A Radical Approach to Computation with Real Numbers

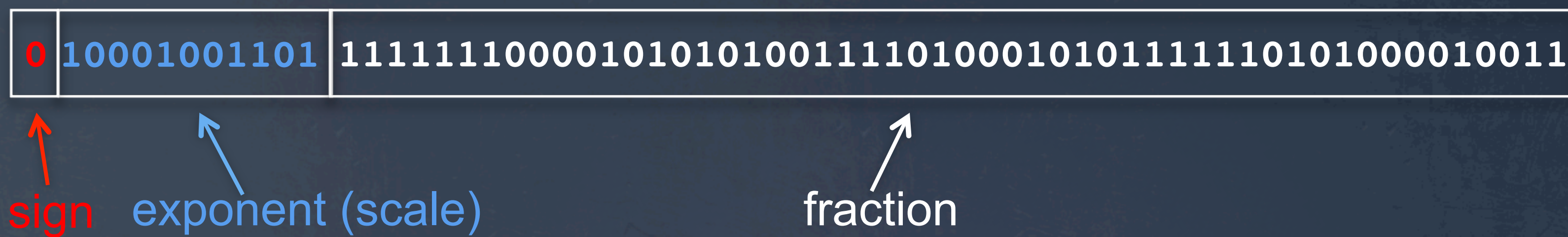
John Gustafson
A*CRC and NUS

“Unums version 2.0”

Updated June 3, 2016. Acknowledgments to Andrew Shewmaker, Alessandro Bartolucci, and William Kahan for many helpful suggestions and corrections

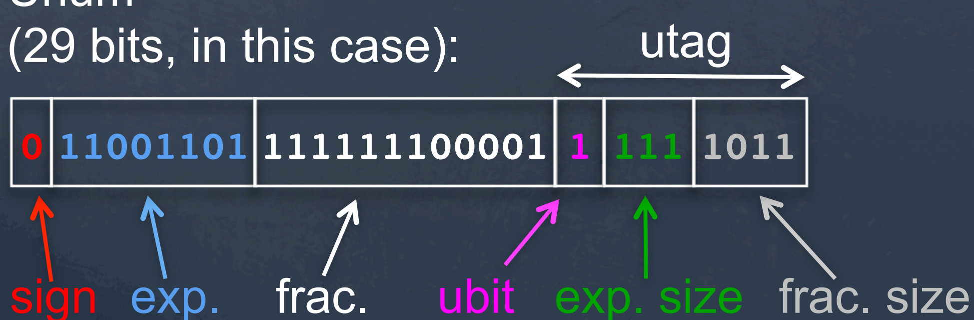
Unums 1.0: upward compatible

IEEE Standard Float (64 bits):



Self-descriptive "utag" bits track and manage uncertainty, exponent size, and fraction size

Unum
(29 bits, in this case):



Flexible dynamic range
Flexible precision
No rounding, overflow, underflow
No "negative zero"
Fixes wasted NaN values
Makes results bit-identical

BUT:
Variable storage size
Adds indirection
Many conditional tests

What would the *ideal* format be?

- All arithmetic operations equally fast
- No penalty for decimal instead of binary
- Easy to build using current chip technology
- No exceptions (subnormals, NaNs, “negative zero”…)
- *One-to-one*: no redundant representations
- *Onto*: No real numbers overlooked
- ~~Upward compatible with IEEE 754~~
- Mathematically sound; no rounding errors

IEEE 754 compatibility prevents all the other goals.

Break *completely* from IEEE 754 floats and gain:

- Computation with mathematical rigor
- Robust set representations with a *fixed* number of bits
- 1-clock binary ops with *no* exception cases
- Tractable "exhaustive search" in high dimensions

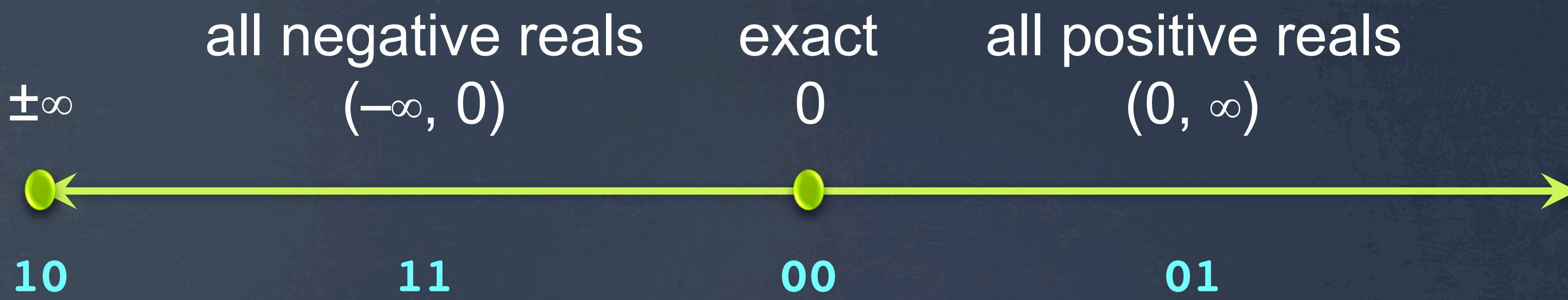
Strategy: Get ultra-low precision right, **then** work up.

All projective reals, using 2 bits

"±∞" is "the point at infinity" and is *unsigned.*

Think of it as the reciprocal of zero.

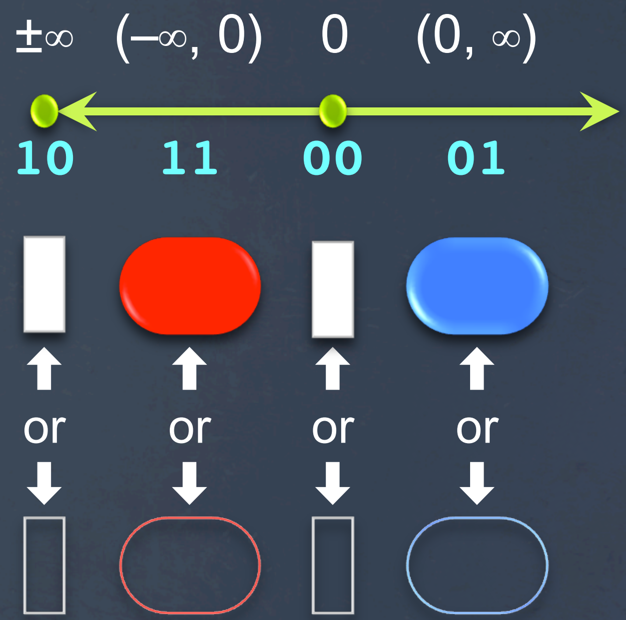
Linear depiction



Maps to the way 2’s complement integers work!

Redundant point at infinity on the right is not shown.

Absence-Presence Bits



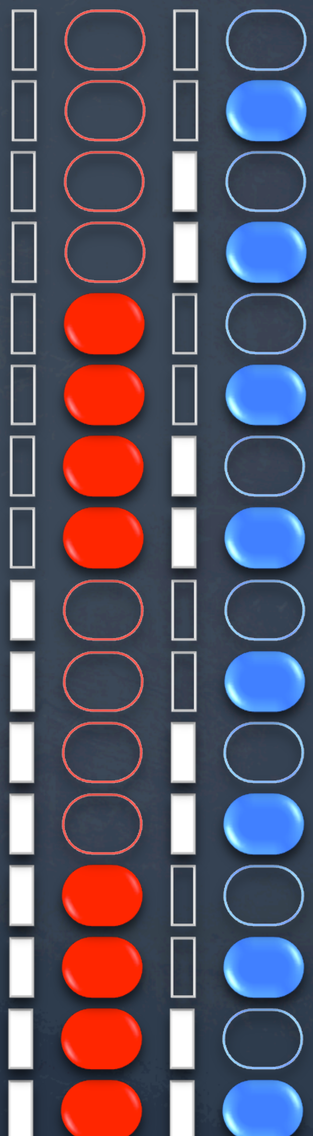
Forms the ***power set*** of the four states.
$2^4 = 16$ possible subsets of the extended reals.

0 (open shape) if absent from the set,
1 (filled shape) if present in the set.

Rectangle if exact, oval or circle if inexact (range)

Red if negative, blue if positive

Sets become *numeric quantities*



- The empty set, { }
- All positive reals (0, ∞)
- Zero, 0
- All nonnegative reals, [0, ∞)
- All negative reals, (–∞, 0)
- All nonzero reals, (–∞, 0) ∪ (0, ∞)
- All nonpositive reals, (–∞, 0]
- All reals, (–∞, ∞)
- The point at infinity, ±∞
- The extended positive reals, (0, ∞]
- The unsigned values, 0 ∪ ±∞
- The extended nonnegative reals, [0, ∞]
- The extended negative reals, [–∞, 0)
- All nonzero extended reals [–∞, 0) ∪ (0, ∞]
- The extended nonpositive reals, [–∞, 0]
- All extended reals, [–∞, ∞]

“SORNs”: Sets Of Real Numbers

Closed under

$x + y \quad x - y$

$x \times y \quad x \div y$

and…x^y

Tolerates division by 0.

No indeterminate forms.

Very different from *symbolic* ways of dealing with sets.

No more "Not a Number"

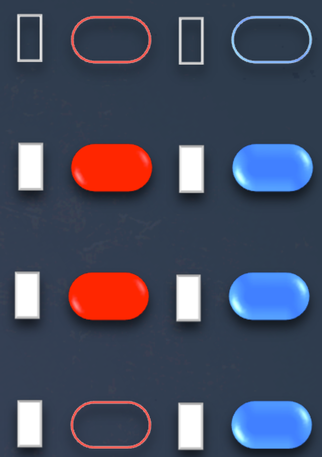
$\sqrt{-1}$ = empty set:

0 / 0 = everything:

$\infty - \infty$ = everything:

1^{∞} = all nonnegatives, $[0, \infty]$:

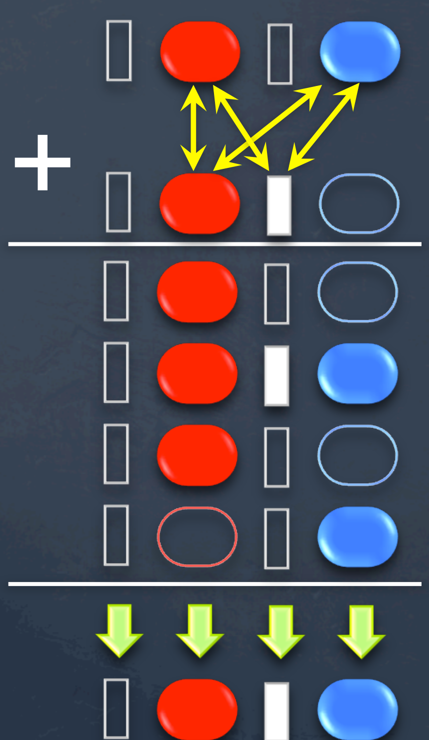
etc.



Answers, as limit forms, are *sets*.
We can express those!

Op tables need only be 4x4

For any SORN, do table look-up for pairwise bits that are set, and find the union with a bitwise OR.



parallel OR

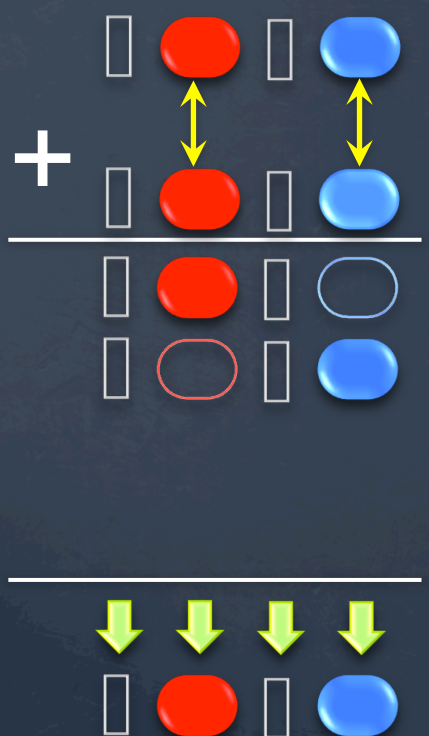
Hardware flag: independent *x* and *y*

Note that three entries “blur”, indicating *information loss.*

Compiler-Hardware Interaction

If a variable occurs more than once, only *reflexive* combinations are needed.

