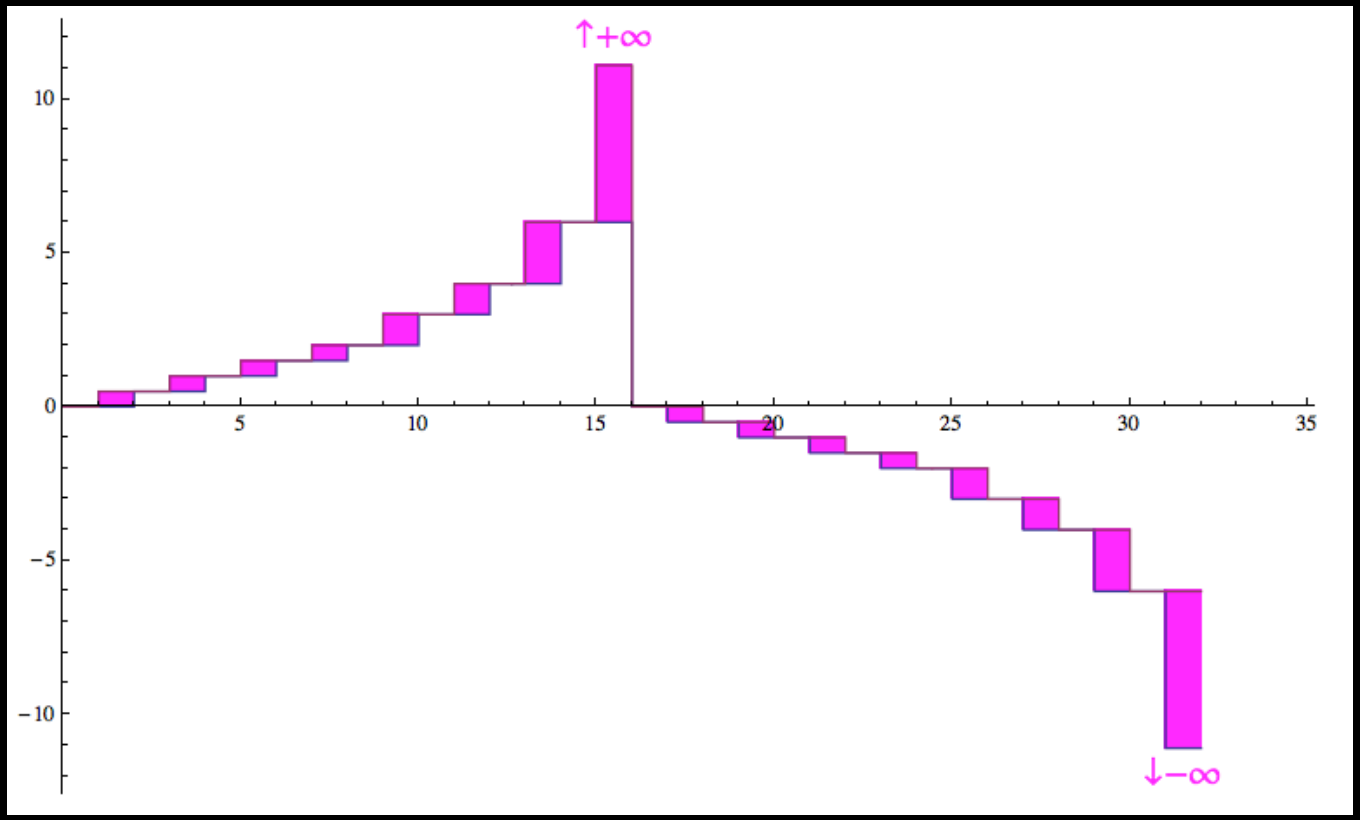
# *Unleashed Computing:*

## *The need to right-size precision to save energy, bandwidth, storage, and electrical power*

Dr. John L. Gustafson, Senior Fellow, AMD



# *Outline*

- What's wrong with floating point
- What's wrong with interval arithmetic
- A possible fix: "unum" representation
- The "ubox" approach
- Interval physics

# *It's time to rethink computer arithmetic*

- Excess size in integers and floating point wastes memory, bandwidth, energy, and power. And time.
- Programmers use excess size so they don't have to think, or because optimal size is not native to the hardware.
- Current methods come from a time when gates were expensive, wires were practically free. Now it's reversed.
- Overkill precision is a luxury we should consider doing without, for everything from mobile devices (better battery life) to *exascale supercomputers* (20 megawatt maximum).
- What if we use representations that use more gates but demand less bandwidth? And for floating point, *also get better answers?*

# *Analogy: Printing in 1970 vs. 2013*

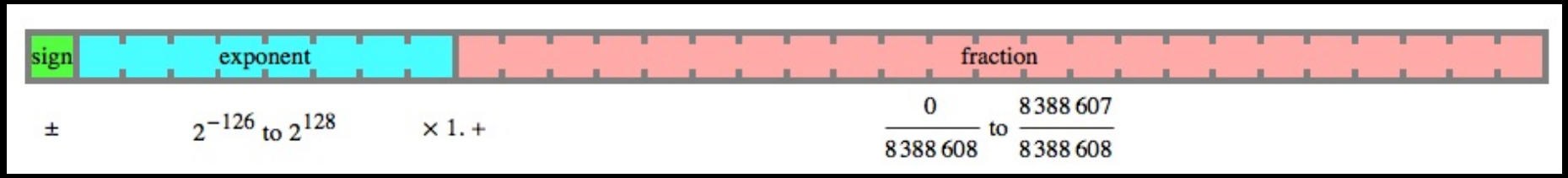
*1970:* 30 sec per page

```
      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
```

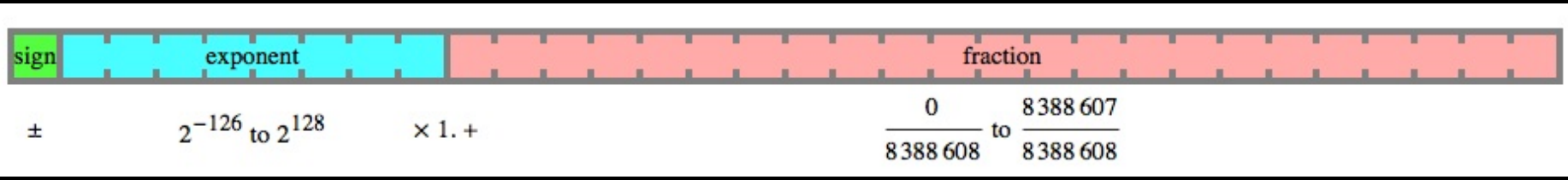
*2013:* 30 sec per page


We use faster technology for *better* prints, not for thousands of lousy prints per second. Why not do the same thing with computer arithmetic?

- What's wrong with floating point
- What's wrong with interval arithmetic
- A possible fix: "unum" representation
- The "ubox" approach
- Interval physics



# *Floating point format is a 99-year-old idea*




- What are the odds that a 1914 design, when gates were precious and wires practically free, is still the right choice?
- Even the IEEE standard (1985) made choices that look dubious now
  - Single precision has a ridiculously large dynamic range, 76 orders of magnitude. (size of known universe to size of proton is 40 orders of magnitude.) Exponent hardware was much easier to build!
  - 8 million different kinds of NaN in single, 4.5 quadrillion kinds of NaN in double precision. Most people use two kinds, at *most*.
  - Given a choice, people will turn underflow exceptions *off*. Rightly so.
  - Negative zero. (ugh!)
- Biggest problem: No accuracy information stored in the number.

# *From Despair.com*

YOU SON OF A BITCH

You just divided by zero, didn't you?

# *Terminology Reminders*

- *Precision* = Digits available to store a number ("32-bit" or "4 decimal", for example)
- *Accuracy* = Number of **valid** digits in a result ("to three significant digits", for example)
- ULP = Unit of Least Precision.

Precise but not accurate: $\pi$ = 3.140000000001
Accurate but not precise: 3 < $\pi$ < 4

Precision is not a goal.
Precision is the means, accuracy is the end.

# *Decades of asking people "How do you know your answer is correct?"*

- "(Laughter) "What do you mean?"*(This is the most common response)*
- "We used double precision."
- "It's the same answer we've always gotten."
- "It's the same answer others get."
- "It agrees with special-case analytic answers."

*Almost no one has the resources for detailed numerical analysis. So we use excess precision.*

# *When rounding error ignorance killed 38 people*

*Patriot missile accident.* Feb1991, American Patriot missile failed to track an Iraqi Scud, instead hit an Army barracks. Traced to inaccurate time caused by incrementing time in tenths of a second. Single-precision FP math silently accumulated error during the 100 hours between system turn-on and actual use.

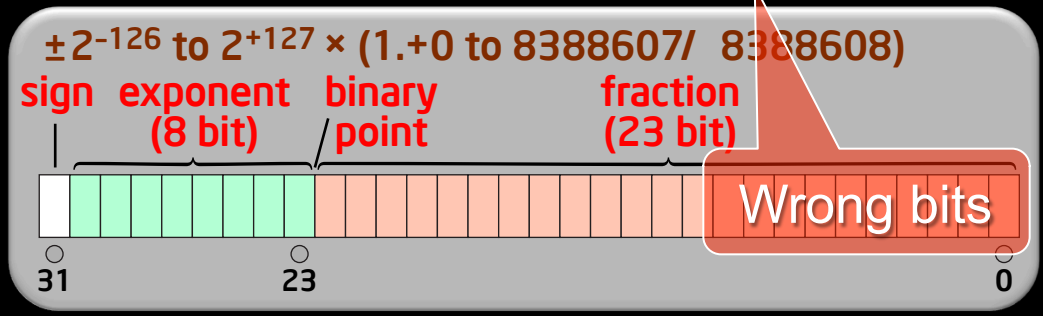
See http://www.fas.org/spp/starwars/gao/im92026.htm

# *Quick Tutorial on Rounding Error*

"0.1" in binary is not exact. It's 0.100000023841857910156 25, rounded.
Programmers and disk archives use decimal, creating rounding error

Clock example: accumulating seconds, 0.1 at a time, for 100 hours, will be off by at least *three minutes*!

Also,
$(a + b) + c$
is NOT the same as
$a + (b + c)$
in floating-point math.




$a$ = 1.0
$b$ = 100000000.
$c$ = –100000000.

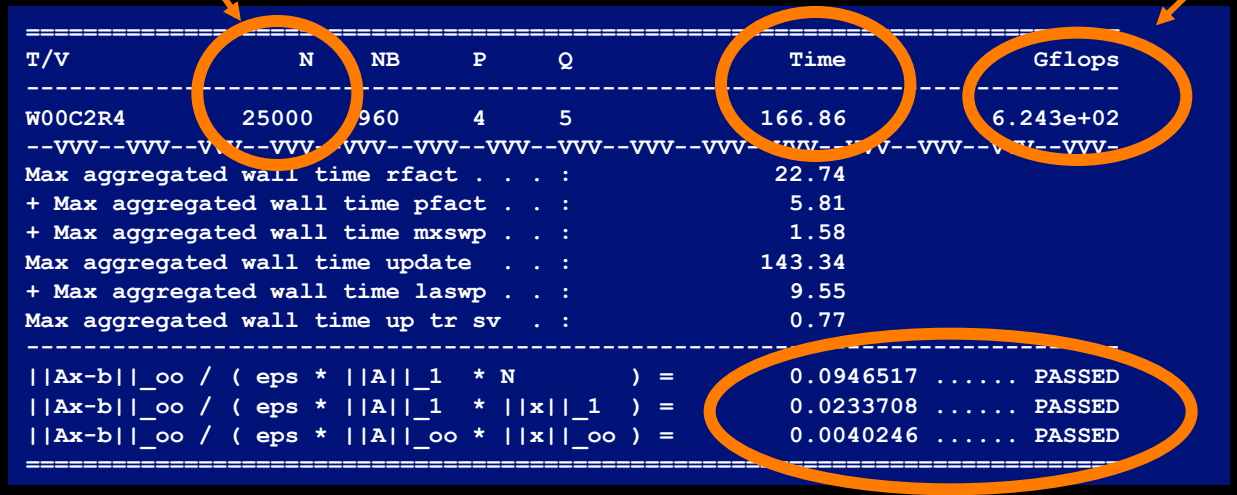
($a$ + $b$) rounds down to = 100000000. Add $c$, get 0.0.
($b$ + $c$) = 0 exactly, with no rounding. Add $a$, get 1.0.

So floating point math flunks algebra!
This is a problem for *parallel* programmers.
Are different answers a *bug* or a *rounding error*?

