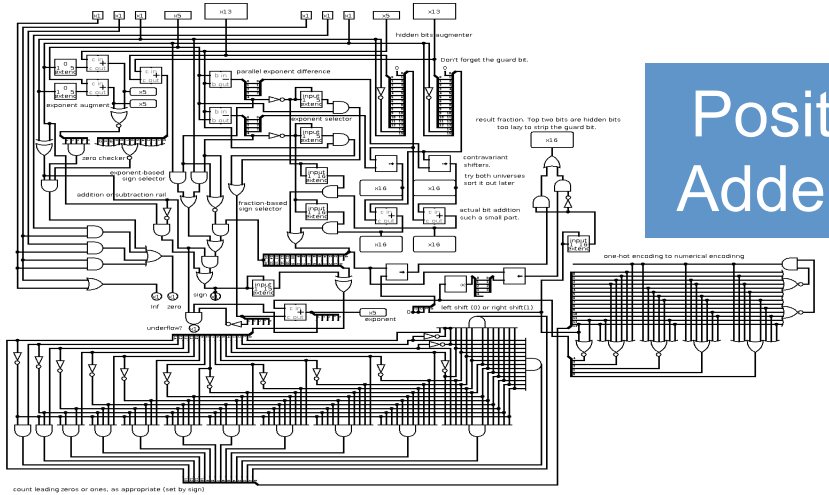
Building posit chips: The race is on

- Like IEEE floats, but *simpler* and *less area* (!)
- REX Computing shipping posit-based multiprocessor to A*STAR by 31 August 2017
- Posit Research Inc. forming to fill out hardware-software-application stack
- Looks ideal for GPUs and Deep Learning; more arithmetic per chip
- Interested companies: Google, IBM, Intel, Samsung, Nvidia, and dozens of others
- LLNL confirmed superior posit performance on their proxy codes, LULESH and Euler2D
- Consortium for Next-Generation Arithmetic is organizing now. Meeting SC'17, SA'18.

Regime Shifter



Posit Adder

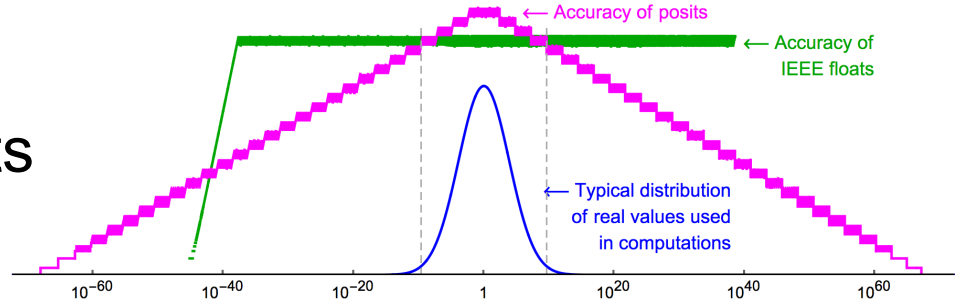


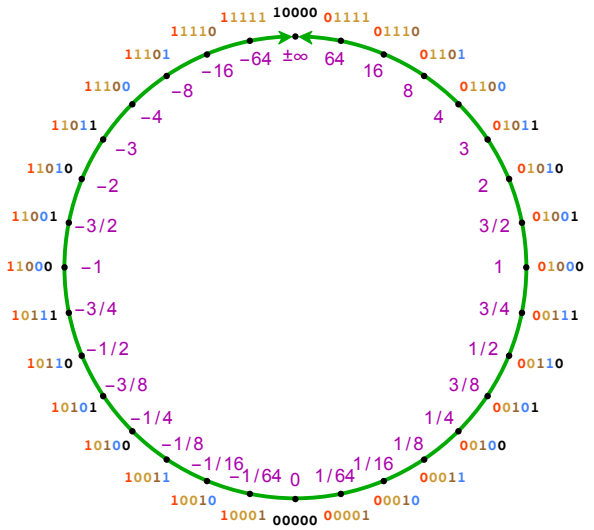
32-bit precision may suffice now!

- Early computers used 36-bit floats.
- IBM System 360 went to 32-bit.
- It wasn't quite enough.
- *What if 32-bit posits could replace 64-bit floats for big data workloads?*
- Potential 2x shortcut to exascale. Or more.

Summary

- Better accuracy with fewer bits
- Consistent, portable results
- Automatic control of rounding errors
- Clean, mathematical design
- Reduces *energy*, *power*, *bandwidth*, *storage*, and *programming costs*
- Potentially halves costs for abundant data challenges





For More Information

http://www.posithub.org

http://www.johngustafson.net/pdfs/BeatingFloatingPoint-superfriversion.pdf

https://www.youtube.com/watch?v=aP0Y1uAA-2Y

https://github.com/interplanetary-robot/SigmoidNumbers