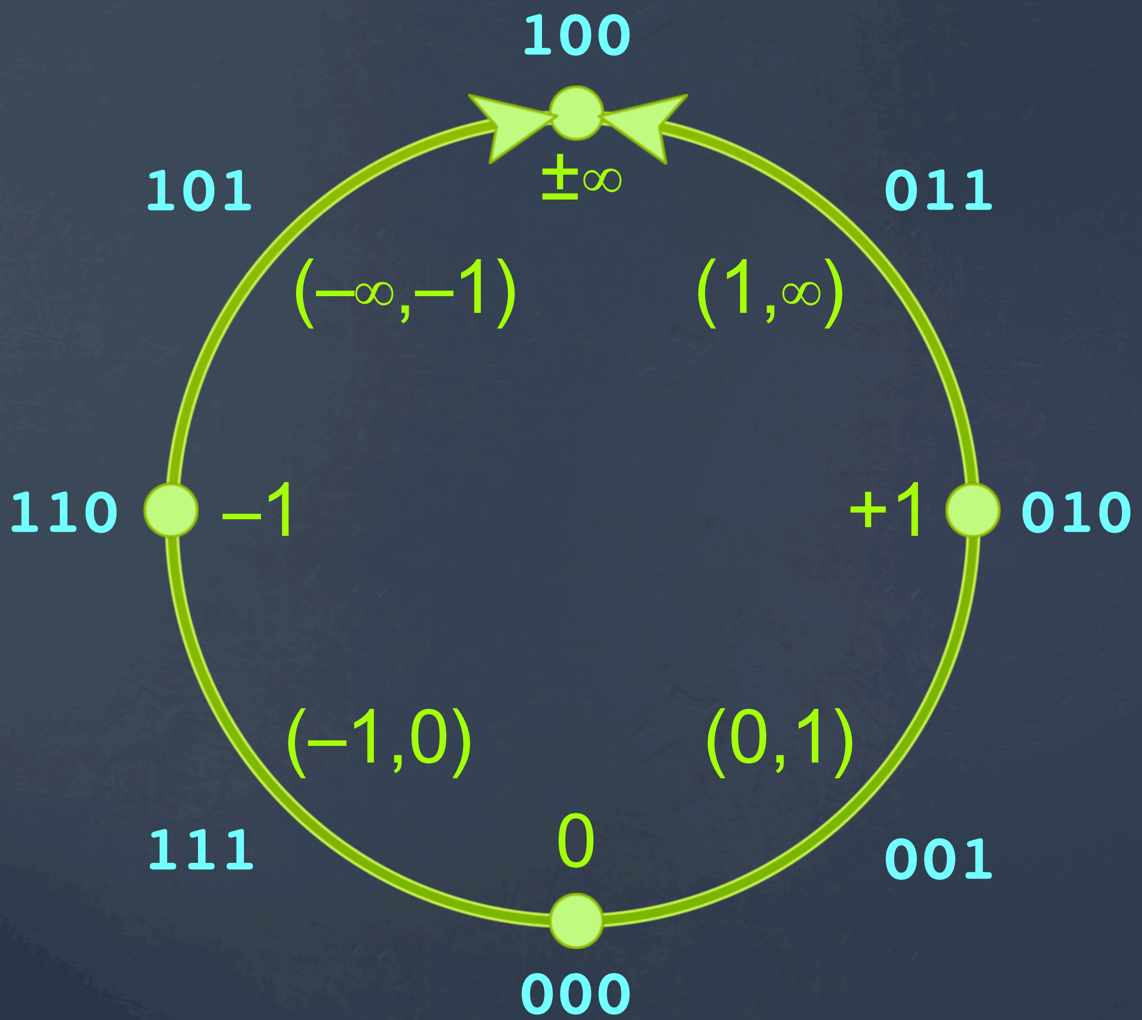
Hardware flag: *dependent* *x* and *y*. (*y* = *x*)



parallel OR

Compiler detects common sub-expressions, so $x + x$ is handled differently from $x + y$

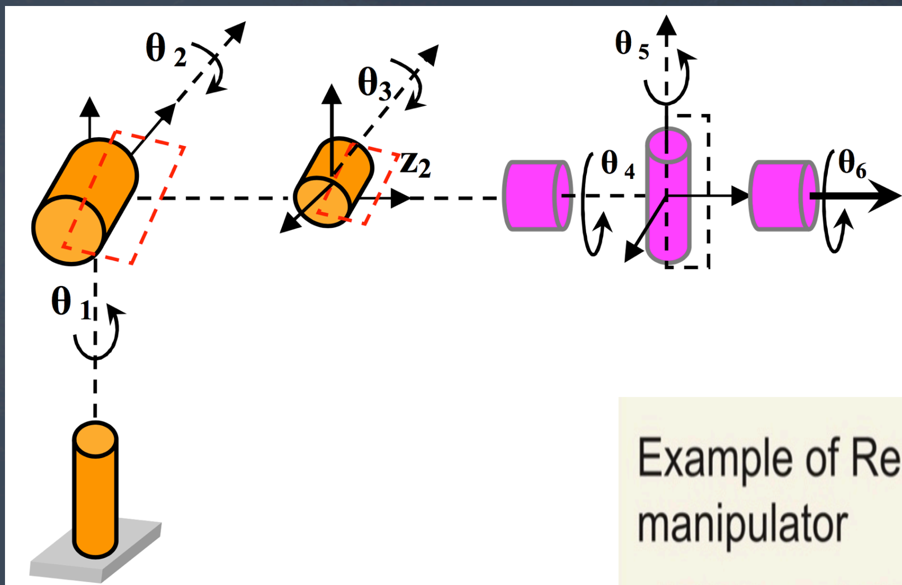
Now include +1 and –1



The SORN is 8 bits long.

This is actually enough of a number system to be useful!

Example: Robotic Arm Kinematics



12-dimensional nonlinear system (!)

Example of Real Constraints: inverse kinematics of an elbow manipulator

$$s_2c_5s_6 - s_3c_5s_6 - s_4c_5s_6 + c_2c_6 + c_3c_6 + c_4c_6 = 0.4077;$$
$$c_1c_2s_5 + c_1c_3s_5 + c_1c_4s_5 + s_1c_5 = 1.9115;$$
$$s_2s_5 + s_3s_5 + s_4s_5 = 1.9791;$$
$$c_1c_2 + c_1c_3 + c_1c_4 + c_1c_2 + c_1c_3 + c_1c_2 = 4.0616;$$
$$s_1c_2 + s_1c_3 + s_1c_4 + s_1c_2 + s_1c_3 + s_1c_2 = 1.7172;$$
$$s_2 + s_3 + s_4 + s_2 + s_3 + s_2 = 3.9701;$$
$$s_i^2 + c_i^2 = 1 \quad (1 \leq i \leq 6)$$

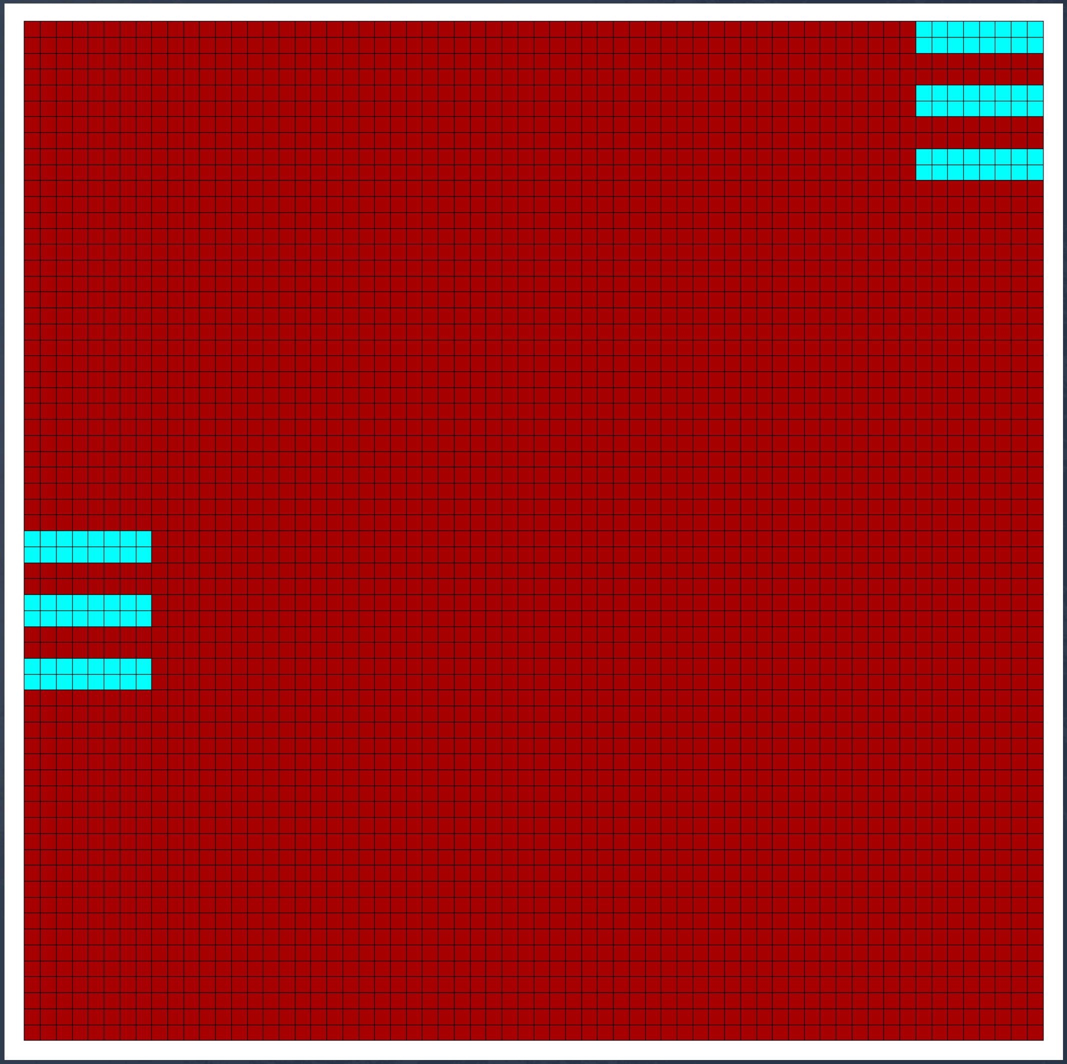
Notice all values must be in [–1,1] ➔

"Try everything"… in 12 dimensions

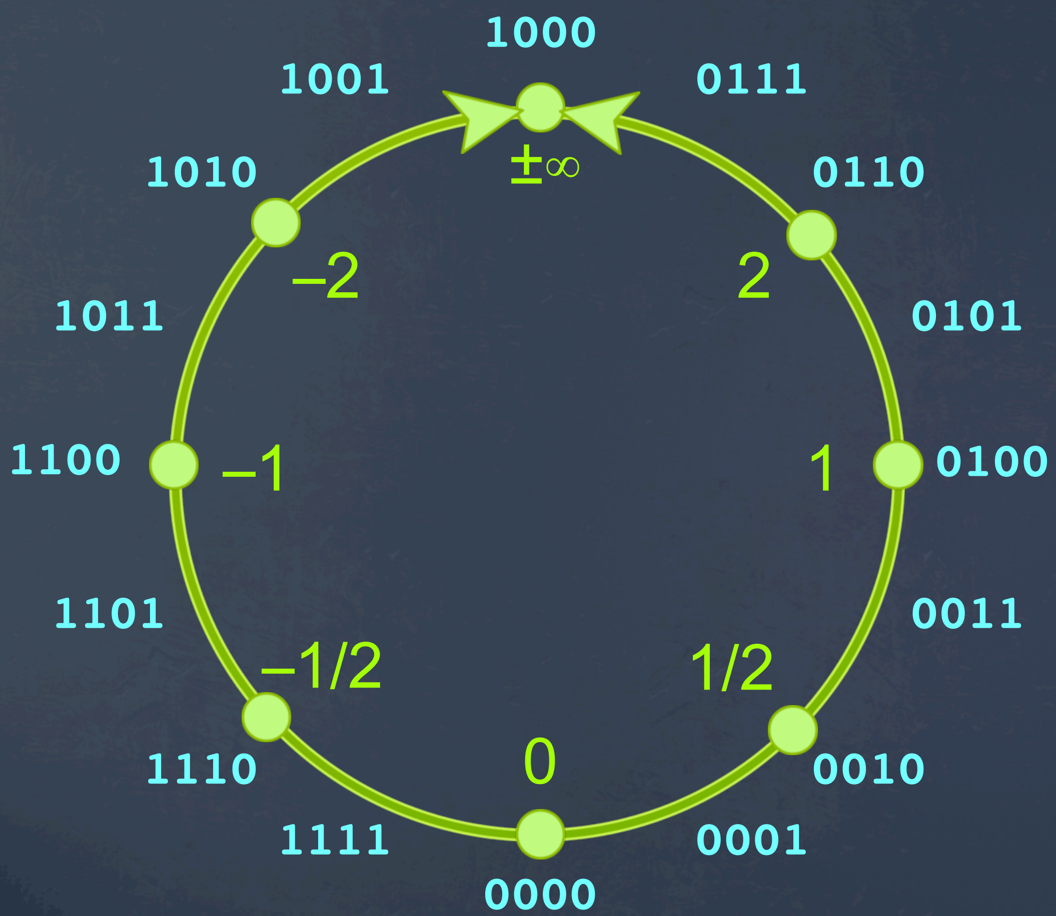
Every variable is in [–1, 1], so split into [–1, 0) and [0, 1] and compute the constraint function to 3-bit accuracy.