# *The LINPACK benchmark checks accuracy, but…*

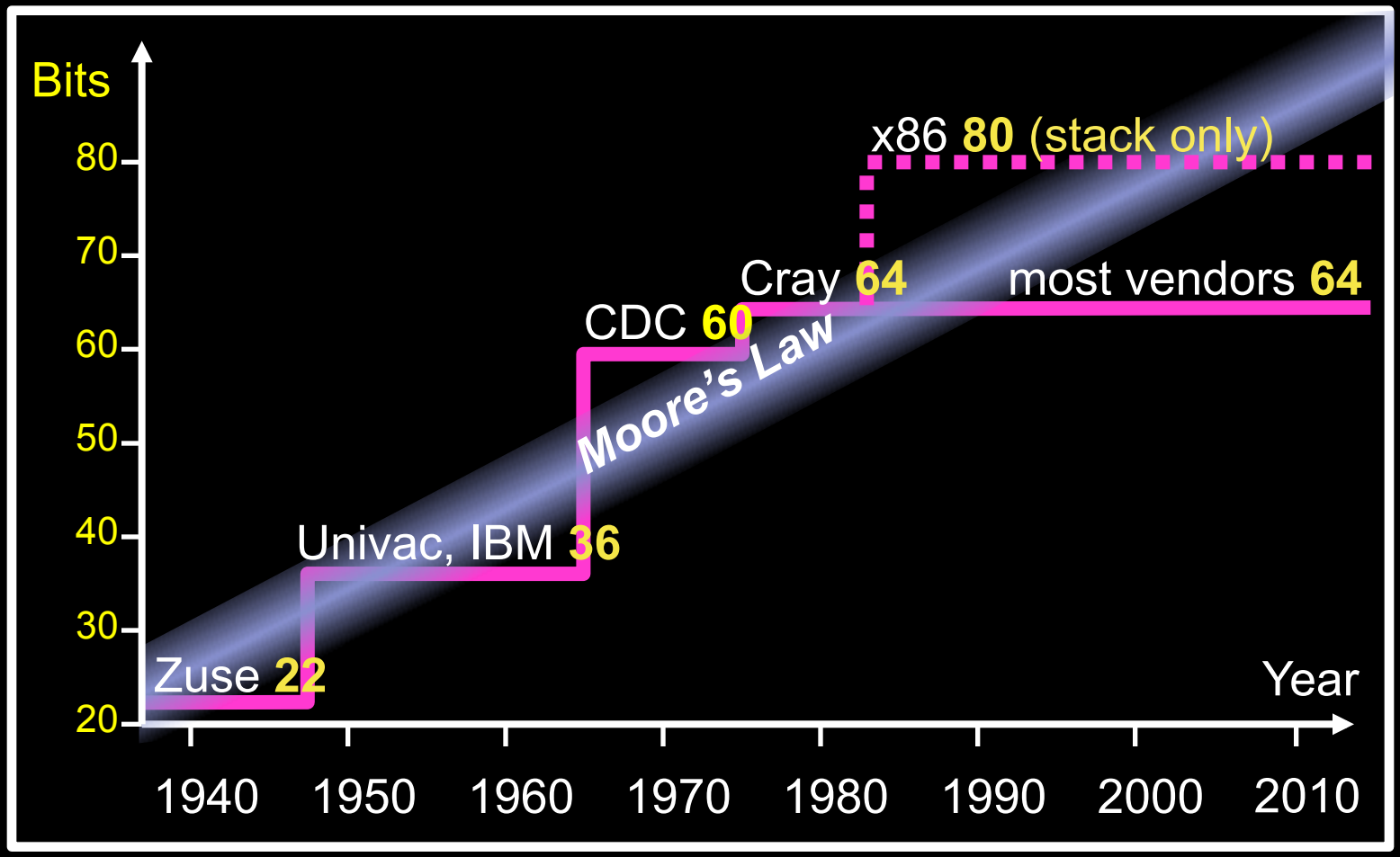
**Number of simultaneous equations**

**Speed (billions of floating-point operations per second)**

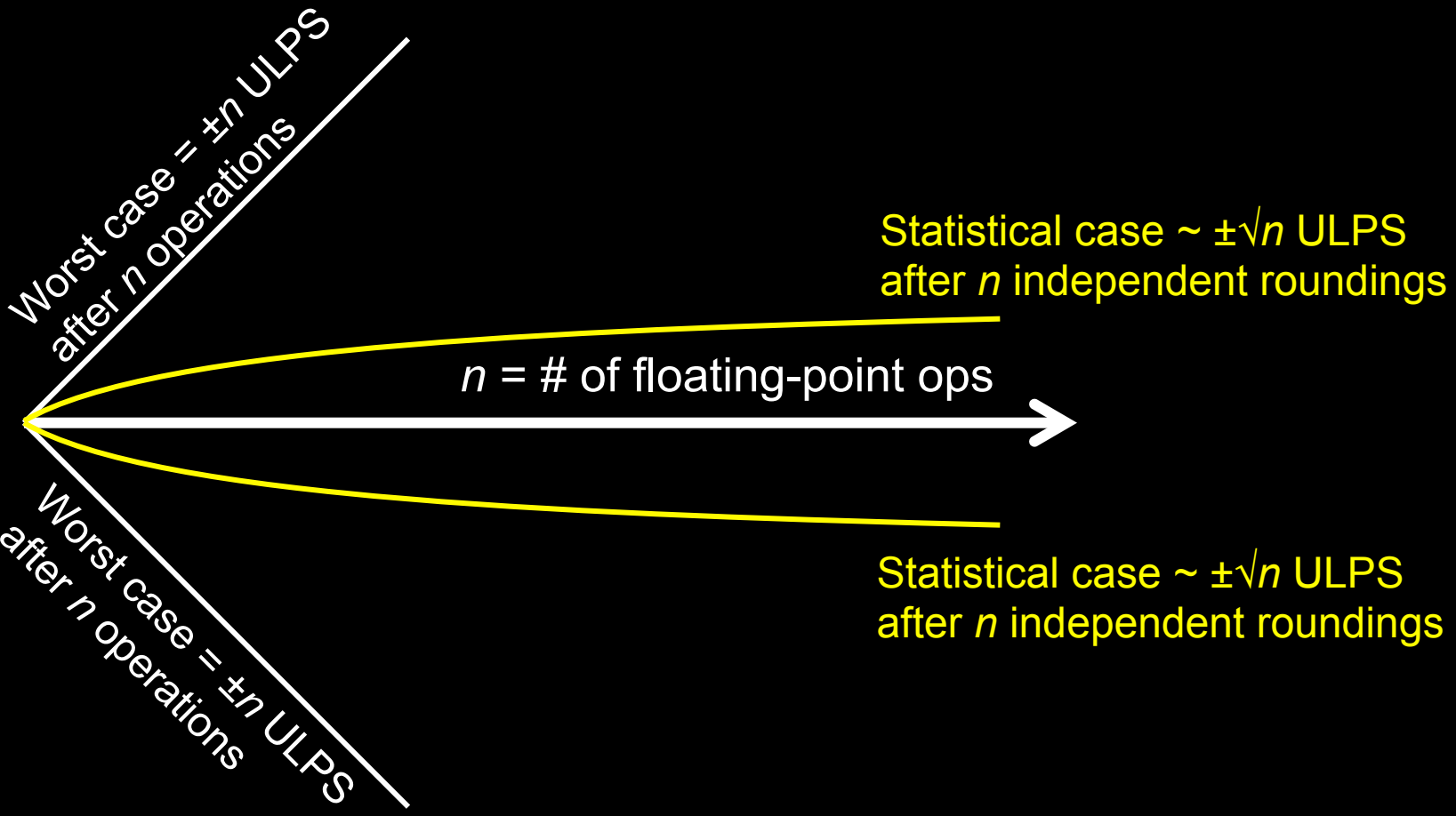



The accuracy of the *Ax* = *b* answer is never published!
What is the cutoff for "PASSED"?

# *Using 64-bit everywhere is like insuring your car for ten million dollars. Some want to jump to 128-bit!*



# *"Unbiased rounding" won't save you*




- Rounding biases are *not* always statistically independent!
- And at petaflops/sec, "creeping crud" accumulates *fast.*

# *Better math could help a wide range of apps*

Angry Birds (box3d games)


Page Rank for Searches

Financial Models

**Black Scholes Pricing Formula**

$$c = S\,N(d_1) - Xe^{-r(T-t)}\,N(d_2)$$

where

$$d_1 = \frac{\ln(S/X) + (r \cdot s^2/2)(T-t)}{s\sqrt{T-t}}$$

$$d_2 = d_1 - s\sqrt{T-t}$$

c = call option price
S = current stock price
X = exercise price
$e^{-r(T-t)}$ = continuously compounded risk free rate
s = standard deviation of stock price returns
T = maturity date
t = date option is being valued
N(x) = cumulative probability distribution function for a standardized normal variable

mp3 Players

# *Excess precision burdens DRAM energy, which improves much slower than Moore's Law*

| *Operation* | *Energy consumed* |
|---|---|
| 64-bit multiply-add | 64 pJ |
| Read/store register data | 6 pJ |
| **Read 64 bits from DRAM** | **4200 pJ** |
| **Read 32 bits from DRAM** | **2100 pJ** |

Using single precision in DRAM instead of double saves as much energy as *30* floating-point operations.

Data is for 32 nm technology ca. 2010

# *It's so counterintuitive…*

Copying numbers…

$$\begin{array}{ll} & 0.583929181098938 \times 10^{12} \\ \times & 0.452284961938858 \times 10^{9} \\ \hline \end{array}$$

# *It's so counterintuitive…*

Copying numbers…

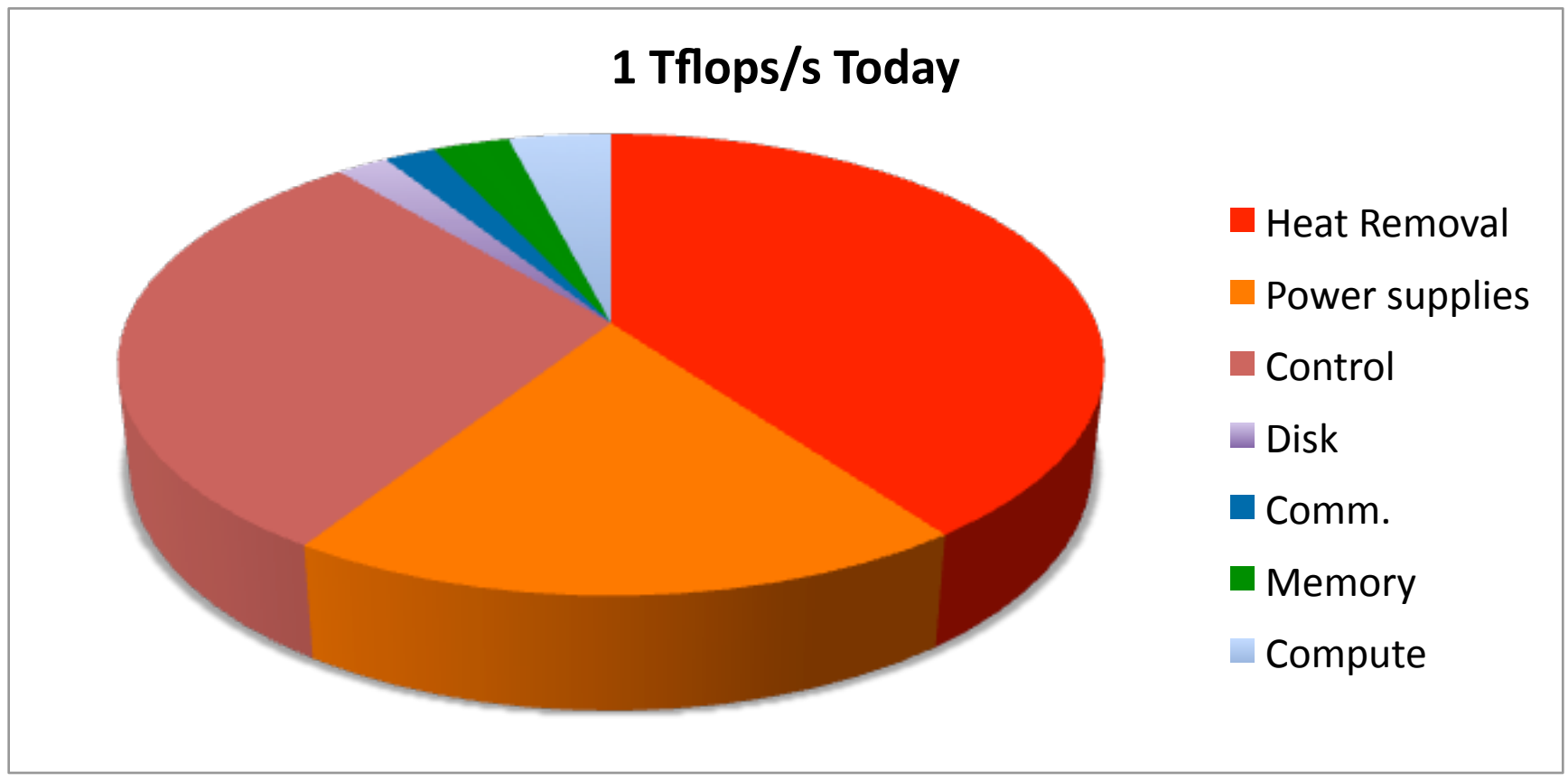
…is harder than doing THIS with them?

$$0.583929181098938 \times 10^{12} \times 0.452284961938858 \times 10^{9}$$

```
  467143344879150 4
 29196459054946 90
 467143344879150 4
 467143344879150 4
 175178754329681 4
 52553626298904 42
 58392918109893 8
 35035750865936 28
 5255362629890 442
 2335716724395 752
 467143344879150 4
 116785836219787 6
 116785836219787 6
 29196459054946 90
 2335716724395 752
```

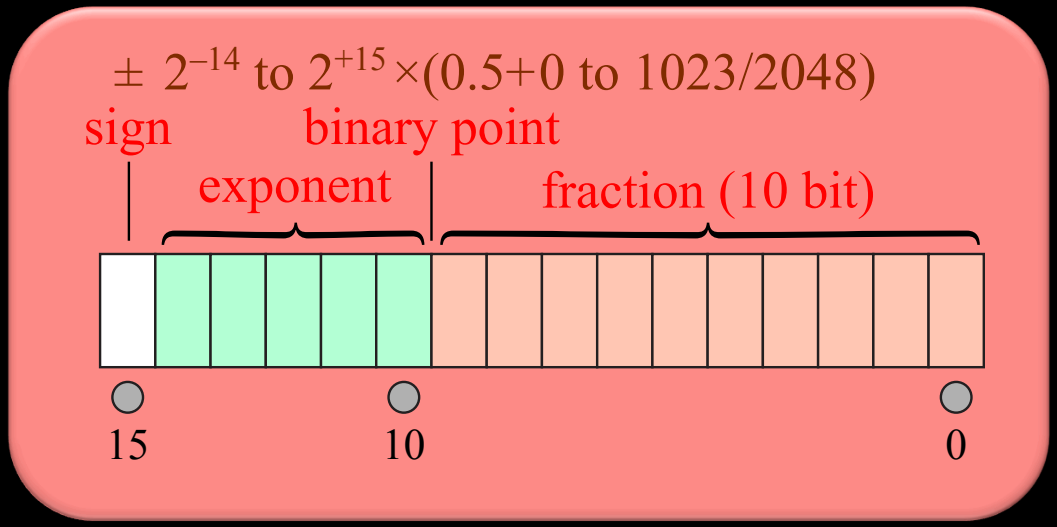
$$0.264102387448322 \times 10^{21}$$

# *LINPACK power consumed (light on DRAM use)*




DRAM transfers more typically consume 25% of the power of a server, and this percentage is going ***up***.

# *Don't laugh: 16-bit floating point is pretty good!*



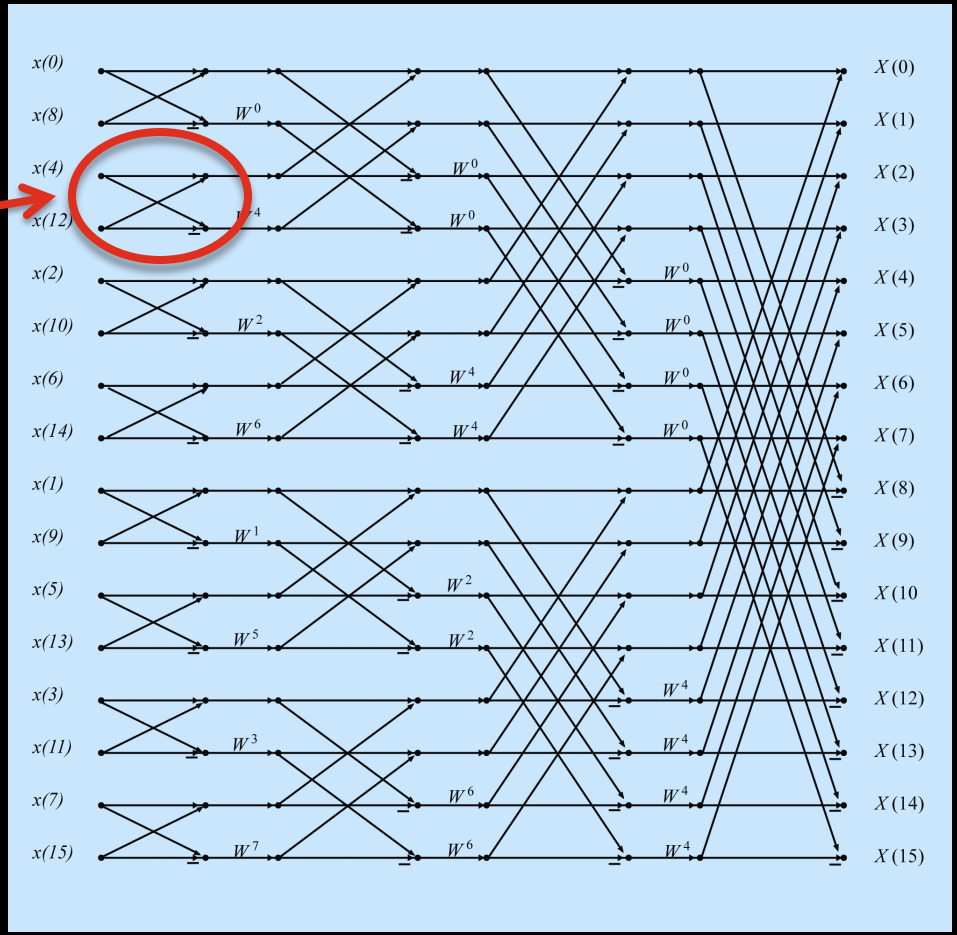

- 50% reduction in bandwidth and memory use, versus 32-bit
- Three decimals of accuracy (0 to 2048 are exact)
- Dynamic range of 12 orders of magnitude ($6\times10^{-8}$ to $6\times10^{4}$)
- 4x savings over 64-bit *if* it can be made numerically safe
- Graphics, seismic, medical imaging are good candidates, but there may be many more uses

# *Think about the Fast Fourier Transform kernel*

Data often comes from A/D converters with 12 or fewer bits of precision.