■ = violates constraints
■ = compliant subset

2^{12} = 4096 sub-cubes can be evaluated in parallel, in a few *nanoseconds*.



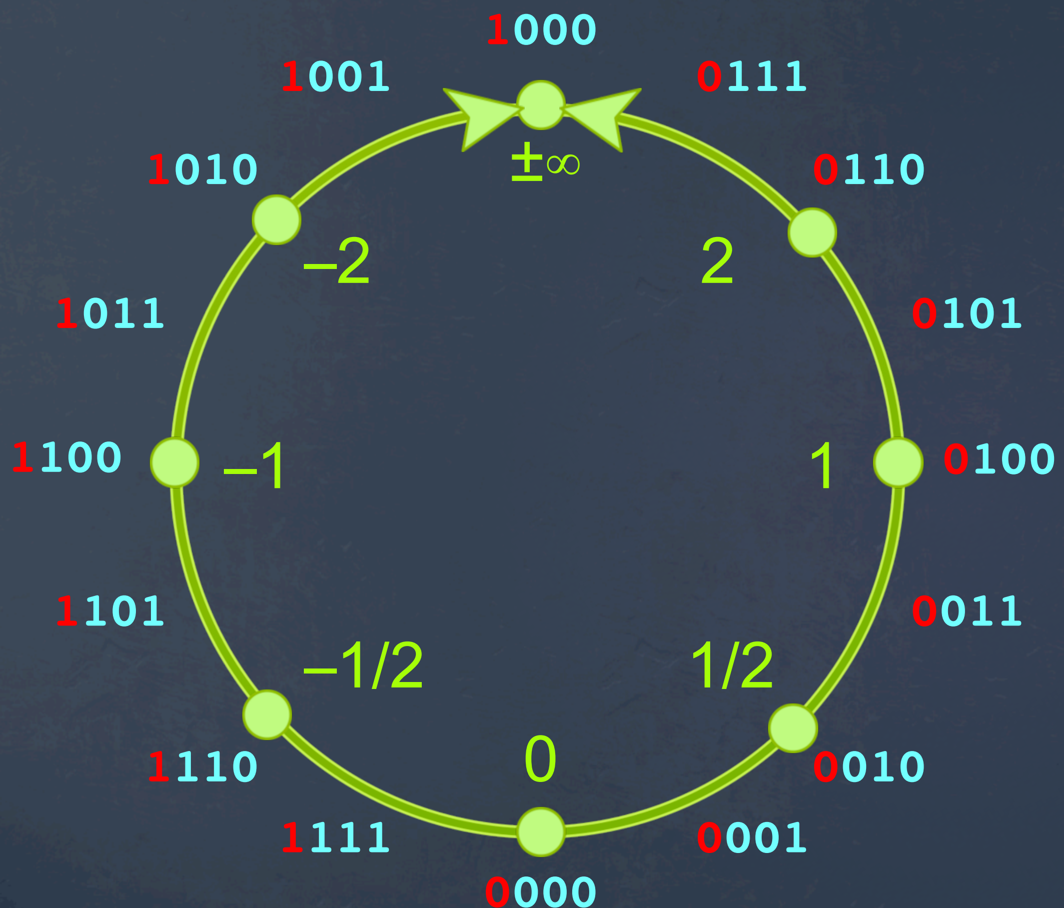
One option: more powers of 2



There is nothing special about 2. We could have added 10 and 1/10, or even *π* and 1/*π*, or *any exact number*.

(Yes, *π* can be numerically exact, if we want it to be!)

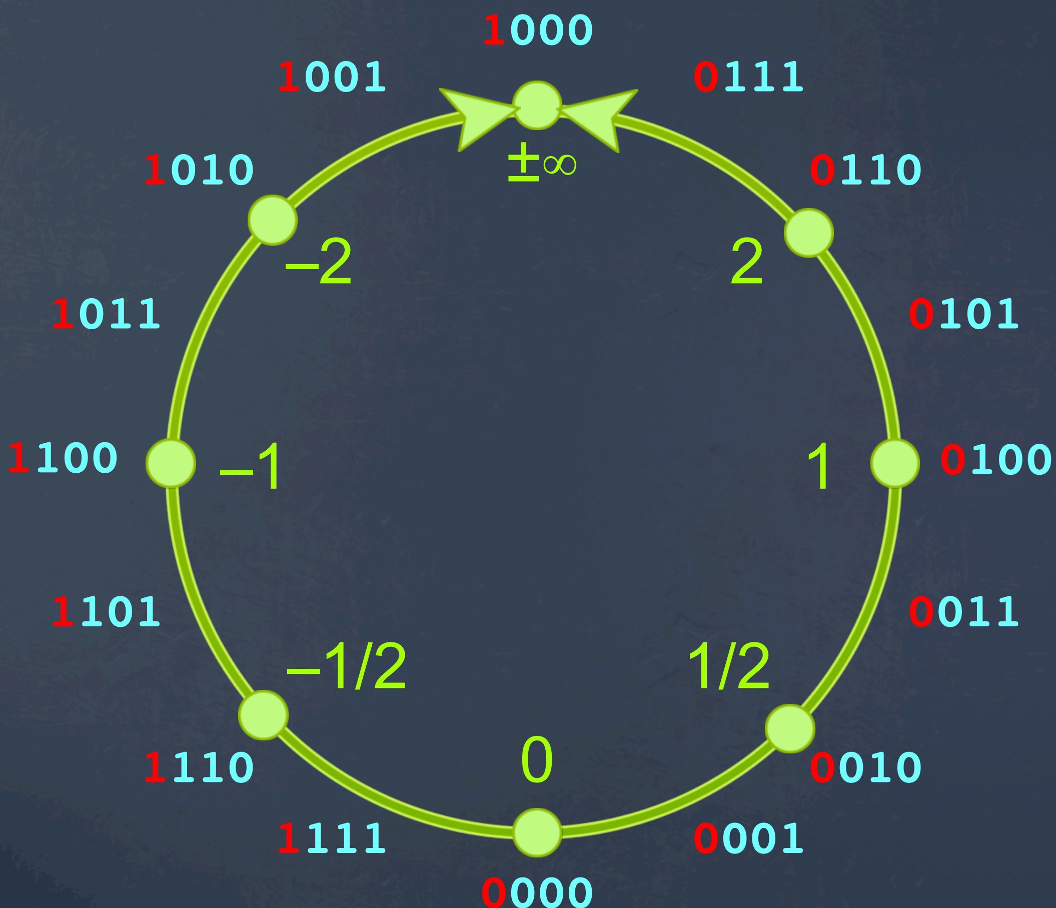
Note: sign bit is in the usual place



The sign of 0 and ±∞ is meaningless, since

0 = –0 and
±∞ = –±∞.

Negation is trivial



To negate, flip horizontally.

Reminder: In 2's complement, flip all bits and add 1, to negate. *Works without exception, even for* 0 *and* ±∞. (They do not change.)

A new notation: Unary “/”

Just as unary “–” can be put before *x* to mean 0 – *x*,
unary “/” can be put before *x* to mean 1/*x*.

Just as we can write –*x* for 0 – *x*, we can write /*x* for 1/*x*.
Pronounce it “over x”

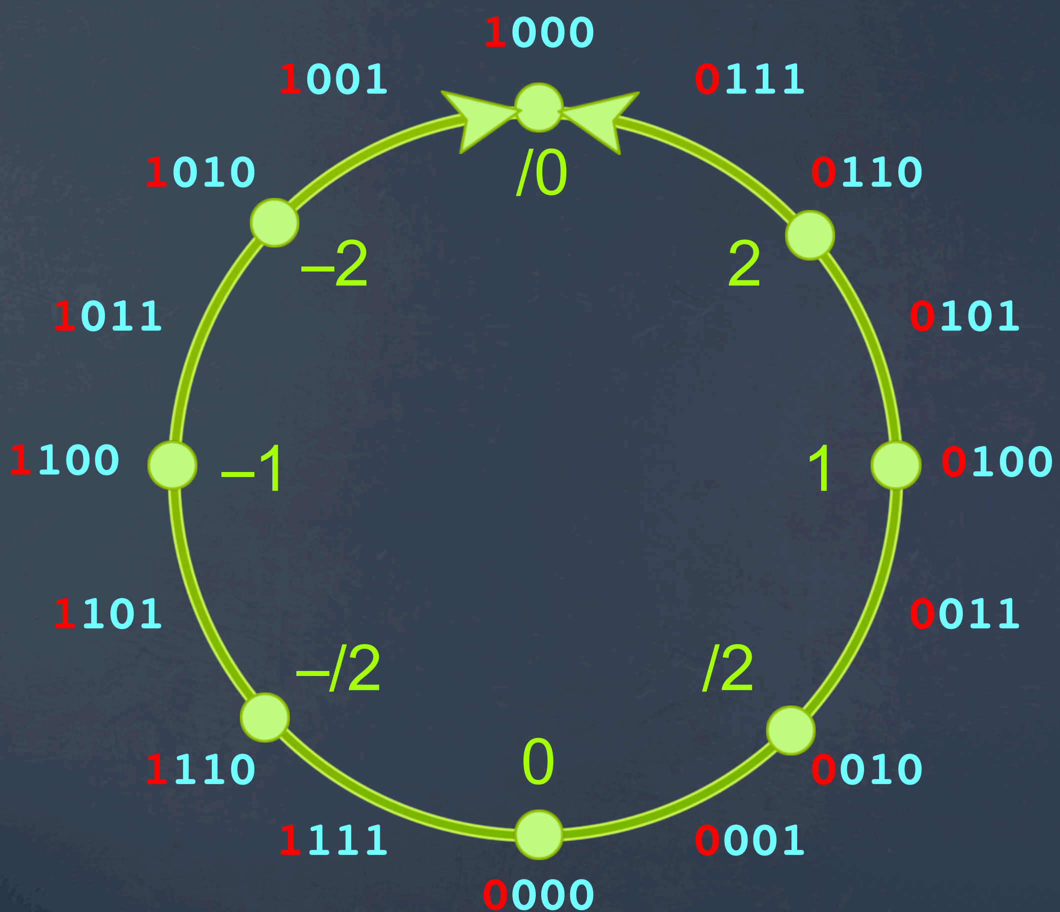
Parsing is just like parsing unary minus signs.

$-(-x) = x$, just as $/(/x) = x$.
$x - y = x + (-y)$, just as $x \div y = x \times (/y)$

These unum systems are lossless (no rounding error) under negation ***and*** reciprocation.

Arithmetic ops + – × ÷ are on **equal footing**.