For each complex “butterfly” operation, you do an $x + y$ and an $x – y$; for one, you always wish you had more bits; for the other, you always wish you didn’t have so many!

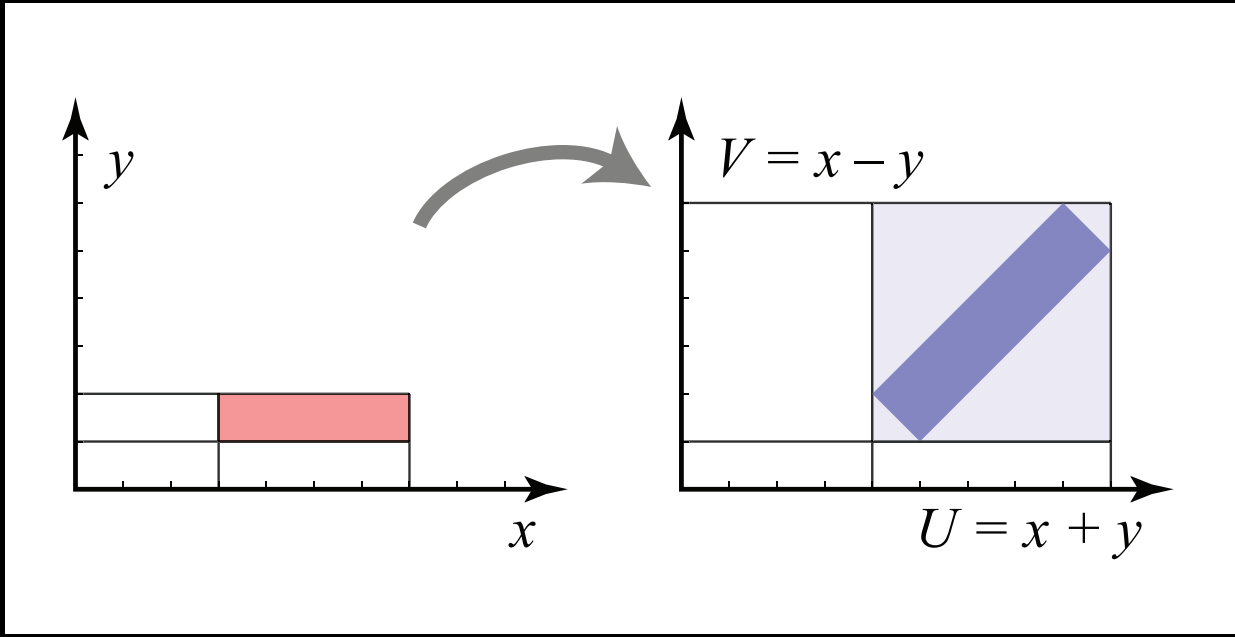
*Current arithmetic doesn’t let you adjust either way.*



- What's wrong with floating point
- What's wrong with interval arithmetic
- A possible fix: "unum" representation
- The "ubox" approach
- Interval physics

[*a*, *b*] means all *x* such that $a \leq x \leq b$

# *Reason 1 why interval math hasn't displaced floating point: The "Wrapping Problem"*




Answer sets are complex shapes in general, but interval bounds are axis-aligned boxes, period.

No wonder interval bounds grow far too fast to be useful, in general.

# *Reason 2: The Single Use Expression Problem (sometimes called the Correlation Problem)*

What wrecks interval arithmetic is simple things like

$$F(x) = x - x.$$

Should be 0, or maybe $[-\varepsilon, +\varepsilon]$. Say $x$ is the interval [3, 4], then interval $x - x$ stupidly evaluates to [–1, +1], which doubles the uncertainty (interval width) and makes the interval solution far inferior to the point arithmetic method.

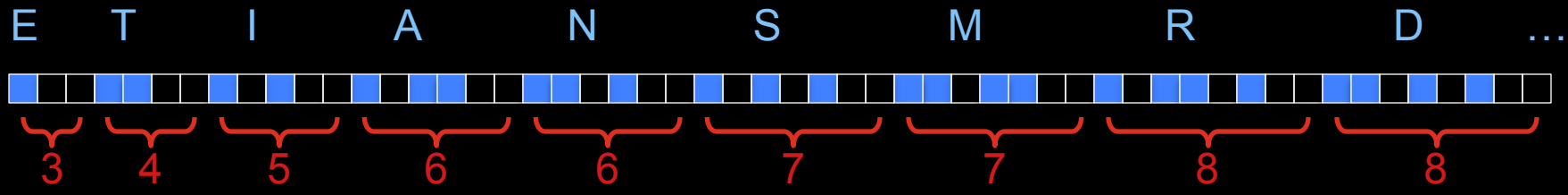
Interval proponents say we should seek expressions where each variable only occurs once (SUE = Single Use Expression). But that's impractical or impossible in general.

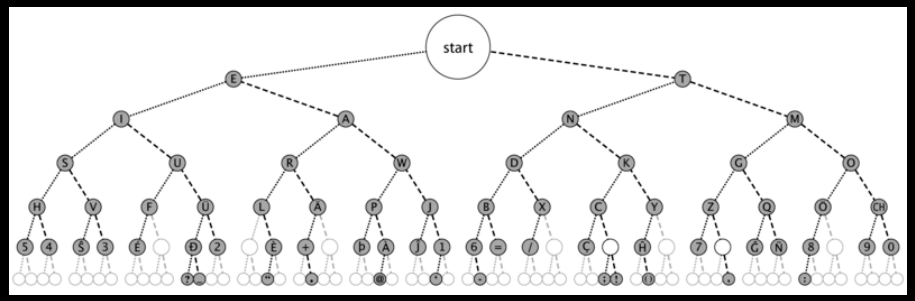
- What's wrong with floating point
- What's wrong with interval arithmetic
- A possible fix: "unum" representation
- The "ubox" approach
- Interval physics

$$-\ \mathbf{0011}\left(=\frac{1}{2^4}\right)\ \times\ 1.\mathbf{11101011110100001011}1\downarrow\ 011\ 10101\ =\ -0.120072$$

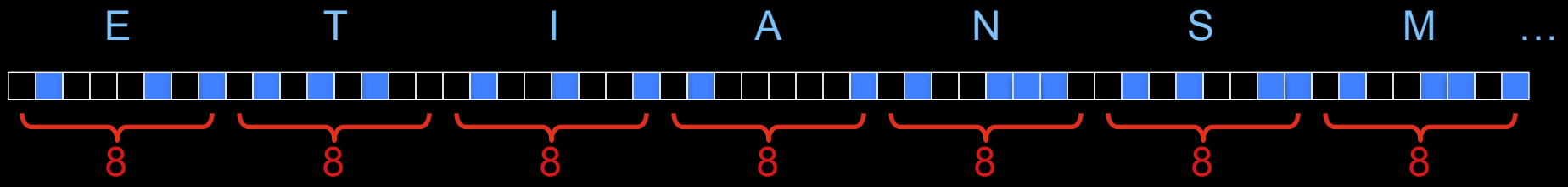
# *Long-distance communication was precious in 1836, so…*

- Morse Code: Use shortest bit strings for commonest data




- Compare that concise efficiency with ASCII:

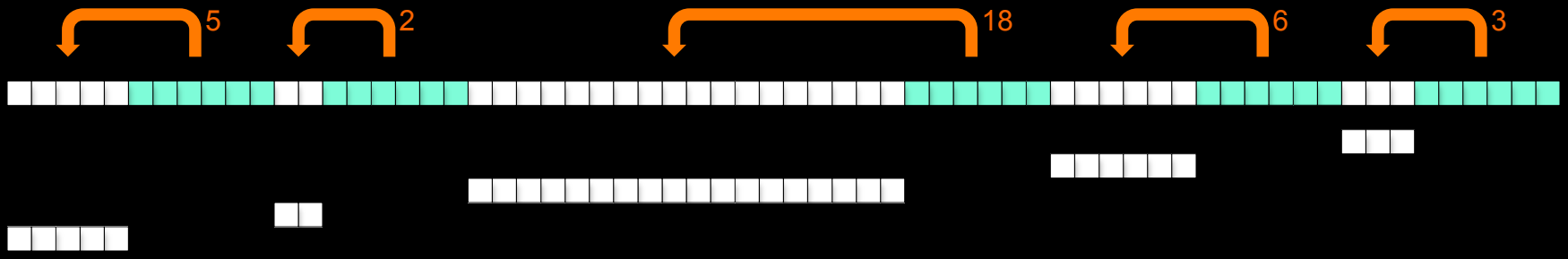


# *Idea: Integer Compression by Size Tagging*

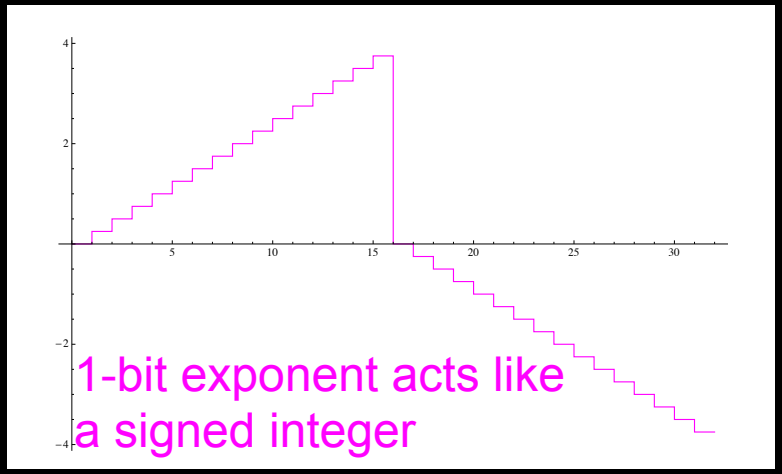
- Just think of how many programmers use a *32-bit* loop counter even when the loop goes from, like, 0 to 3.
- Approach 1: Last two bits (on byte boundary) indicate four possible lengths on the left: 6-bit, 14-bit, 30-bit, or 62-bit integers
  - Preserves byte alignment (but disrupts word alignment)
  - No more “short” “long int” etc. cases. Just “int”. (“char” is still a byte.)
  - Hardware promotes as needed (for addition, multiplication) up to the limit, and demotes if possible (for subtraction, division)
  - Doesn’t overflow until you exceed 4,611,686,018,427,387,903.
  - Simplifies integer instruction set, but adds gates to the ALU. I’m betting this is a net win within the CPU, and then you also get about a 2x reduction in bandwidth based on preliminary experiments.

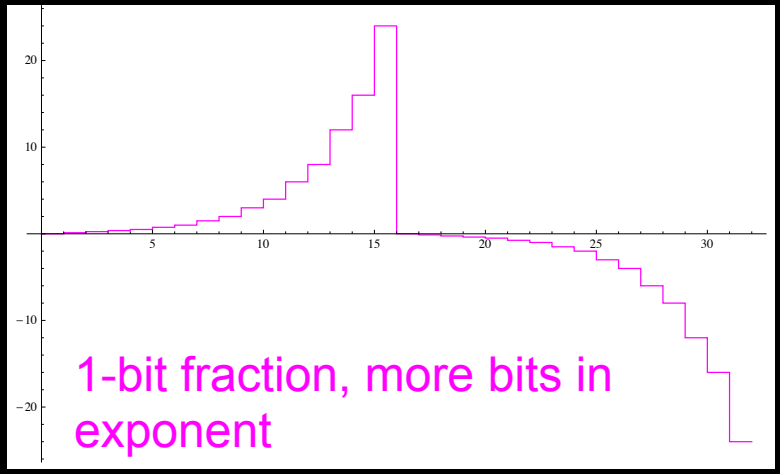
# *Here's where you'll start to get uncomfortable*

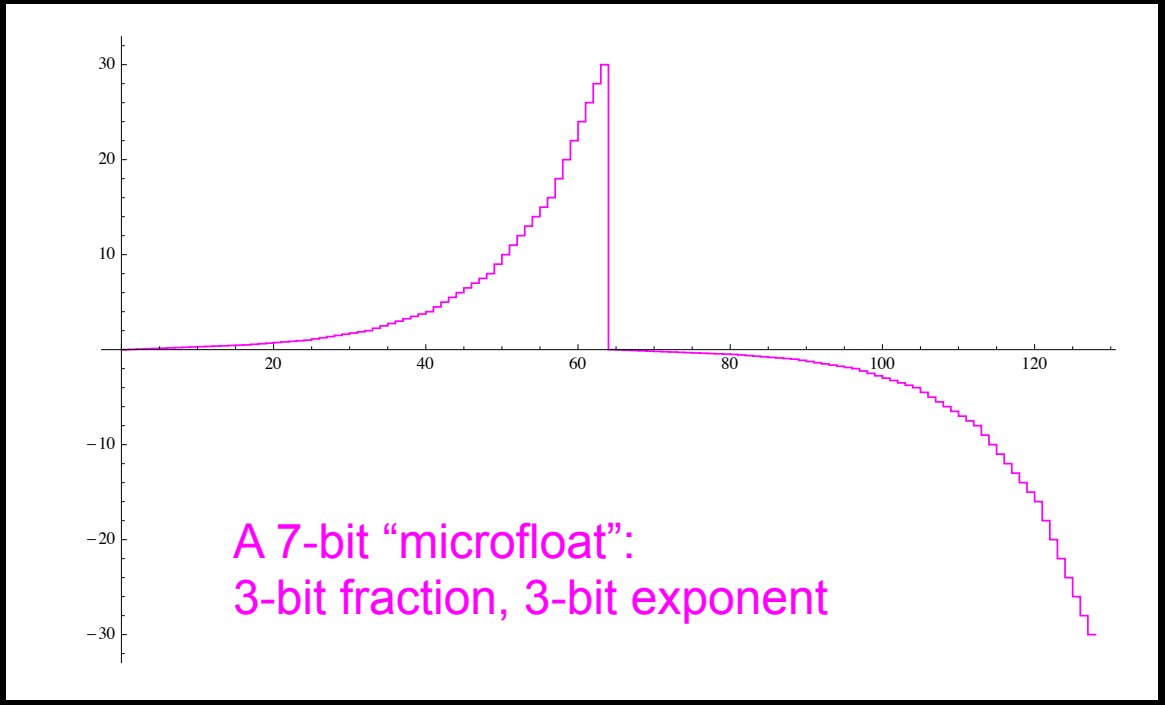
- Approach 2: Six-bit tag indicates *one less than* size of integer
  - Tag = 000000 means a Boolean,
  - Tag = 001101 means a fourteen-bit int
- Looks like a three-dollar bill to experienced architects
- Ivan Sutherland talks about "Overcoming the Tyranny of the Clock"; I am suggesting "Overcoming the Tyranny of the Word Size".
- Packing and unpacking data blocks introduces garbage collection and bit addressability but might be well worth it. Hey, transistors are cheap.
- Imagine automatic upsizing and downsizing int size after arithmetic, instead of allowing "worst case" fixed size everywhere.



# *Flexible exponent and fraction sizes can adjust to needs*

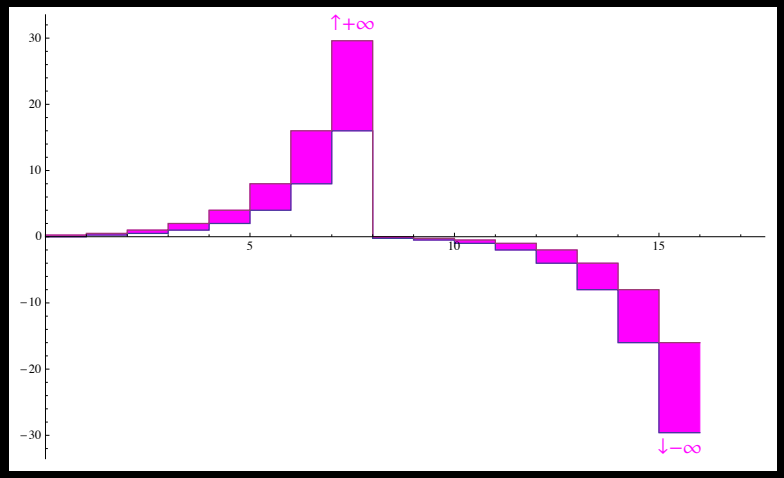



x axis is unsigned integer binary;
y axis is value it represents as a float

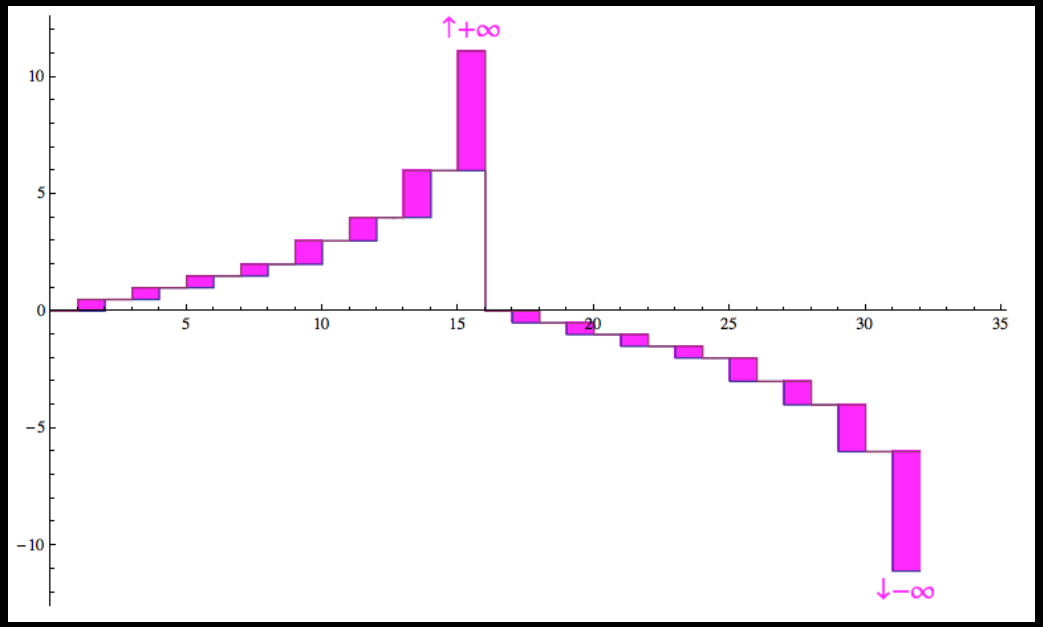
# *Store the "inexact flag" in the number.*




With inexact bit set, the represented values are open intervals one ULP wide.

Regard last bit as the inexact flag and you get two monotone functions.

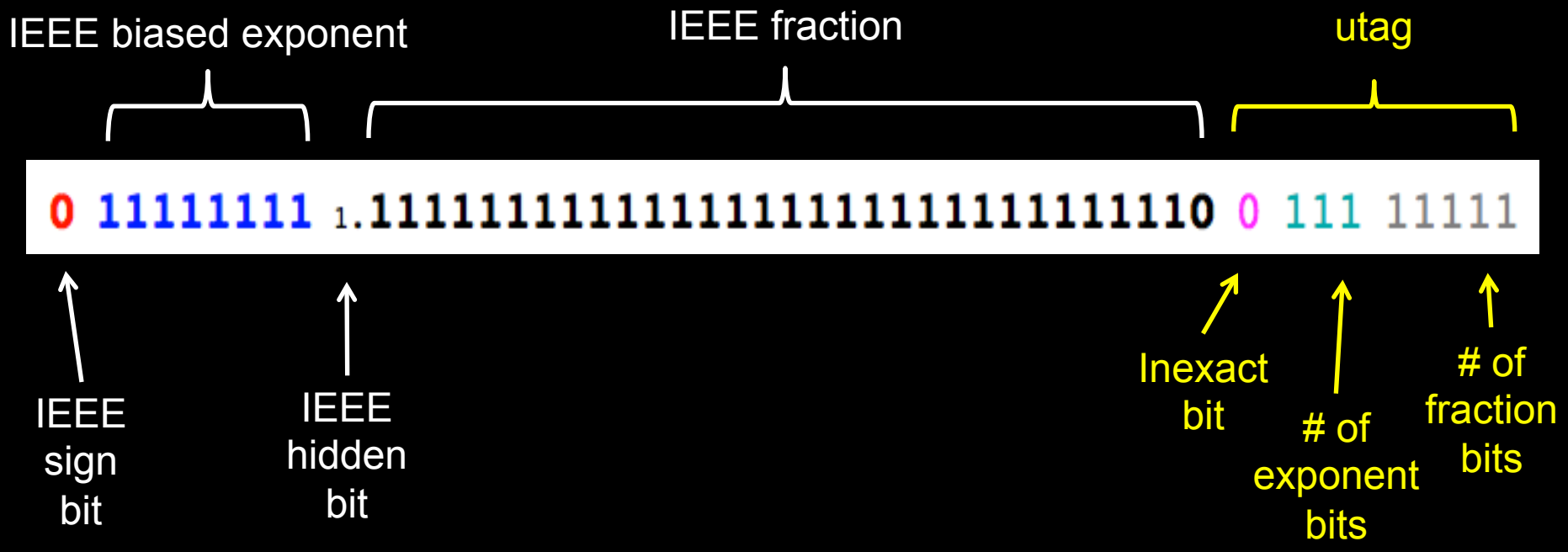
Combine that with flexible precision, and you have the key to an *accuracy-aware numerical environment*



# *That was all warm-up for this: Unums*

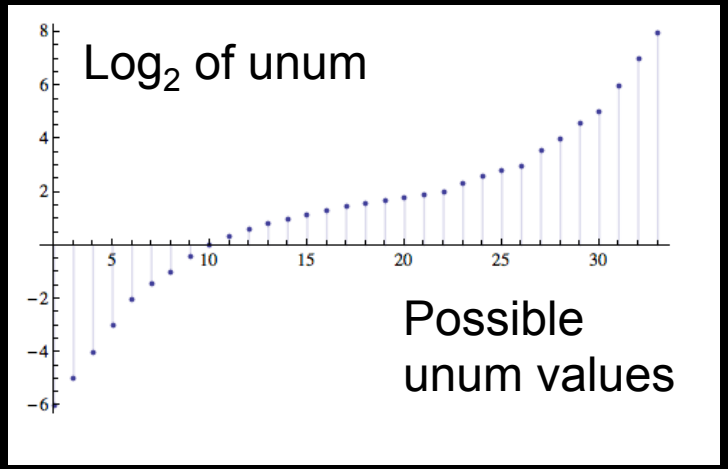
**"Unums"(universal numbers) are to floating point what floating point is to fixed point.**

Floating-point values self-describe their scale factor, but fix the exponent and fraction size. Unums self-describe the **exponent size, fraction size**, and **inexact state**, and include fixed point and IEEE floats as *special cases*.



# *Advantages of Unum Format*

- Common numbers like 0, –1/2, 3, use fewer bits. Rare numbers use more bits. Net: about a 2x savings.
- Subtracting similar numbers destroys significance; unums automatically reduce storage *and* track the significance
- Adding numbers automatically promotes significance (up to a limit), often eliminating roundoff errors, preserving associativity
- No need for underflow, overflow, rounding
- A 12-bit utag allows everything from IEEE half precision to quad precision. And a lot more.


- Math libraries can be much faster since they need only compute to the stored level of significance

# A Related Idea: Asynchronous Design

## An Asynchronous Floating-Point Multiplier

Basit Riaz Sheikh and Rajit Manohar
Computer Systems Laboratory
Cornell University
Ithaca, NY 14853, U.S.A.
{basit,rajit}@csl.cornell.edu

*Abstract*—We present the details of our energy-efficient asynchronous floating-point multiplier (FPM). We discuss design trade-offs of various multiplier implementations. A higher radix array multiplier design with operand-dependent carry-propagation adder and low handshake overhead pipeline design is presented, which yields significant energy savings while preserving the average throughput. Our FPM also includes a hardware implementation of denormal and underflow cases. When compared against a custom synchronous FPM design, our asynchronous FPM consumes 3X less energy per operation while operating at 2.3X higher throughput. To our knowledge, this is the first detailed design of a high-performance asynchronous IEEE-754 compliant double-precision floating-point multiplier.

we primarily focus on double-precision format since it is commonly used in most scientific and emerging applications.

We introduce a number of micro-architectural and circuit level optimizations to reduce power consumption in the floating-point multiplier (FPM) datapath. A floating-point multiplier consumes significantly more energy compared to a floating-point adder (FPA) [21,24]. This combined with the knowledge that the frequency of floating-point multiplication operations in emerging applications is similar to that of floating-point addition computations makes energy and power optimizations in the FPM datapath highly essential for an

- Overcoming “the tyranny of the clock” is like overcoming the tyranny of fixed word sizes… major increases in speed, energy efficiency.
- The effects compound each other.

**An Operand-Optimized Asynchronous IEEE 754 Double-precision floating-point adder**

**Basit Riaz Sheikh and Rajit Manohar**

*We present the design and implementation of an asynchronous high-performance IEEE 754 compliant double-precision floating-point adder (FPA). We provide a detailed breakdown of the power consumption of the FPA datapath, and use it to motivate a number of different data-dependent optimizations for energy-efficiency. Our baseline asynchronous FPA has a throughput of 2.15 GHz while consuming 69.3 pJ per operation in a 65nm bulk process. For the same set of nonzero operands, our optimizations improve the FPA's energy-efficiency to 30.2 pJ per operation while preserving average throughput, a 56.7% reduction in energy relative to the baseline design. To our knowledge, this is the first detailed design of a high-performance asynchronous double-precision floating-point adder.*

# *Disadvantages of the Unum Format*

- It's new. And it's different. No standards committee has approved it. Introducing it is "like trying to boil the ocean".
- Non-power-of-two alignment. Needs packing and unpacking, garbage collection.
- Tag bit overhead leads to possibility that some workloads would *increase* the total bits for a specified accuracy.
- Unum operations require more logic in processor than floating-point, in exchange for reduced storage/bandwidth/energy/power. (Wait, maybe this should be listed as an advantage!)

# *Test Drive for unums: Quadratic Equation*

- Programmer needs to solve $ax^2 + bx + c = 0$
- Recalling elementary school math, naïvely uses $r_1, r_2 = (-b \pm (b^2 - 4ac)^{1/2})/(2a)$
- But $(b^2 - 4ac)^{1/2}$ might be very close to $\pm b$, resulting in left-digit destruction for one root.

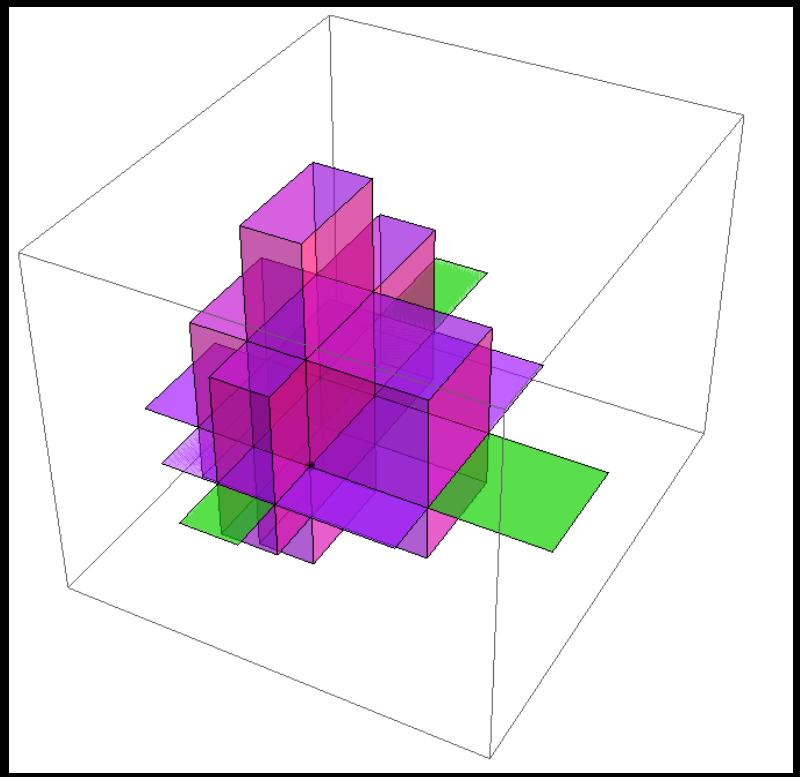
Try unums for $a = 3$, $b = 100$, $c = 2$,
using unum representation.