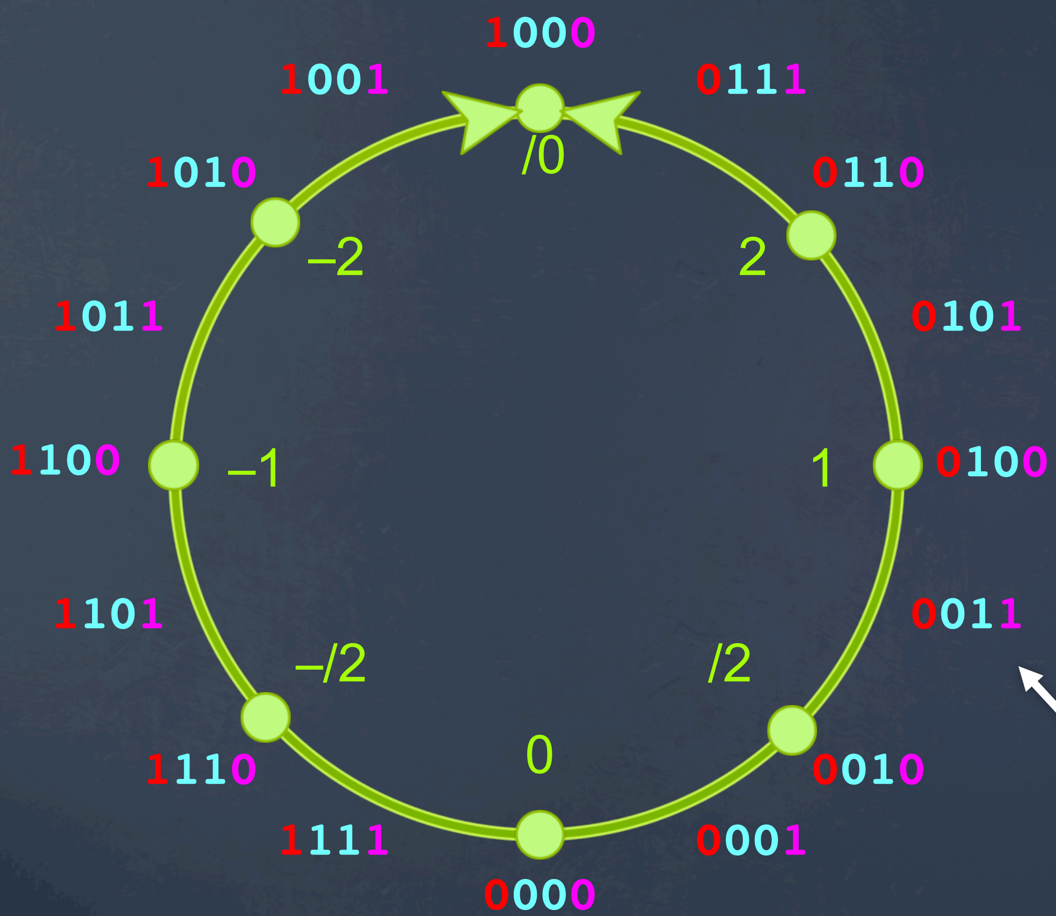
Reciprocation is trivial, too!



To reciprocate, flip *vertically*.

Reverse all bits but the first one and add **1**, to reciprocate. *Works without exception*. +1 and –1 do not change.

The last bit serves as the *ubit*



ubit = **0** means exact

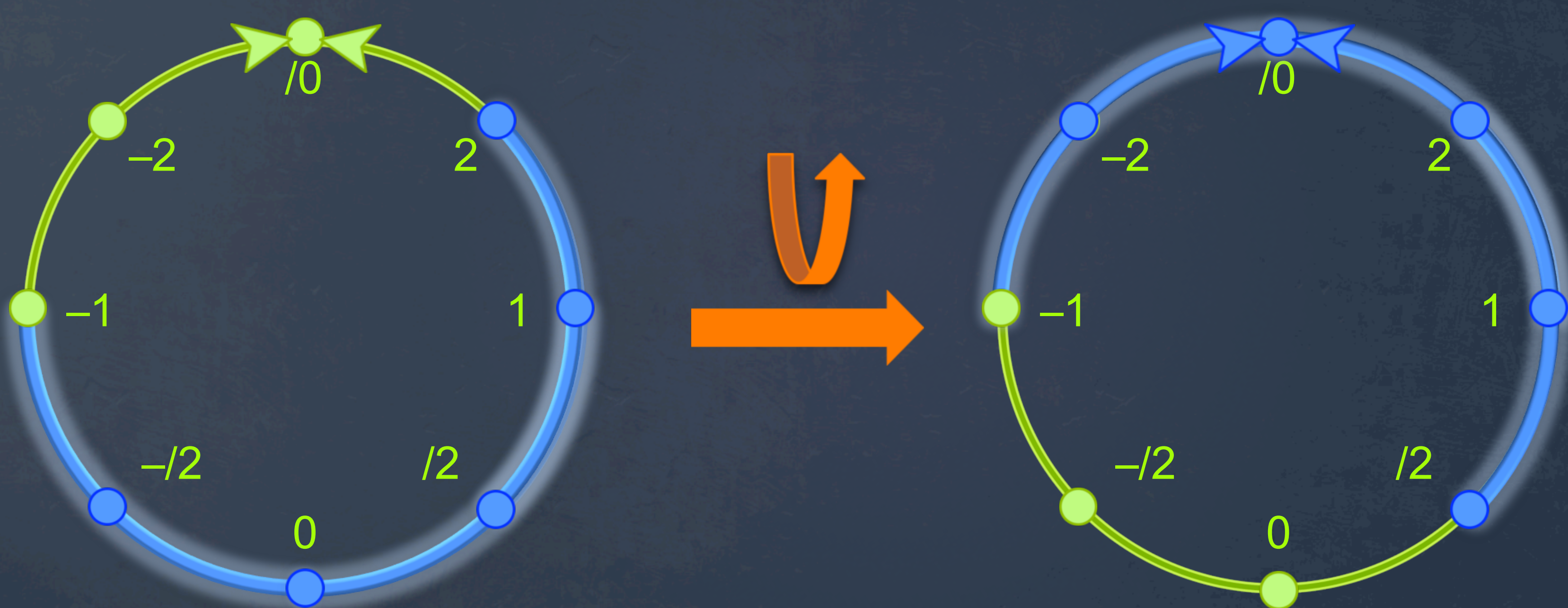
ubit = **1** means *the open interval between exact numbers*.

"uncertainty bit".

Example: This means the open interval (½, 1). Or (get used to it), (/2, 1).

Divide by 0 mid-calculation and still get the *right answer*

What is 1 / (1/*x* + ½) for –1 < *x* ≤ 2?



10-unum SORN
for *x* = (–1, 2]

lossless SORN
for 1/*x* = [½, –1)

Divide by zero is an ordinary operation.

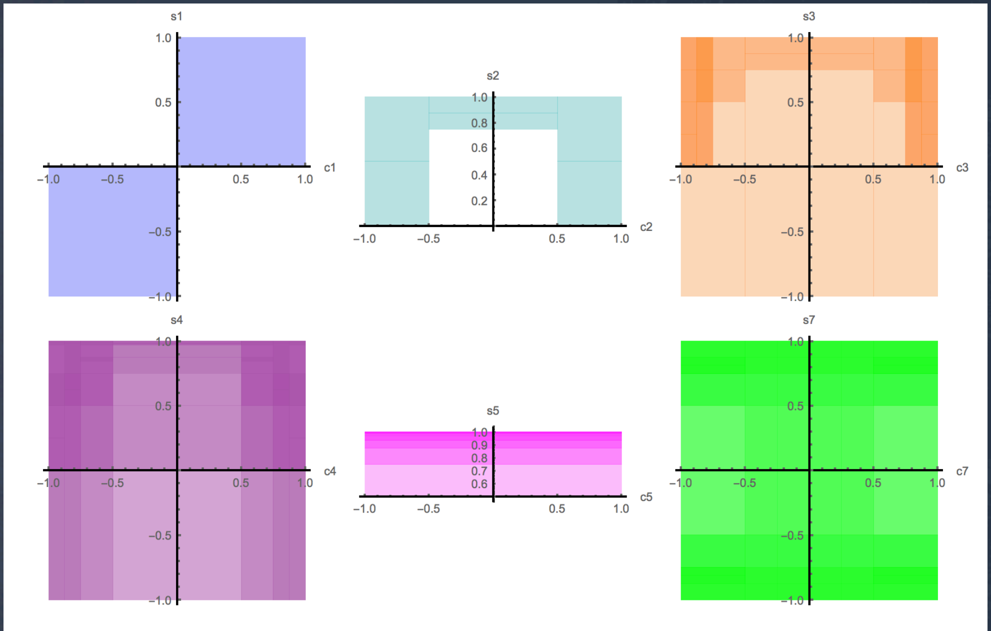
Add ½, reciprocate again

Back to kinematics, with exact 2^k

Split one dimension at a time. Needs only 1600 function evaluations (microseconds).

Display six 2D graphs of *c* versus *s* (cosine versus sine… should converge to an arc)

Here is what the *rigorous bound* looks like after one pass.

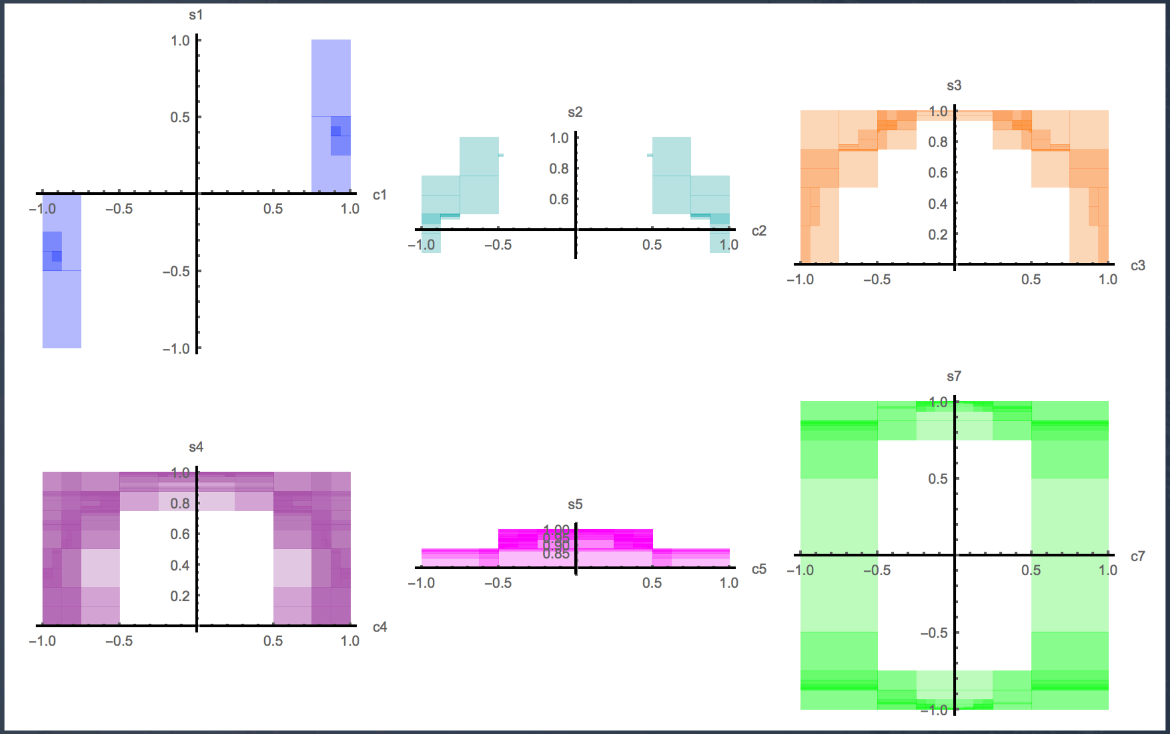


Information = /uncertainty.

Uncertainty = answer volume.