# *Quadratic Equation Energy and Accuracy Result*

- Input unums take 12, 18, 11 bits; intermediate results take 12 to 45 bits
- Paper estimate: 1442 pJ for unum, 3200 for 32-bit float (55% less energy)
  - Correct answer to nine places: –0.0200120144…
  - Answer with 32-bit floating point: –0.02001167
  - Answer with unums: $-0.020012014 \pm 3\times10^{-9}$

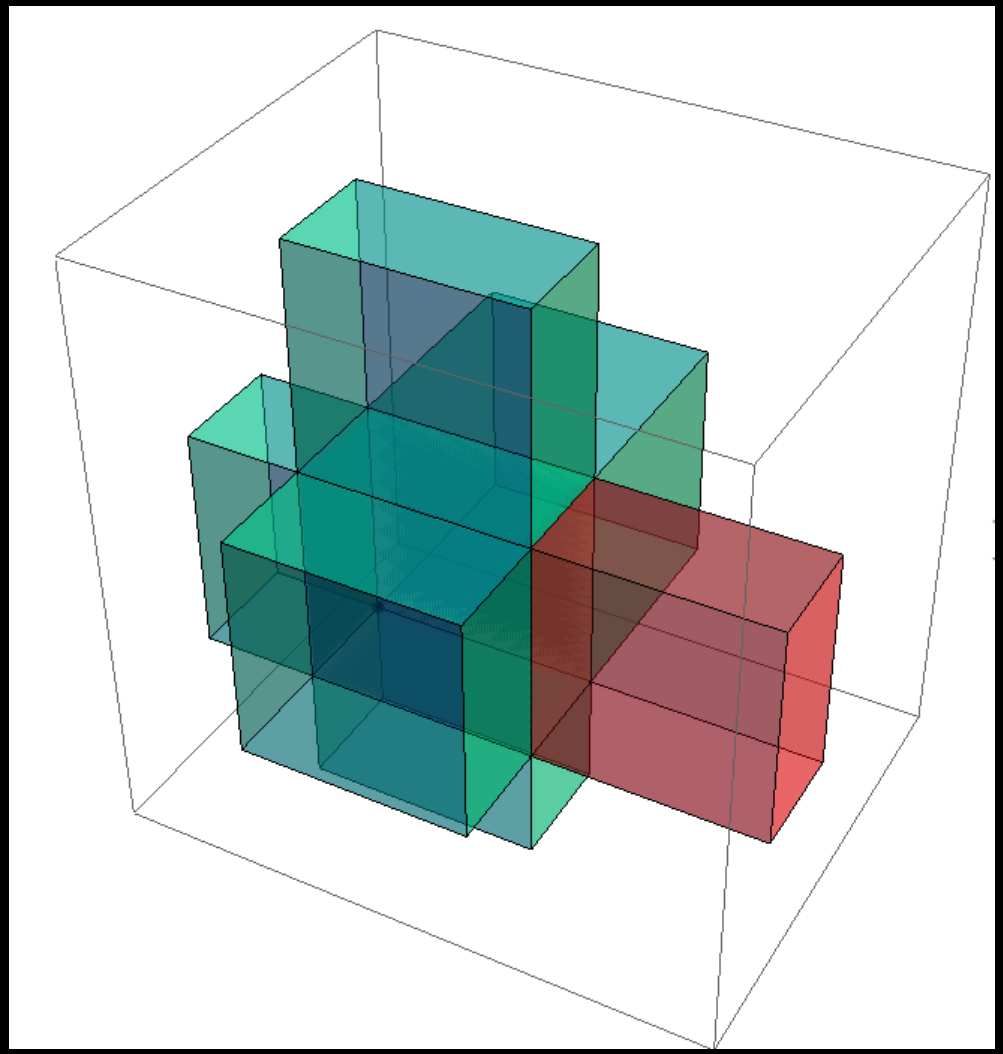
- Rigorous bound on result.
- More accurate result.
- Less energy used.
- Less storage and bandwidth used.
- Loss of accuracy is *part of the answer.*
- Much more like the way people compute with pencil and paper, but without the programmer having to think about precision.

- What's wrong with floating point
- What's wrong with interval arithmetic
- A possible fix: "unum" representation
- The "ubox" approach
- Interval physics

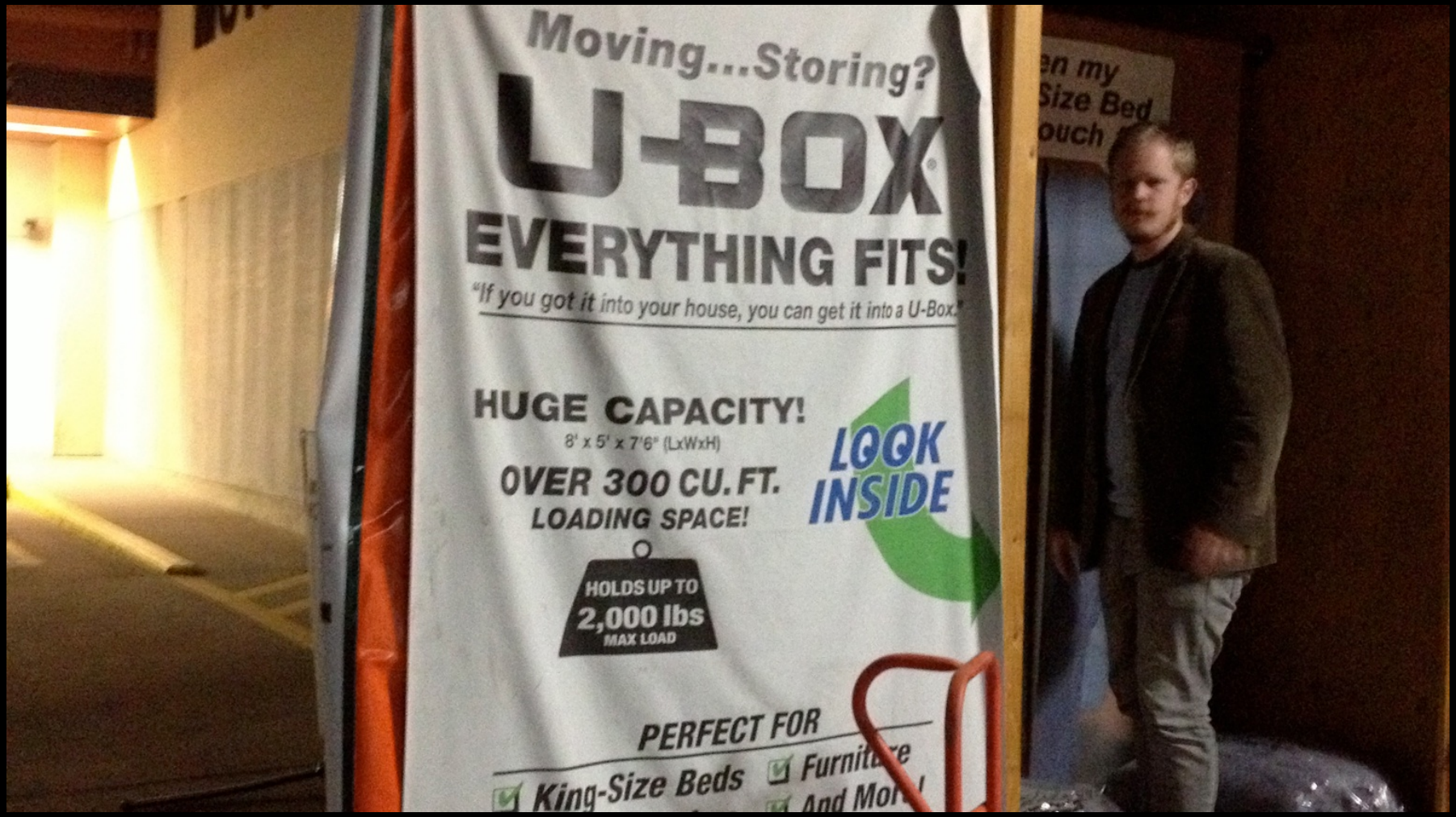


# *Interval version of the unum: The "ubox"*

- A *ubox* is a multidimensional unum, where the number of dimensions is the degrees of freedom in the answer.
- Dimensions are 0 (exact) or a single ULP (inexact).
- *Sets* of uboxes form the best-possible answer with a given amount of precision. All possible answers (green) and none that cannot be in the answer set (red).

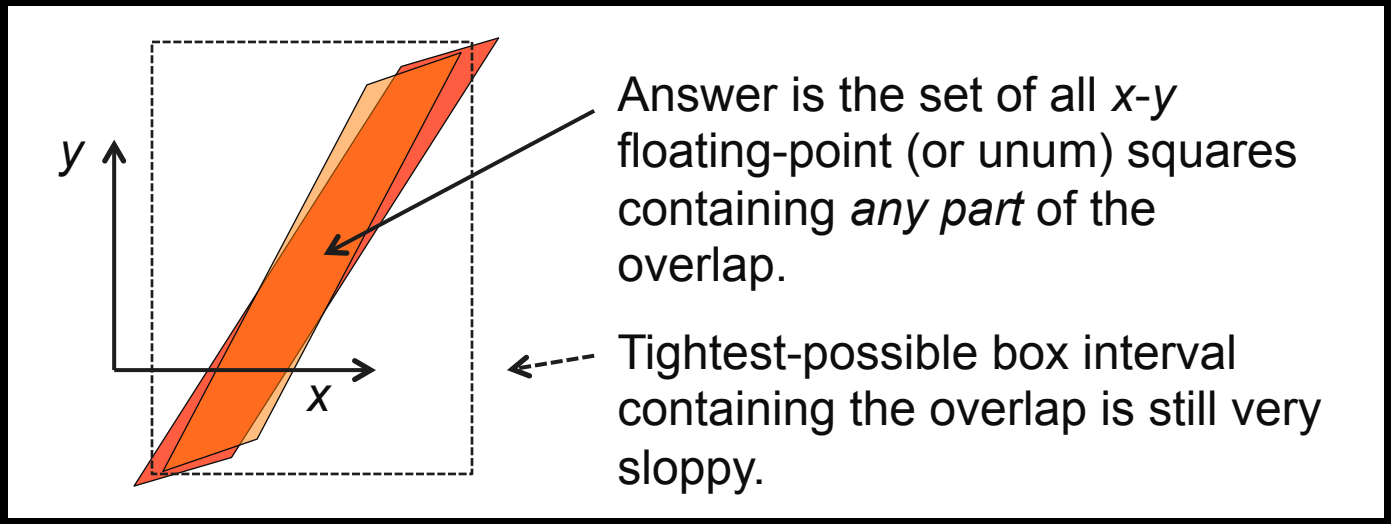


*I hope U-Haul won't mind.*
*This "Ubox" has no hyphen.*



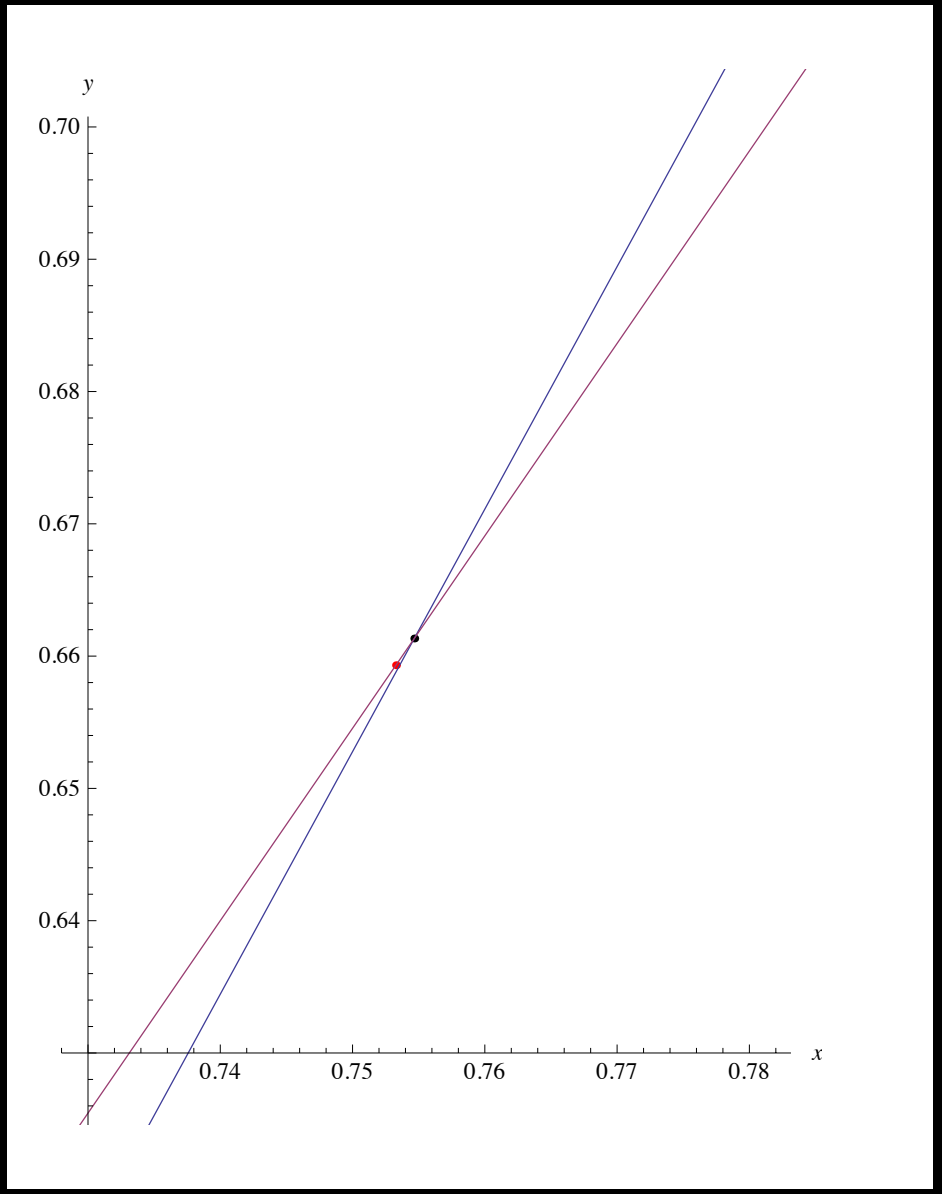
# *Ubox example: Rigorous linear solvers*

- Even 2 equations in 2 unknowns *rigorously* (interval bounds) involves computational geometry… intersecting 8 half-planes (2 parallelograms).
- "Ill-posed" problems much less of a problem with ubox methods!
- Ultimate solution is the minimum "containment set."
- Gaussian elimination with interval values leads to VERY sloppy (usually useless) bounds!



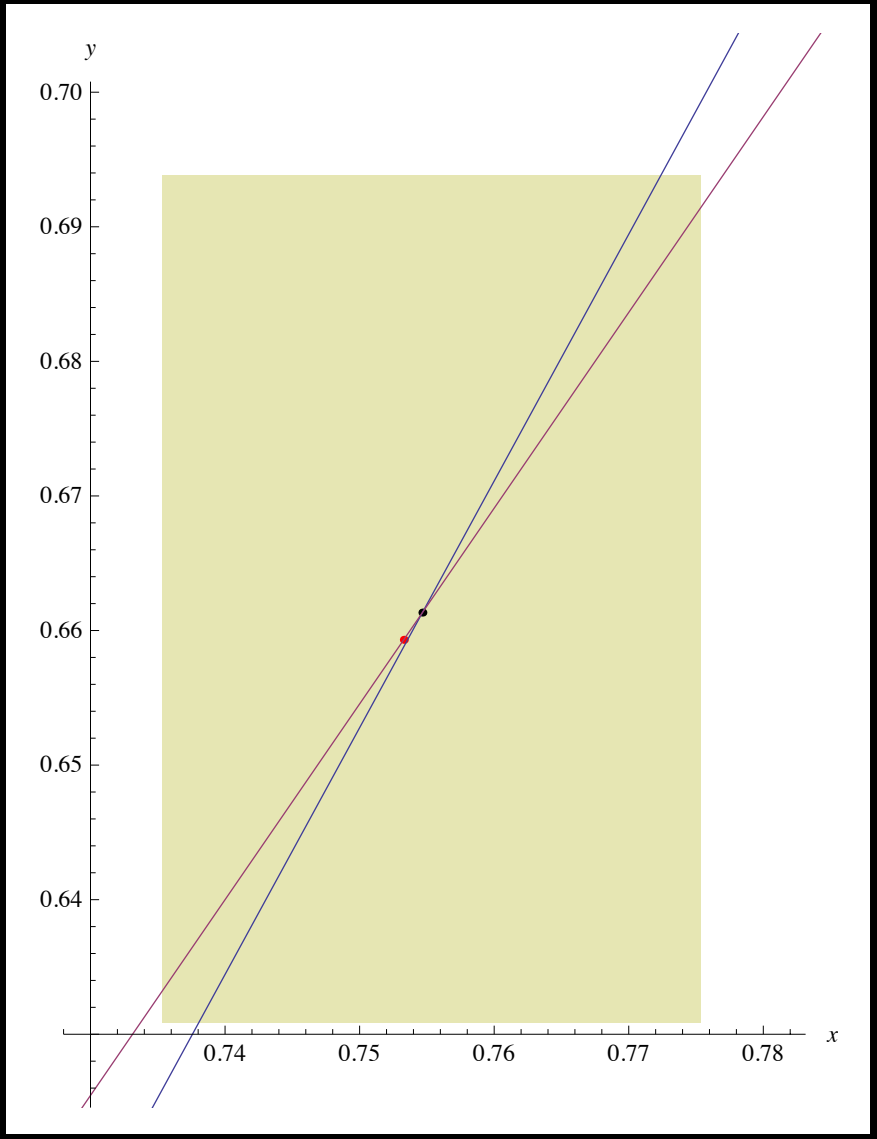
# *2 equations in 2 unknowns as an intersection problem*

- If the *A* and *b* values in *Ax*=*b* are rounded, the “lines” have width from uncertainty
- Apply a standard solver, and get the red dot as “the answer”, *x*. A pair of floating-point numbers.
- Check it by computing *Ax* and see if it rigorously contains *b*. Yes, it does.
- Hmm… are there any other points that also work?

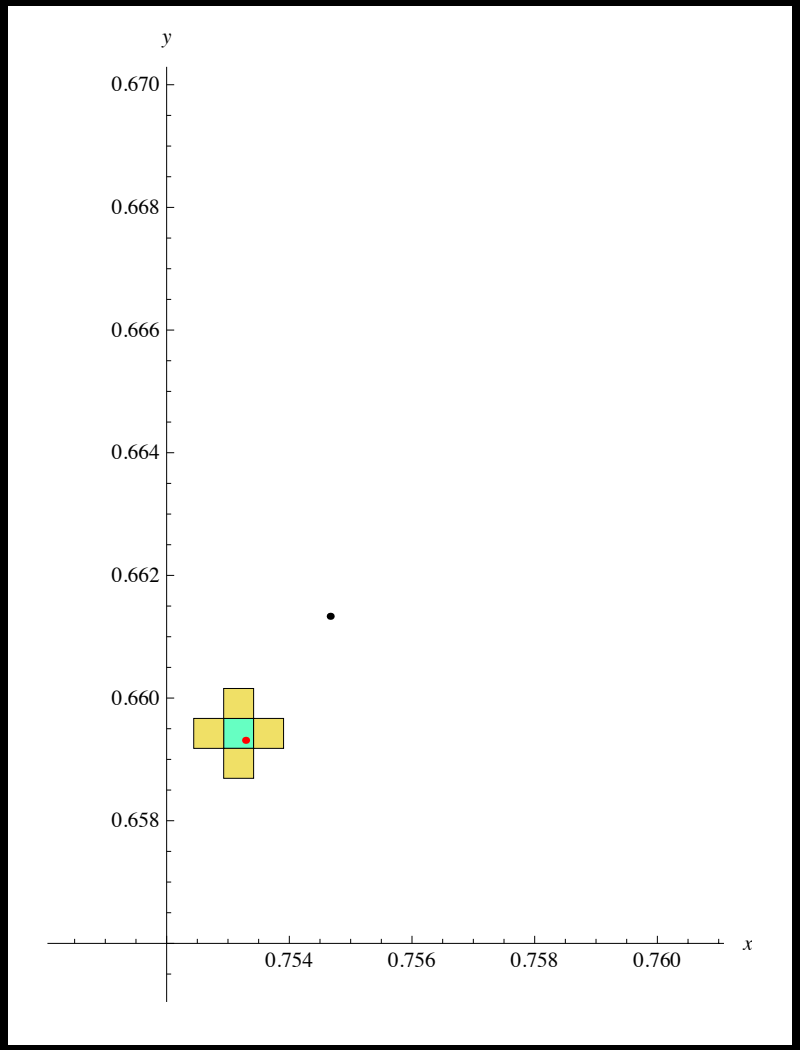


# *The yellow rectangle is what you get with interval arithmetic*

The “correlation problem” is a killer for Ax=b problems.

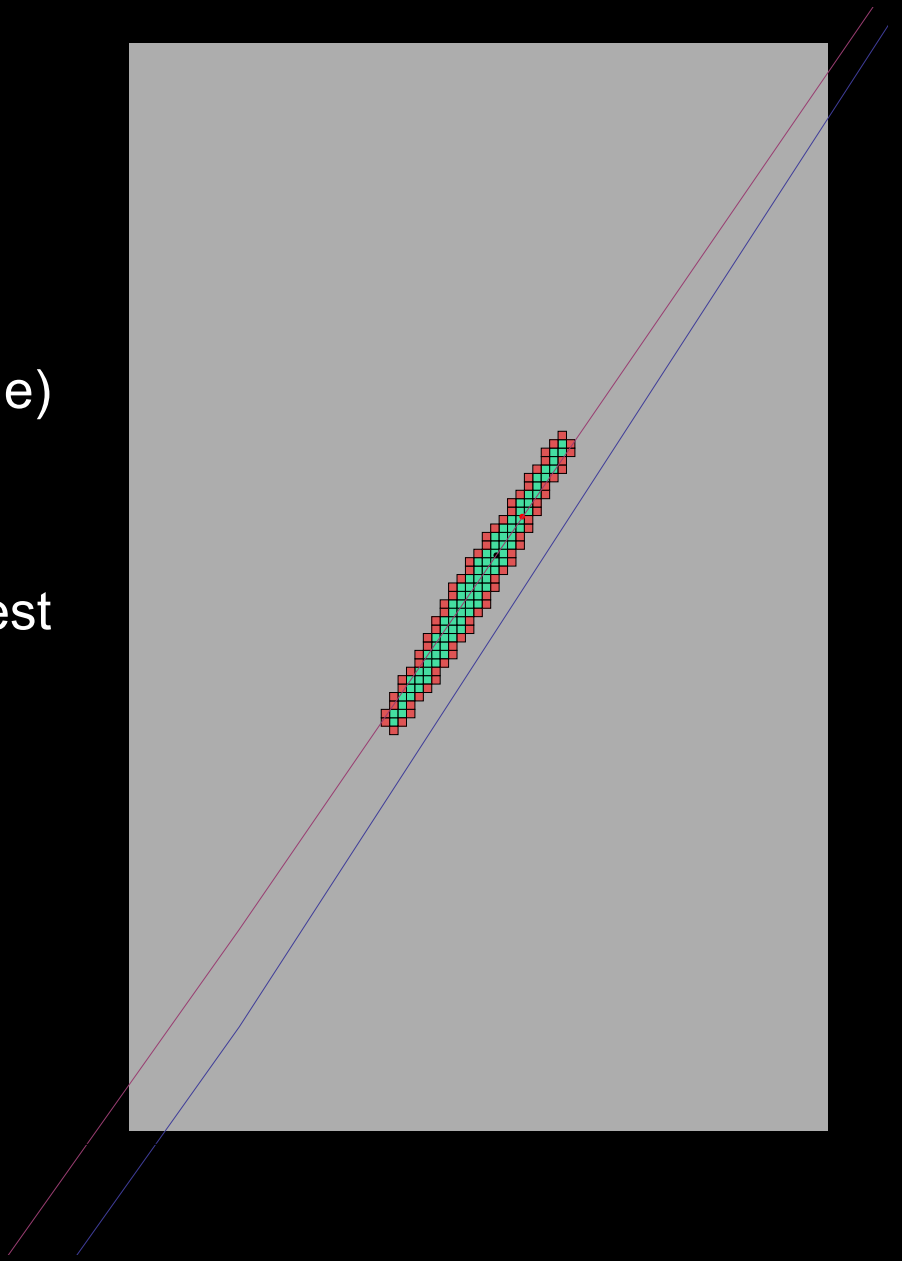


# *With one ubox that works, try its neighbors. (It's like a 'paint bucket' fill-in algorithm.)*



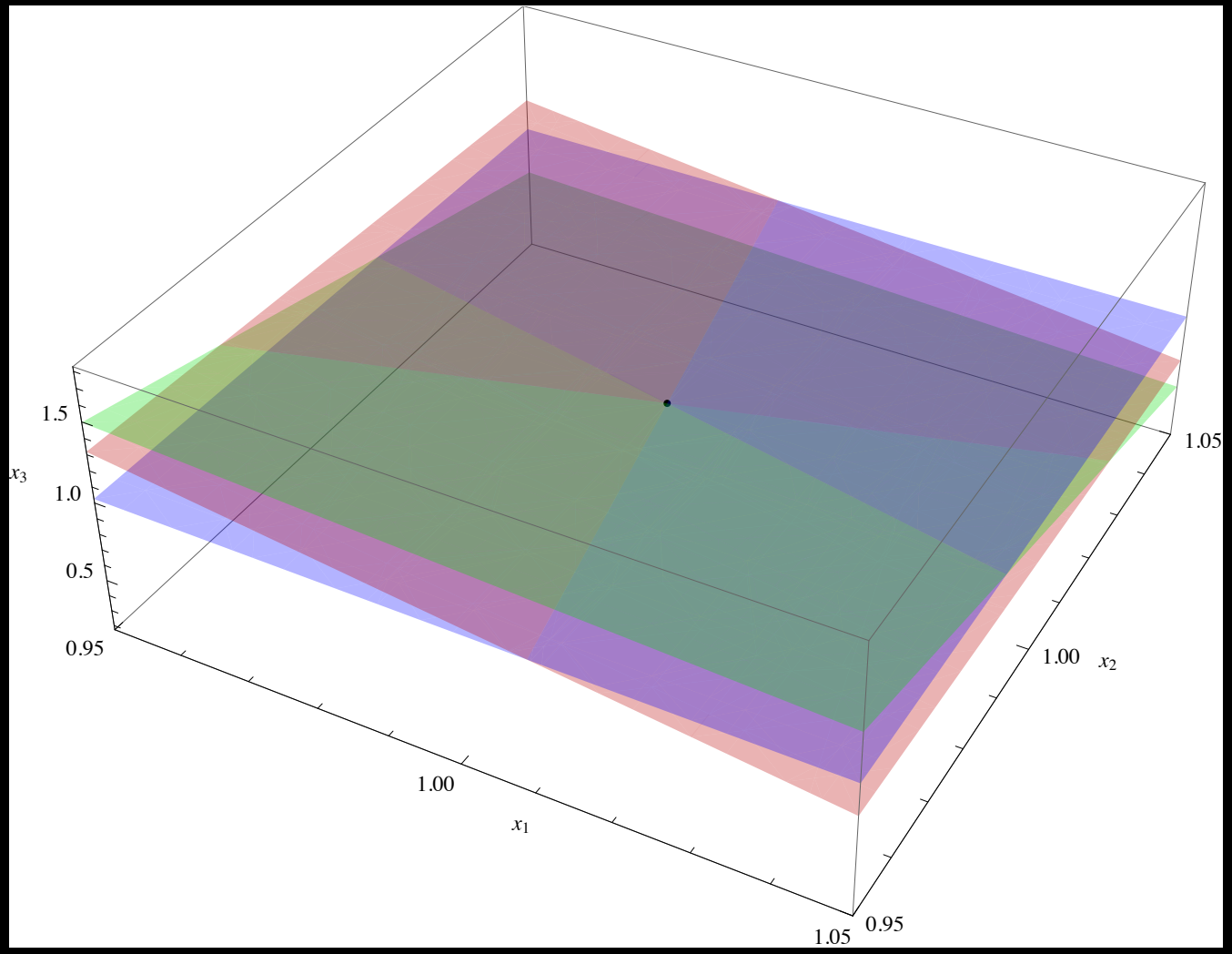
# *Ubox, Floating Point, and Interval Solutions*

- The point solution (black dot) just gives *one of many* solutions, and disguises the instability of the answer
- The interval method (gray rectangle) yields far too loose a bound to be useful
- The ubox method (green) is the best you can do for a given precision
  - Only now I'm using *more* storage
  - …and doing more computation
  - …but a lot less thinking about numerical analysis!

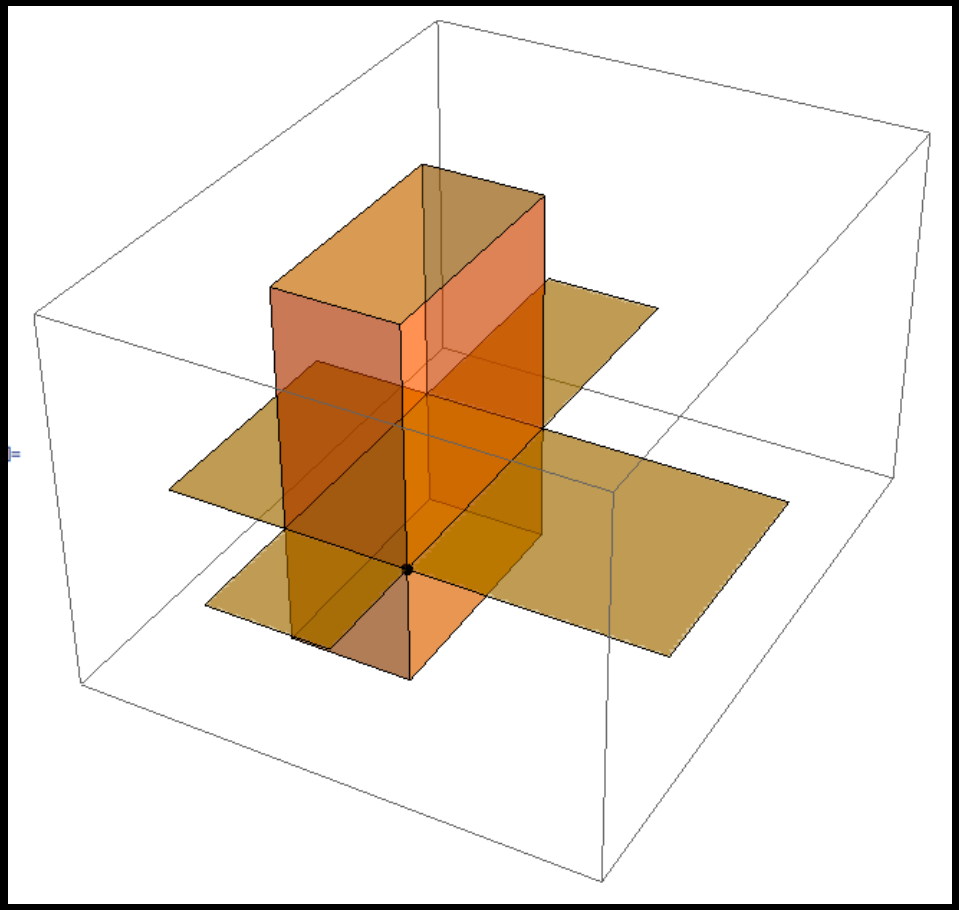


# *Try ubox method on a 3-equation Ax = b*

Solving three simultaneous linear equations is equivalent to finding the intersection of three planes.