Information increases by 1661×

Make a second pass



Still using ultra-low precision

Starting to look like arcs (angle ranges)

457306 function evaluations (μsecs,using parallelism)

Information increases by a factor of 3.7×10^6

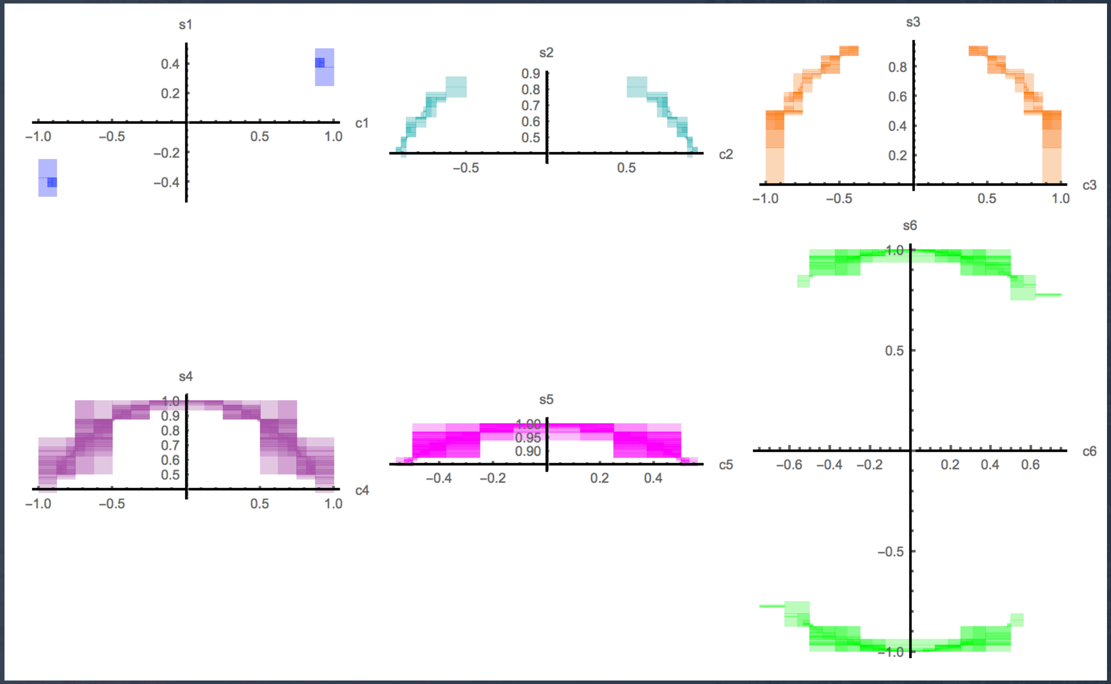
A third pass allows robot decision

Transparency helps show 12 dimensions, 2 at a time.

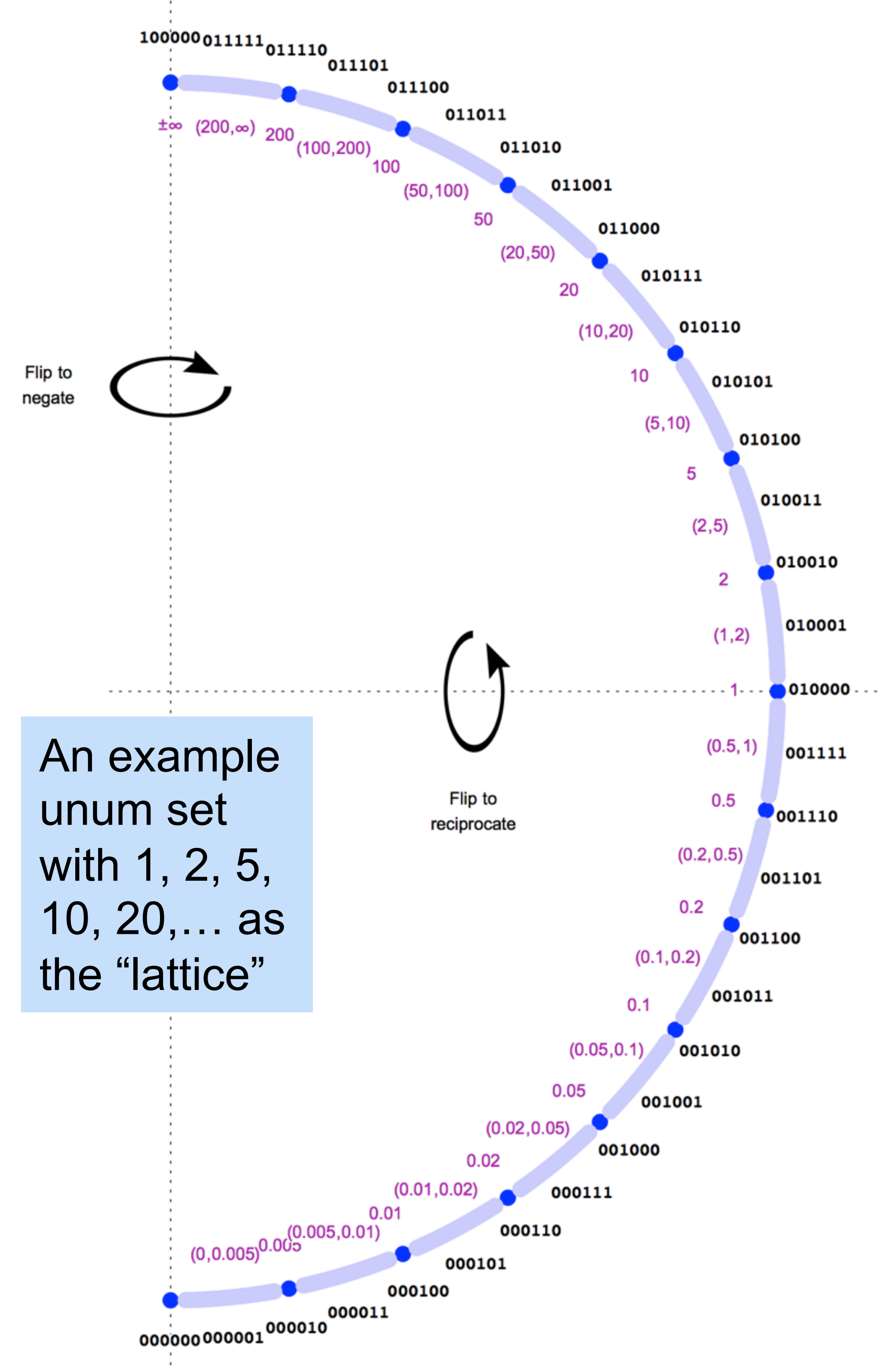
Starting to look like arcs (angle ranges).

6 million function evaluations (a few msec)

Information increases by a factor of 1.8×10^{11}



Remember, this is a **rigorous bound** of all possible solutions. Gradient-type searching with floats can only ***guess***.



An example unum set with 1, 2, 5, 10, 20,… as the “lattice”

Unums 2.0

Still Universal Numbers. They are like the original unums, but:

- Fixed size
- *Not* an extension of IEEE floats
- ULP size variance becomes *sets*
- No redundant representations
- No wasted bit patterns
- No NaN exceptions
- No penalty for using decimals!
- No errors in converting human-readable format to and from machine-readable format.

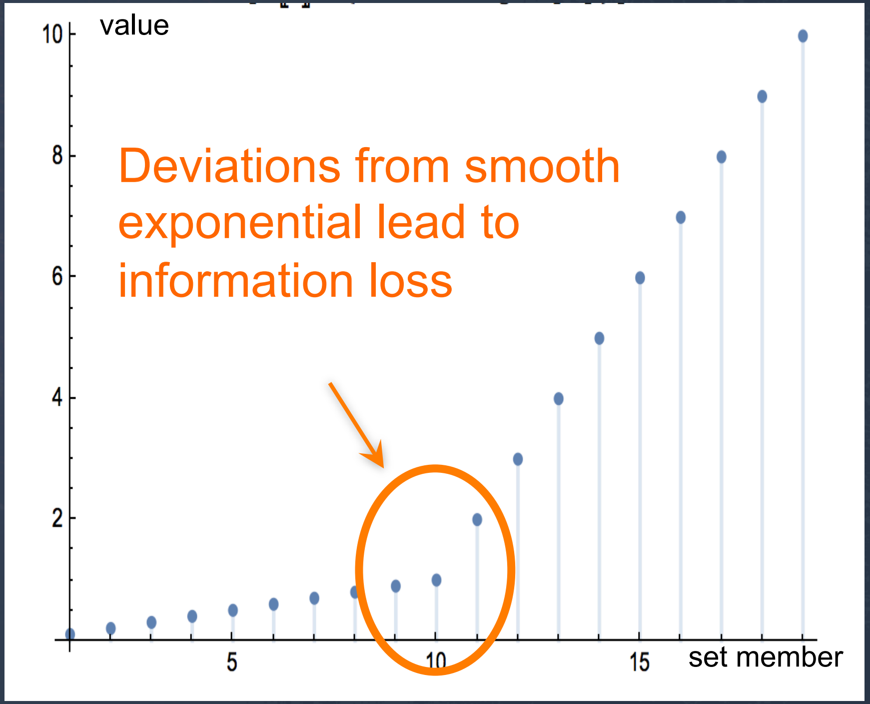
Time to get serious

What is the best possible use of an *8-bit byte* for real-valued calculations?

Start with kindergarten numbers:
1, 2, 3, 4, 5, 6, 7, 8, 9, 10.

Divide by 10 to center the set about 1:
0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9,
1, 2, 3, 4, 5, 6, 7, 8, 9, 10

This has the classic problem with decimal IEEE floats: "***wobbling precision***."



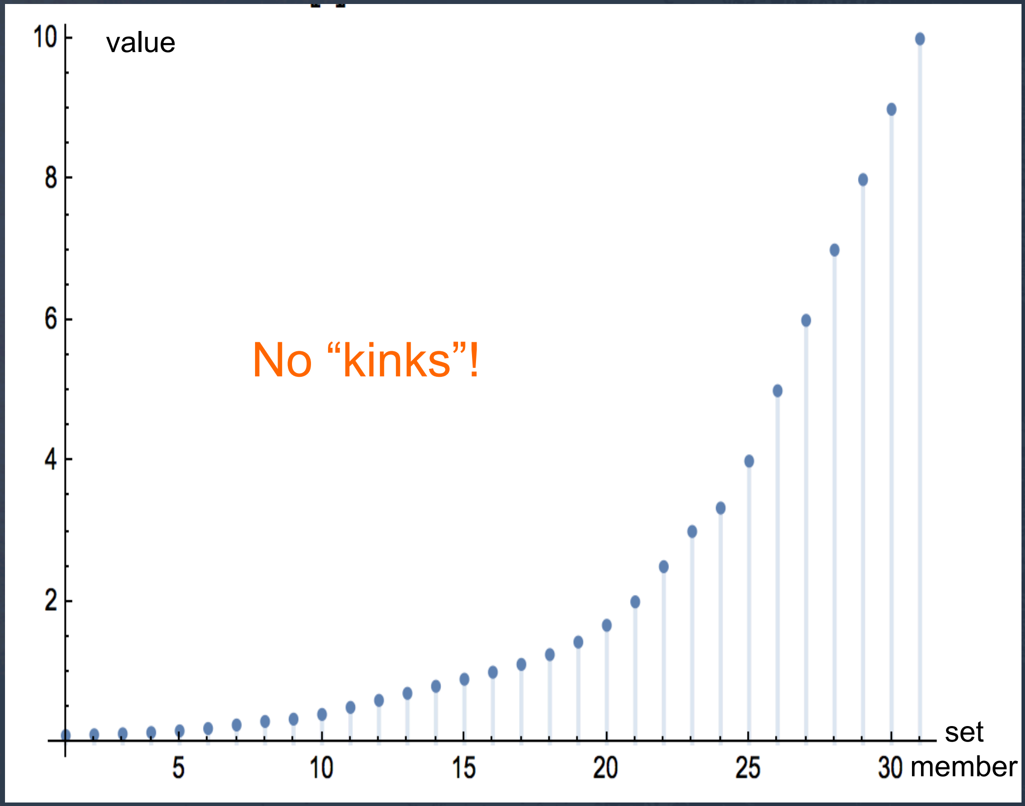
Reciprocal closure cures wobbling precision