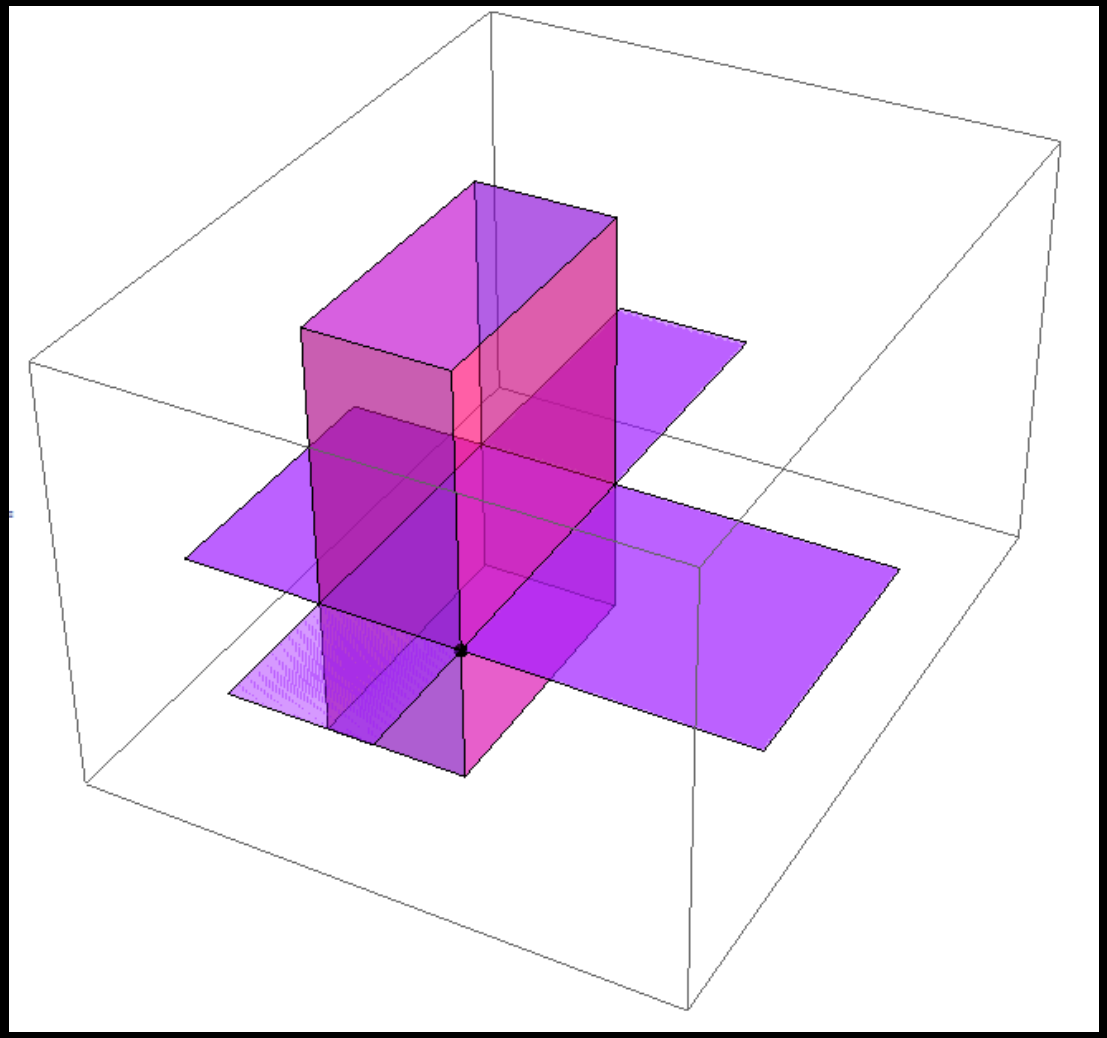
# *Assume every input in Ax = b is an interval 1 ULP wide. Intersecting slabs.*



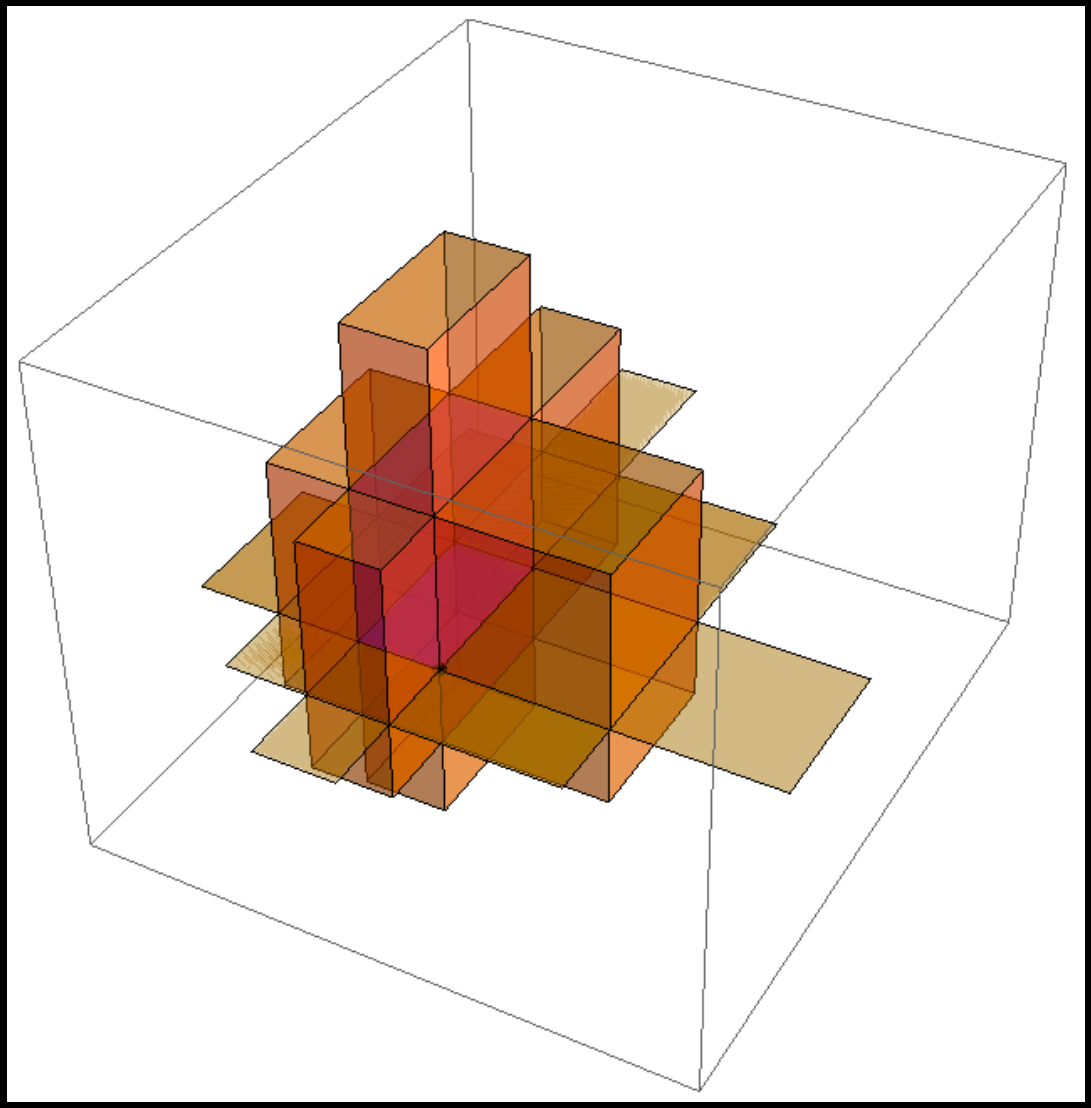
Find ubox containing x.
Find neighbors of that ubox.

(Some answers just happened to be exact, so zero in that dimension. Flat uboxes.)