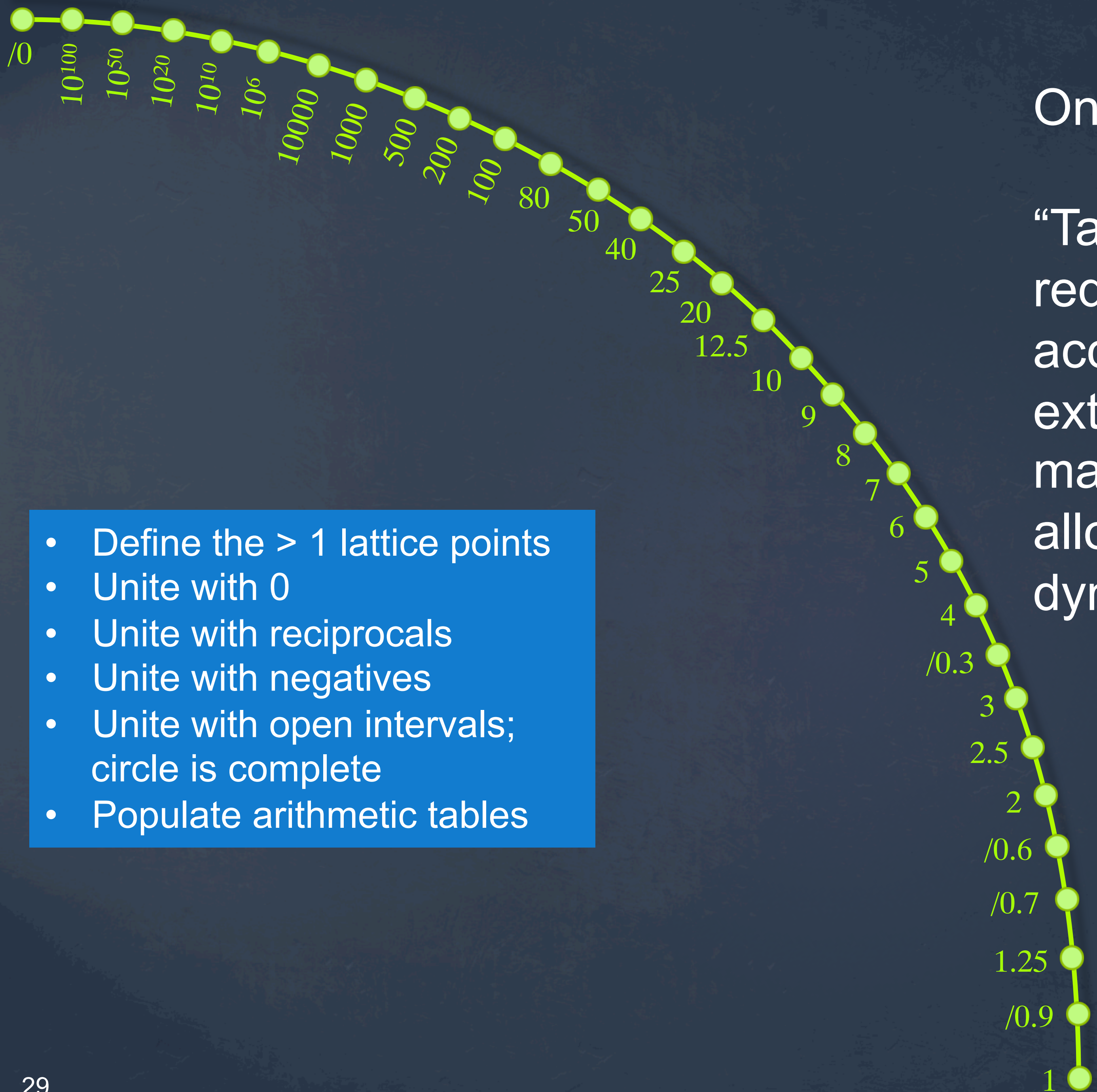
Unite set with the reciprocals of the values, guaranteeing closure:

0.1, /9, 0.125, /7, /6, 0.2, 0.25,
0.3, /3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9,

1, /0.9, 1.25, /0.7, /0.6, 2, 2.5,
3, /0.3, 4, 5, 6, 7, 8, 9, 10

That's 30 numbers. Room for 33 more.

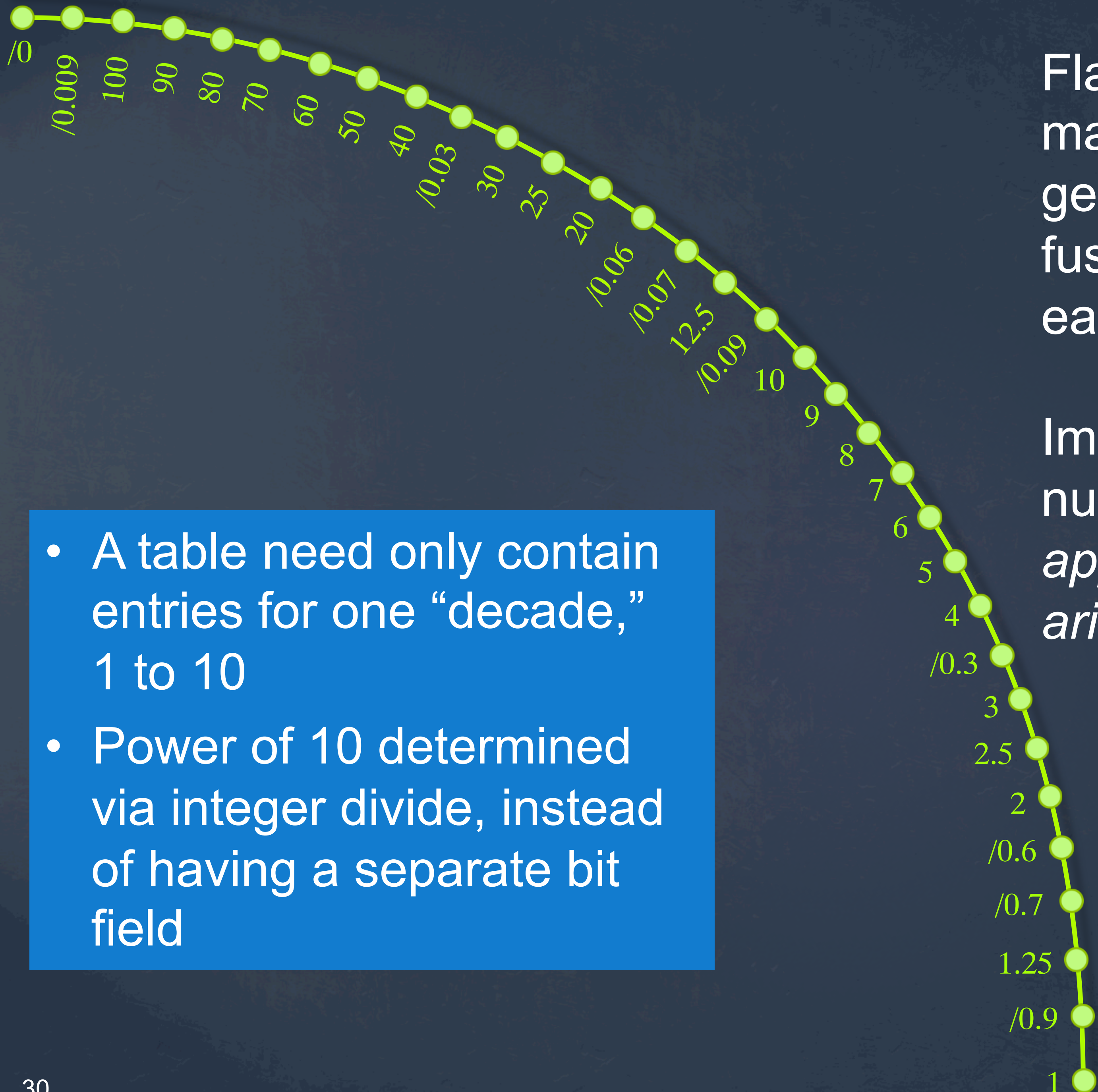




One approach:

"Tapered Precision" reduces relative accuracy for extreme magnitudes, allowing very large dynamic range.

- Define the > 1 lattice points
- Unite with 0
- Unite with reciprocals
- Unite with negatives
- Unite with open intervals; circle is complete
- Populate arithmetic tables



Flat precision makes table generation and fused operations easier.

Imagine: custom number systems for *application-specific arithmetic*

- A table need only contain entries for one "decade," 1 to 10
- Power of 10 determined via integer divide, instead of having a separate bit field

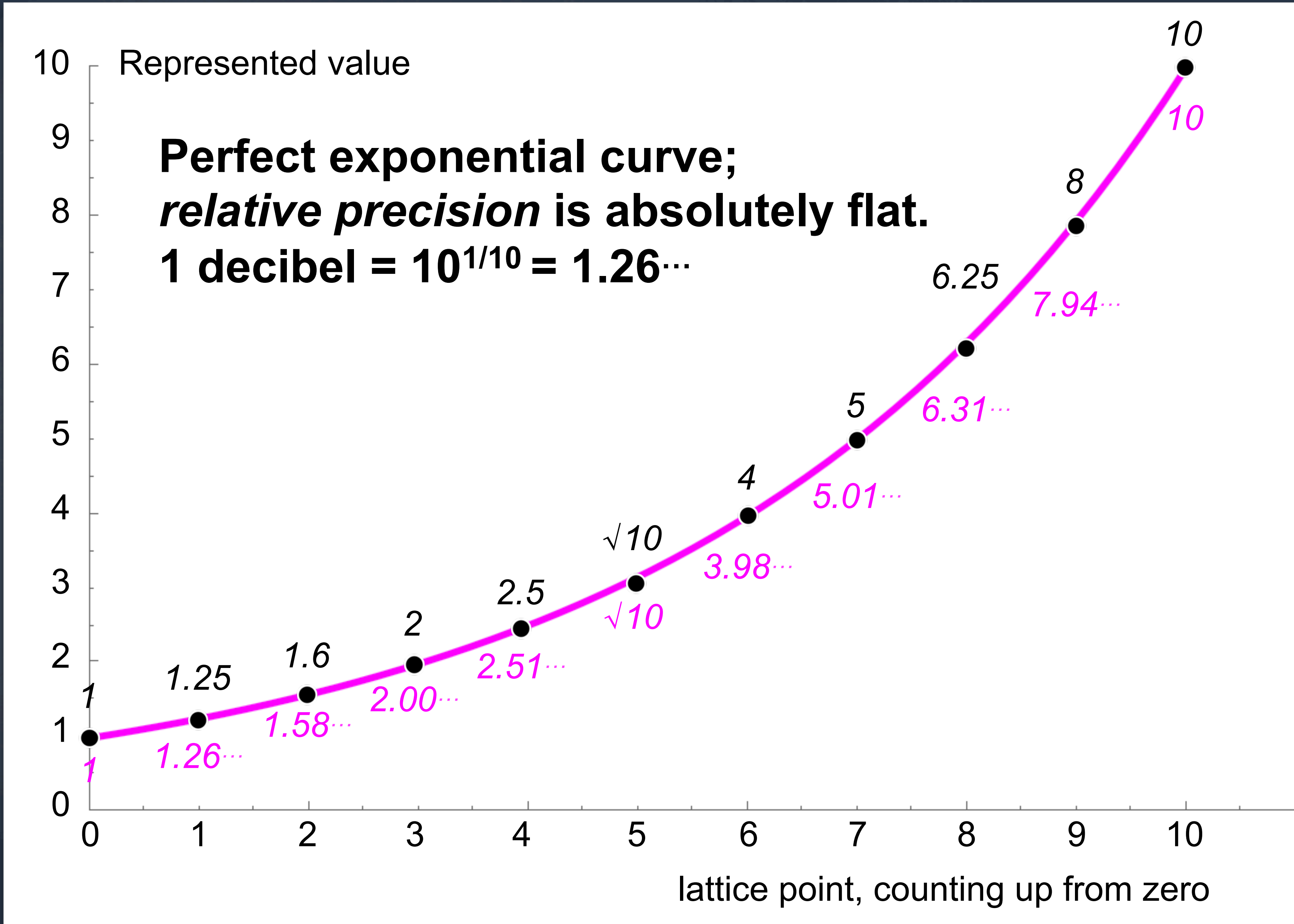
A very cool coincidence

Low powers of two: 1, 2, 4, 8, 16.
Low powers of five: 1, 5, 25, 125, 625.
Throw in the square root of ten, $\sqrt{10}$.
Scale to be between 1 and 10, and sort:

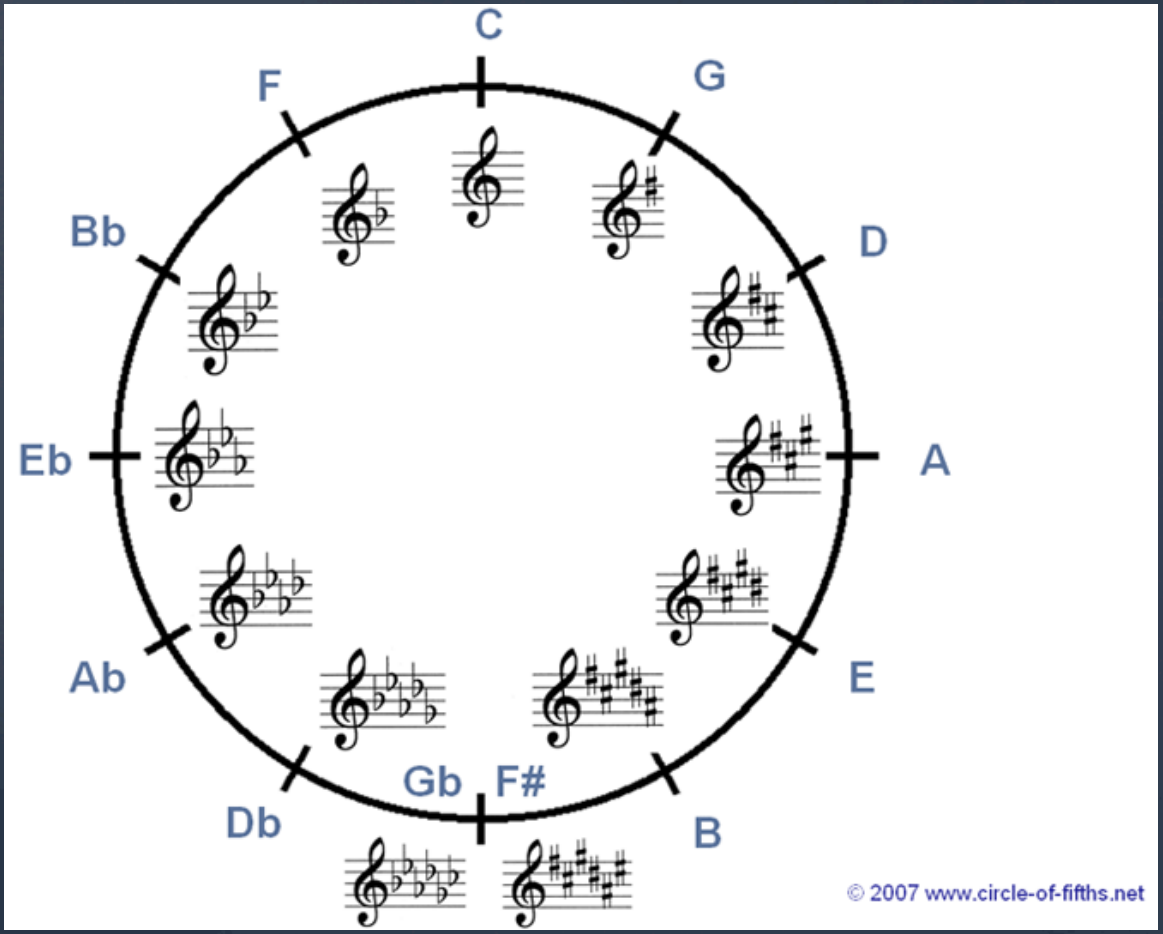
1, 1.25, 1.6, 2, 2.5, $\sqrt{10}$, 4, 5, 6.25, 8, 10

So what?

Why learn a weird new way to count from 1 to 10?


Represented value
Perfect exponential curve;
relative precision is absolutely flat.
1 decibel = $10^{1/10}$ = 1.26···
1 1.25 1.6 2 2.5 √10 4 5 6.25 8 10
1 1.26··· 1.58··· 2.00··· 2.51··· √10 3.98··· 5.01··· 6.31··· 7.94··· 10
0 1 2 3 4 5 6 7 8 9 10
0 1 2 3 4 5 6 7 8 9 10
lattice point, counting up from zero

Like the "circle of fifths" in music



Made possible by another logarithmic coincidence.

Interval of an octave is 2:1
Interval of a fifth is 3:2

Go up a fifth, twelve times. What is the ratio?

1.5^{12} is almost exactly seven octaves!

The equal-tempered scale is logarithmic, yet closely approximates the ratio of small integer ratios.

Non-negative exact values

0, 0.0008,

0.001, 0.00125, 0.0016, 0.002, 0.0025,
0.001√10, 0.004, 0.005, 0.00625, 0.008,

0.01, 0.0125, 0.016, 0.02, 0.025,
0.01√10, 0.04, 0.05, 0.0625, 0.08,

0.1, 0.125, 0.16, 0,2 , 0.25,
0.1√10, 0.4, 0.5, 0.625, 0.8,

1, 1.25, 1.6, 2 , 2.5,
√10, 4, 5, 6.25, 8,

10, 12.5, 16, 20 , 25,
10√10, 40, 50, 62.5, 80,

100, 125, 160, 200, 250,
100√10, 400, 500, 625, 800, 1000,

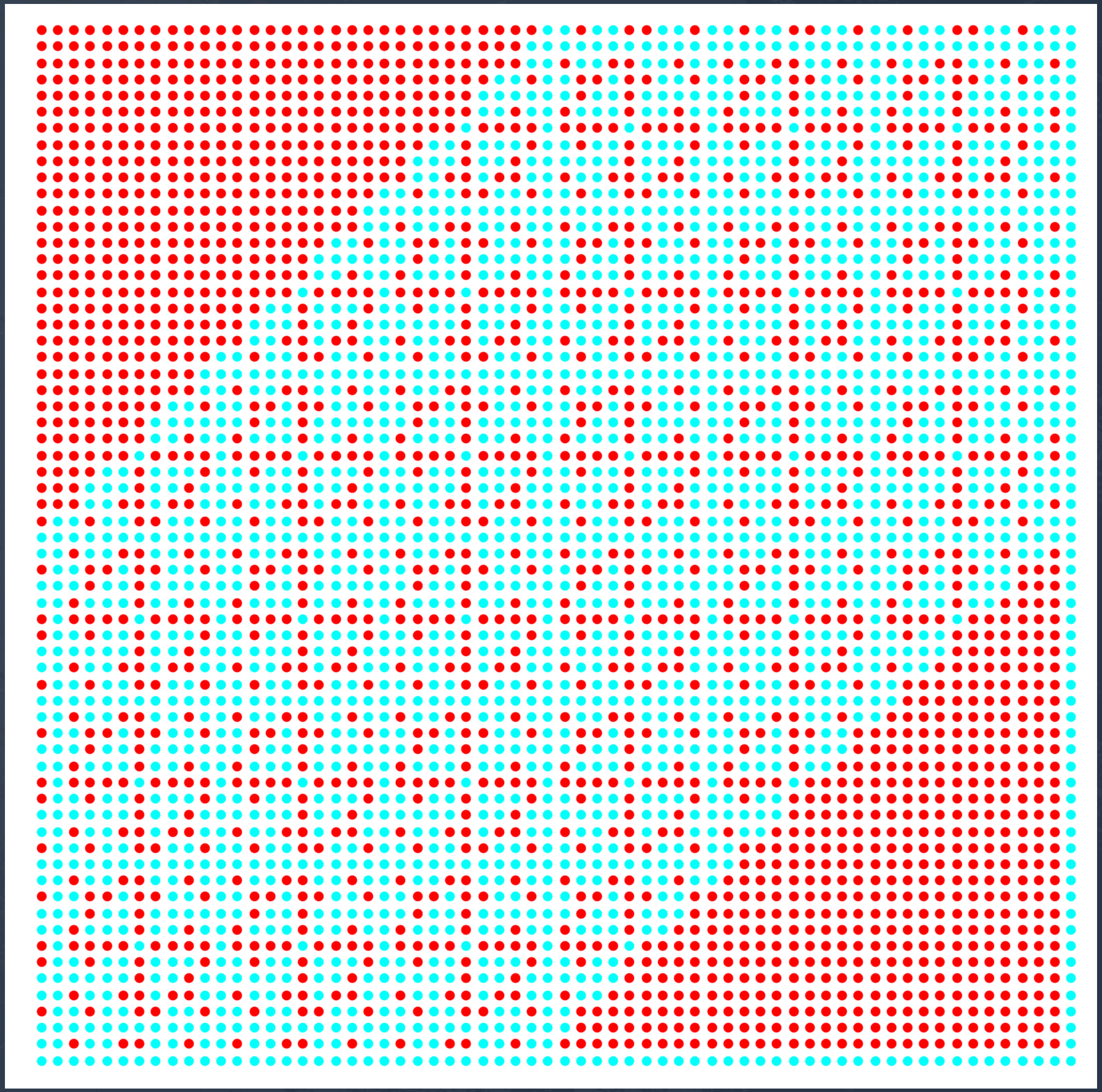
1250, /0

- With negatives and open ranges, 256 values (1 byte)
- Over six orders of magnitude
- Only one digit precision, but the precision is *flat*
- Exact *decimals*, except for √10. (If you don't like it, ignore it)

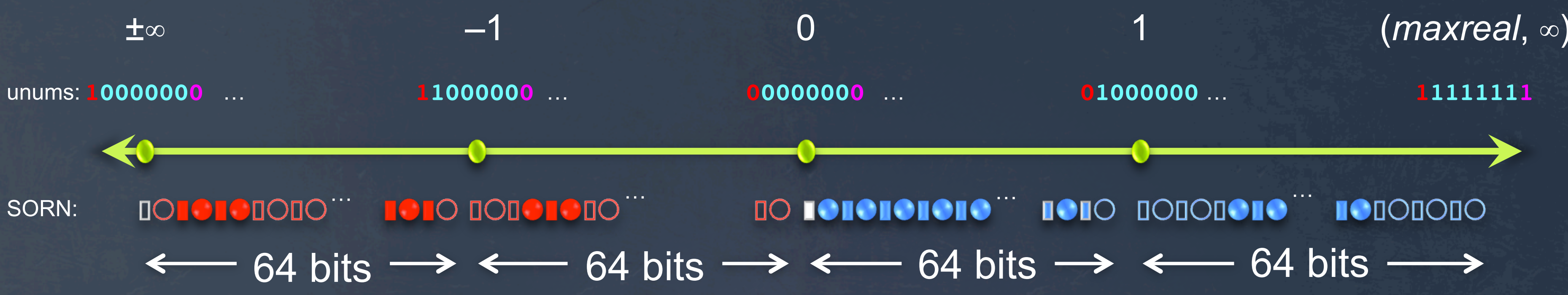
Closure plot for multiply, divide

- ● = Exact result
- ● = Inexact (single ULP range)

Embedded ● are where the power of 2 and the power of 5 differ by more than 4.



8-bit unum means 256-bit SORN



Ultra-fast parallel arithmetic on *arbitrary* subsets of the real number line.

Ops can still finish within a single clock cycle, with a tractable number of parallel OR gates.

Only need 16-bit SORN for + – × ÷ ops

Connected sets *remain connected* under + – × ÷ , even division by zero!

Run-length encoding of a contiguous block of 1s amongst 256 bits only takes 16 bits.

`00000000 00000000` means all 256 bits are `0`s
`xxxxxxxx 00000000` means all 256 bits are `1`s (if any `x` is nonzero)
`00000010 00000110` means there is a block of 2 `1`s starting at position 6

↑ 2 ↑ 6

Trivial logic still serves to negate and reciprocate compressed form of value.

Table look-up background

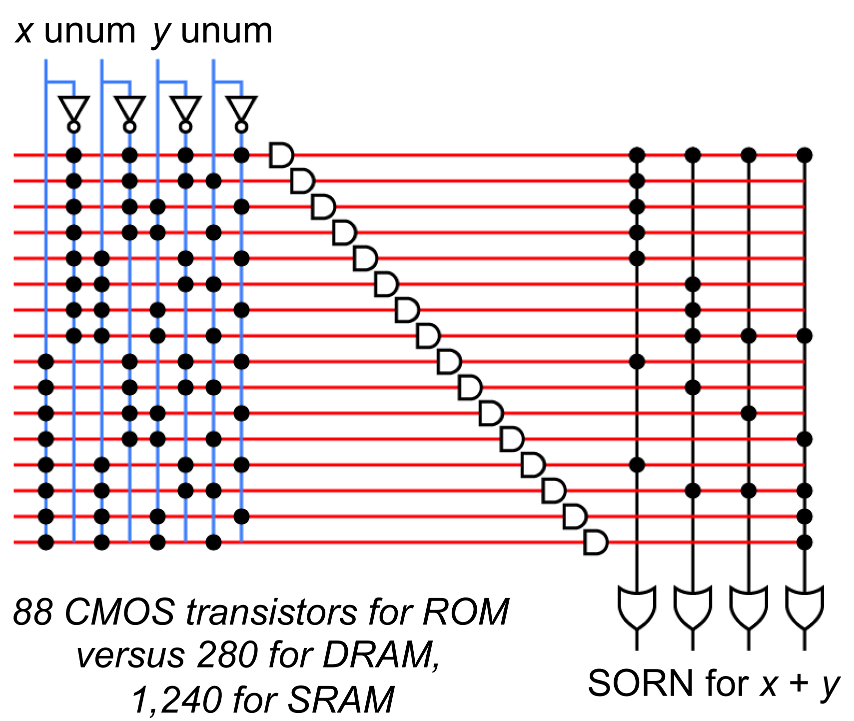
In 1959, IBM introduced its 1620 Model 1 computer, internal nickname "CADET."

All math was by *table look-up*.

Customers decided CADET stood for "Can't Add, Doesn't Even Try."

Table look-up requires ROM

- Read-Only Memory needs very few *transistors*. ~3x denser than DRAM, ~14x denser than SRAM.
- Billions of ROM bits per chip is easy.
- Imagine the *speed*… all operations take 1 clock! Even x^y.
- 1-op-per clock architectures are much easier to build.
- Single argument-operations require tiny tables. Trig, exp, you name it.



88 CMOS transistors for ROM versus 280 for DRAM, 1,240 for SRAM

Low-precision *rigorous* math is possible at 100x the speed of sloppy IEEE floats.

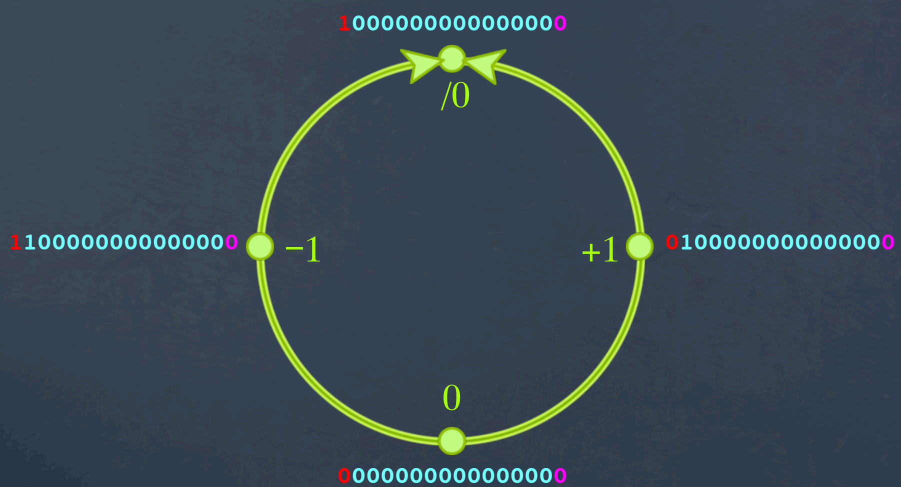
Cost of + – × ÷ tables (naïve)

- Addition table: 256×256 entries, 2-byte entries = 128 kbytes
- Symmetry cuts that in half, if we sort *x* and *y* inputs so $x \le y$. Other economizations are easy to find.
- Subtraction table: just reflect the addition table
- Multiplication table: same size as addition table
- Division table: just reflect the multiplication table!
- Estimated chip cost: < 0.01 mm^2, < 1 milliwatts

128 kbytes total for all four basic ops.
Another 64 kbytes if we also table x^y.

What about, you know, *decent* precision? Is 3 decimals enough?

IEEE half-precision (16 bits) has ~3 decimal accuracy
9 orders of magnitude, 6×10^{-5} to 6×10^{4}.
Many bit patterns wasted on NaN, negative zero, etc.
Can a 16-bit unum do better, and *actually express decimals exactly*?



65536 bit patterns. 8192 in the “lattice”.
Start with set = {1.00,1.01, 1.02,…, 9.99}.
Unite with reciprocals.
While set size < 16384:
    unite with 10× set.
Clip to 16384 elements centered at 1.00
Unite with negatives.
Unite with open intervals between exacts.
What is the *dynamic range*?

Answer: *9*⁺ orders of magnitude

/0.389×10⁻⁵ to 0.389×10⁵

This is 1.5 times *larger* than the range for IEEE half-precision floats.

```
nbits = 16;
digits = 3; set = Range[10^(digits-1), 10^digits - 1] / 10^(digits-1);
set = Union[set, 10 / set];
set = Union[set, set / 10];
While[Length[set] < 2^(nbits-2),
  set = Union[set, set / 10, set * 10]];
Off[General::infy]
m = ⌈Length[set] / 2⌉;
set = Union[{0, 1 / 0},
   Take[set, {m - 2^(nbits-3) + 1, m + 2^(nbits-3) - 1}]];
set = Union[set, -set];
Length[set]

32 768
```

This is the *Mathematica* code for generating the number system.

Notice: no “gradual underflow” issues to deal with. No subnormal numbers.

IEEE Intervals vs. SORNs

- Interval arithmetic with IEEE 16-bit floats takes 32 bits
  - Only 9 orders of magnitude dynamic range
  - NaN exceptions, no way to express empty set
  - Requires rare expertise to use; nonstandard methods
  - Uncertainty grows *exponentially* in general (or worse)

- SORN arithmetic with connected sets takes 32 bits
  - Over 9 orders of magnitude dynamic range
  - No indeterminate forms; closed under + – × ÷
  - Automatic control of information loss
  - Uncertainty grows *linearly* in general

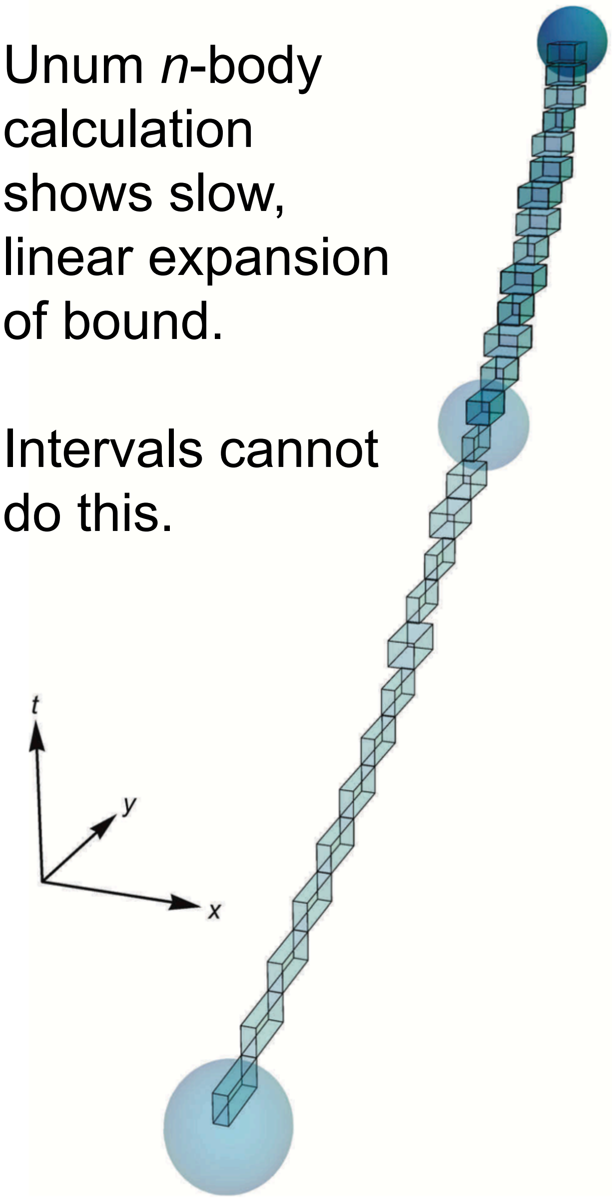
Why unums don't have the interval arithmetic problem

Intervals: Each step starts from the *interval* produced in the previous step.
⇒ Bounds grow *exponentially*

Unums: Each stage of a calculation starts with values that are either *exact* or *one ULP wide*, and then takes the *union* of the results.
⇒ Bounds grow *linearly*.

Unum *n*-body calculation shows slow, linear expansion of bound.

Intervals cannot do this.



“Dependency Problem” ruins interval arithmetic results

“Let x = [2, 4]. Repeat several times: $x \leftarrow x - x$; Print x.”

Intervals (128 bits):

[–2, 2]
[–4, 4]
[–8, 8]
[–16, 16]
[–32, 32]
[–64, 64]
[–128, 128]

Unstable*. The uncertainty feeds on itself, so interval widths grow exponentially.*

SORNs (8-bit unums):

(–1, 1)
(–0.2, 0.2)
(–0.04, 0.04)
(–0.01, 0.01)
(–0.002, 0.002)
(–0.0004, 0.0004)
(–0.0008, 0.0008)

Stable*. Converges to the smallest open interval containing zero.*

Another classic example of "the Dependency Problem"

"Let x = [2, 4]. Repeat several times: $x \leftarrow x / x$; Print x."

Intervals:

[1/2, 2]
[1/4, 4]
[1/16, 16]
[1/256, 256]
[1/65536, 65536]
⋮

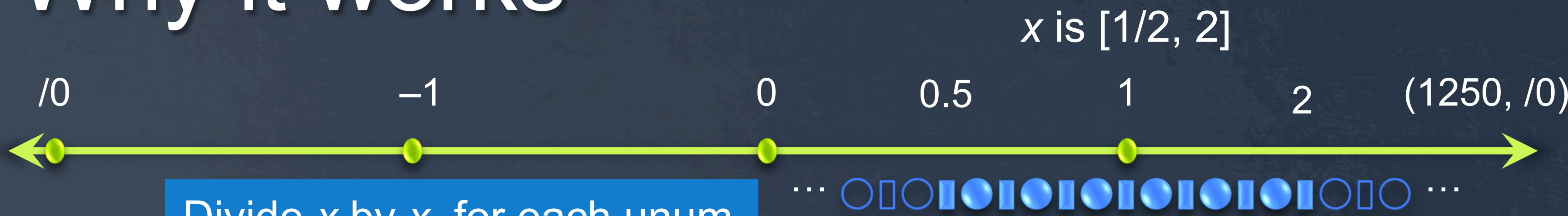
Unstable*. Again, the interval widths grow very rapidly.*

SORNs (8-bit unums):

(0.625, 1.6)
(0.625, 1.6)
(0.625, 1.6)
(0.625, 1.6)
(0.625, 1.6)
⋮

Stable*. Contains the correct value, 1, despite only single-digit accuracy*

Why it works



Divide *x* by *x*, for each unum whose presence bit is set. (Ideally, do these in parallel.)

Compiler sets the hardware mode to “dependent” so all table look-ups are *reflexive*, not all-to-all.

0.5 / 0.5	= 1
(0.5, 0.625) / (0.5, 0.625)	= (0.8, 1.25)
0.625 / 0.625	= 1
(0.625, 0.8) / (0.625, 0.8)	= (0.78125, 1.28) ◗ (0.625, 1.6)
0.8 / 0.8	= 1
(0.8, 1) / (0.8, 1)	= (0.8, 1.25)
1 / 1	= 1
(1, 1.25) / (1, 1.25)	= (0.8, 1.25)
1.25 / 1.25	= 1
(1.25, 1.6) / (1.25, 1.6)	= (0.78125, 1.28) ◗ (0.625, 1.6)
1.6 / 1.6	= 1
(1.6, 2) / (1.6, 2)	= (0.8, 1.25)
2 / 2	= 1

OR the SORN bits to form the union: = (0.625, 1.6)

Future Directions

- Create 32-bit and 64-bit unums with new approach; table look-up still practical?
- Compare with IEEE single and double
- General SORNs need run-length encoding.
- Build C, D, Julia, Python versions of the arithmetic
- Test on various workloads, like
  - Deep learning
  - *N*-body
  - Ray tracing
  - FFTs
  - Linear algebra done right (complete answer, not sample answer)
  - Other large dynamics problems

Summary

A complete break from IEEE floats ***may be worth the disruption***.

- Makes every bit count, saving storage/bandwidth, energy/power
- Mathematically superior in every way, as sound as integers
- Rigor without the overly pessimistic bounds of interval arithmetic

This is a path beyond exascale.