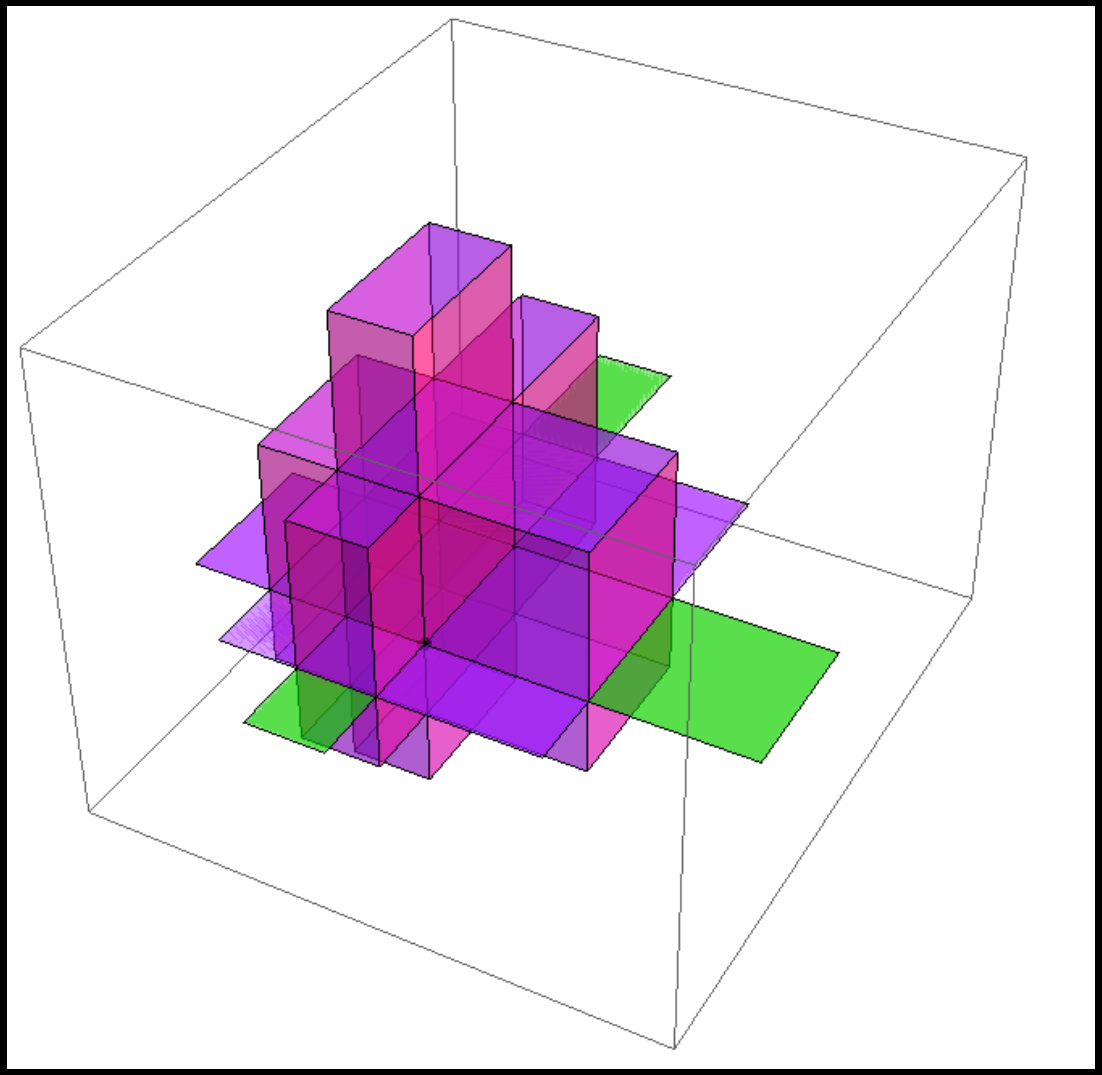
# *These all check out. Mark as OK.*



# *Find candidates of new solution uboxes*

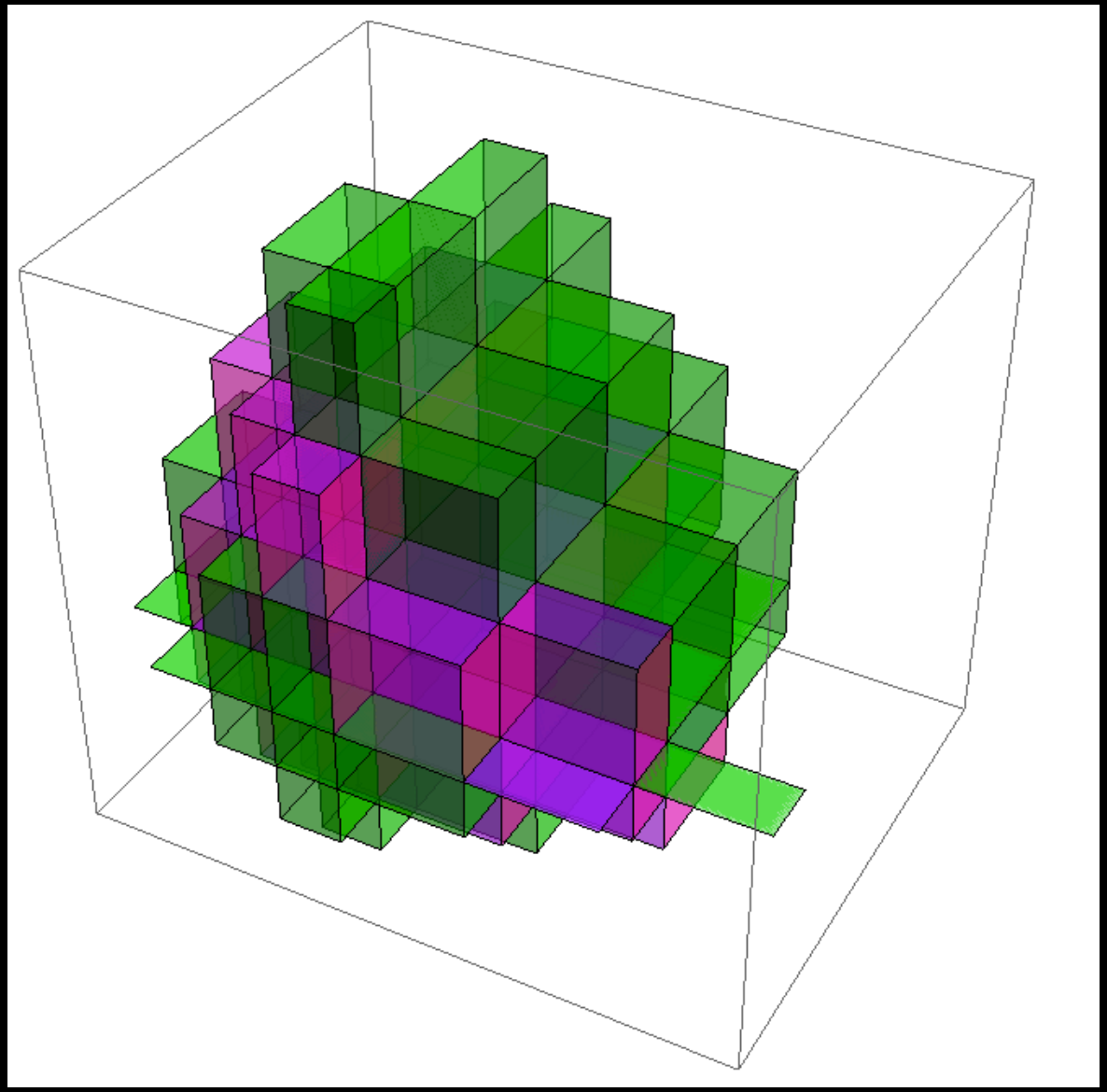


# *Some completely fail Ax = b check*



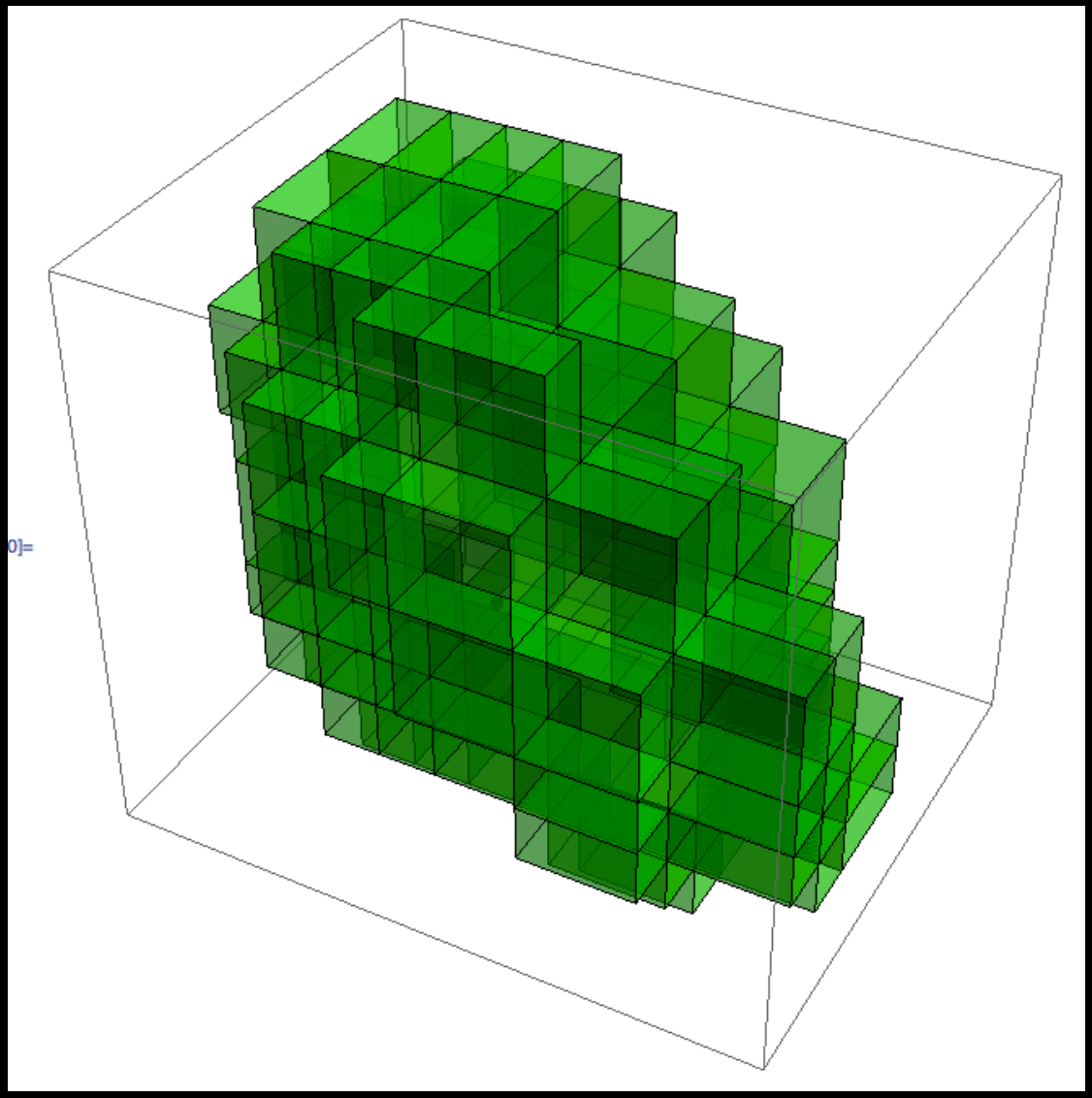
Green uboxes are marked as boundary and are not tested again.

# *After a few iterations*

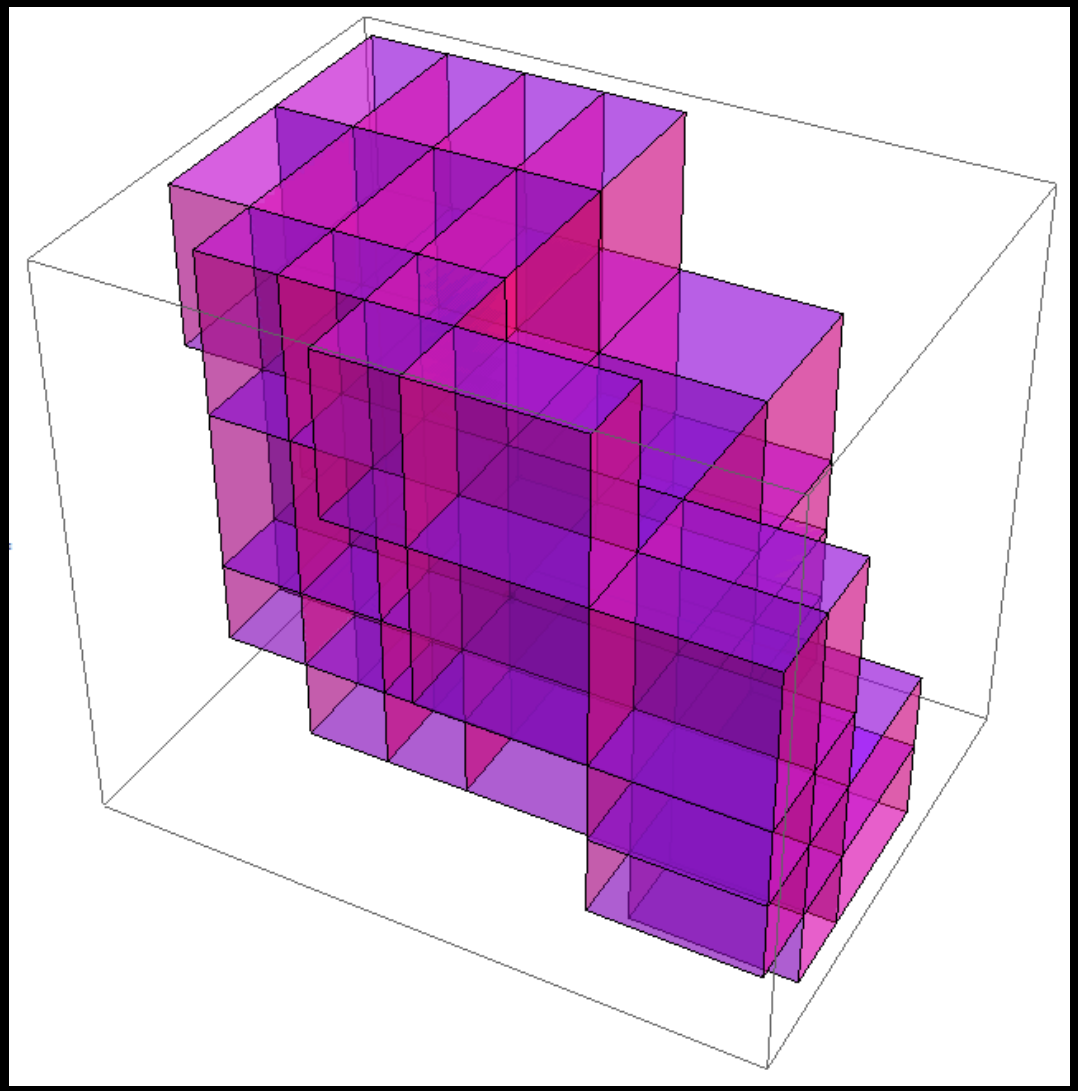


(Note: this example uses magenta as “intersects answer” and green as “completely outside answer.” Later I switched to go light and stop light colors.)

# *Final shell enclosing answer*

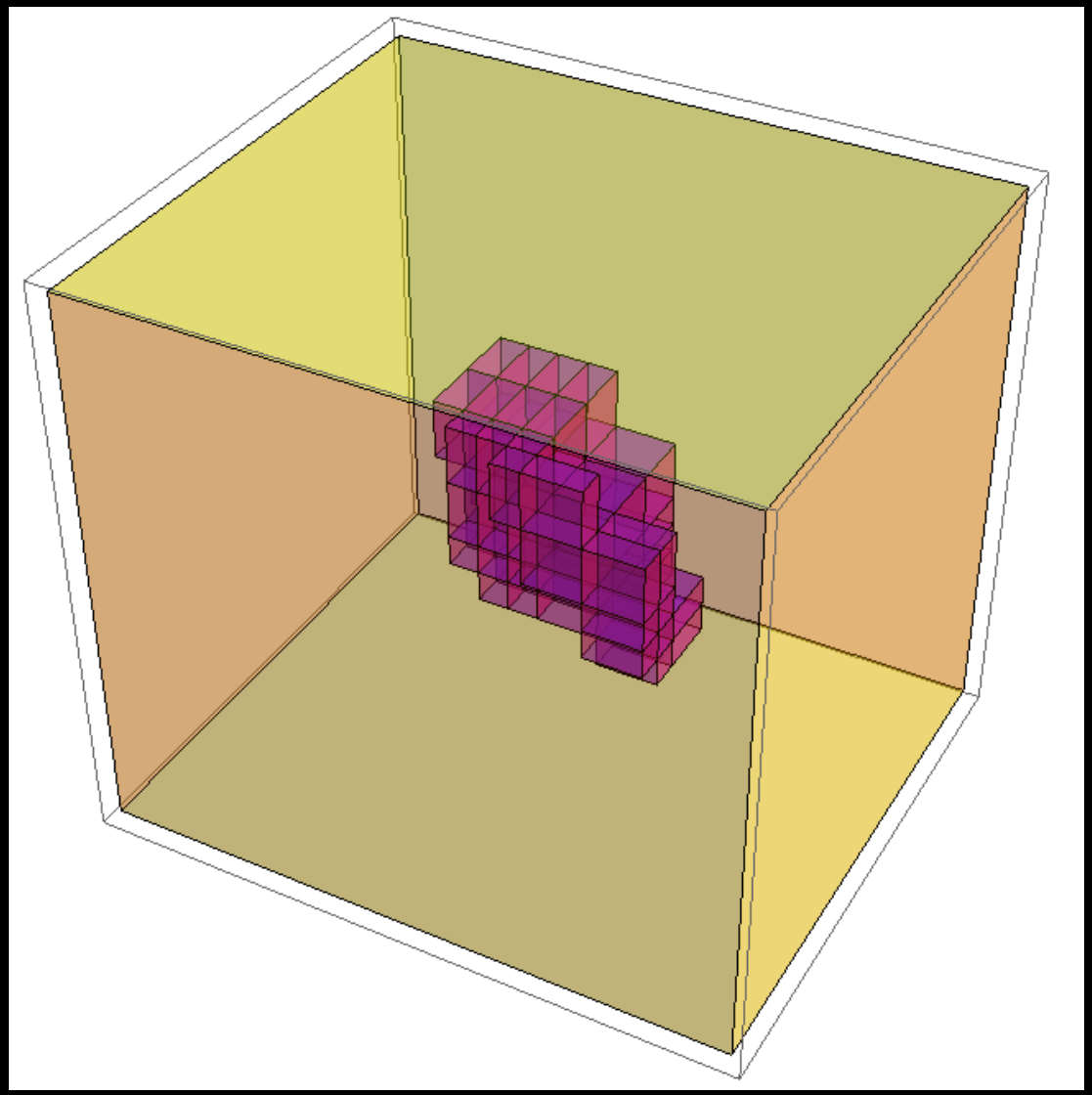


# *The complete ubox-accurate answer*



This is every ubox in *x* for which *Ax* intersects *b*, and there are no uboxes that fail to intersect *b*.

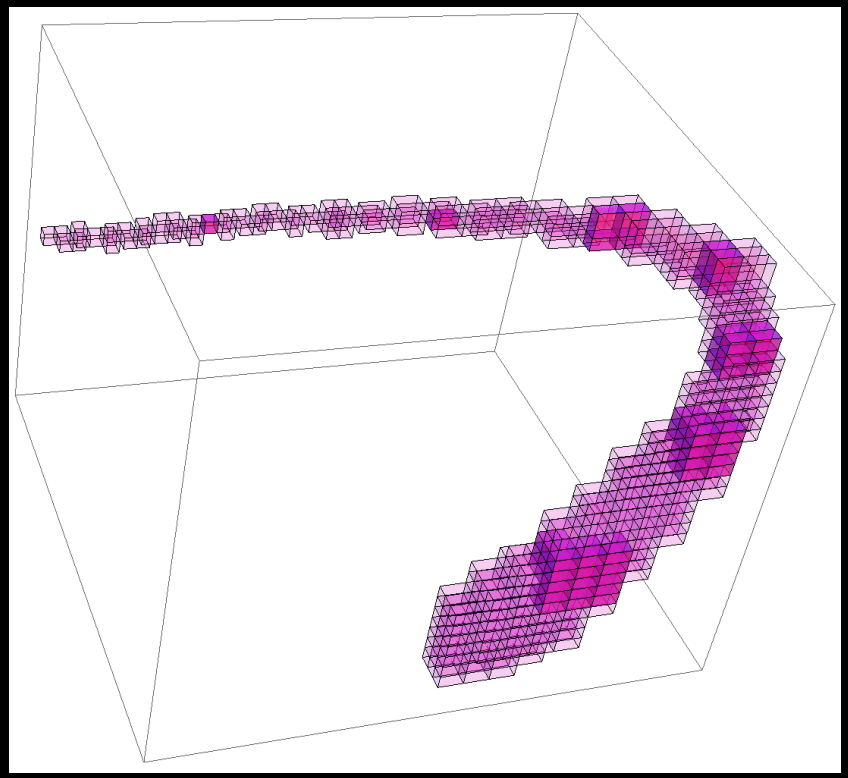
# *Compare with naïve interval method*



The yellow box is the interval bound by applying, say, Gaussian elimination to the interval version of *A*.

Getting good answers starts to look like a “big data” problem! Lots of parallelism to exploit, and an easily-selectable tradeoff between memory and answer quality.

- What's wrong with floating point
- What's wrong with interval arithmetic
- A possible fix: "unum" representation
- The "ubox" approach
- Interval physics



# *Example Definition of "Quality"*

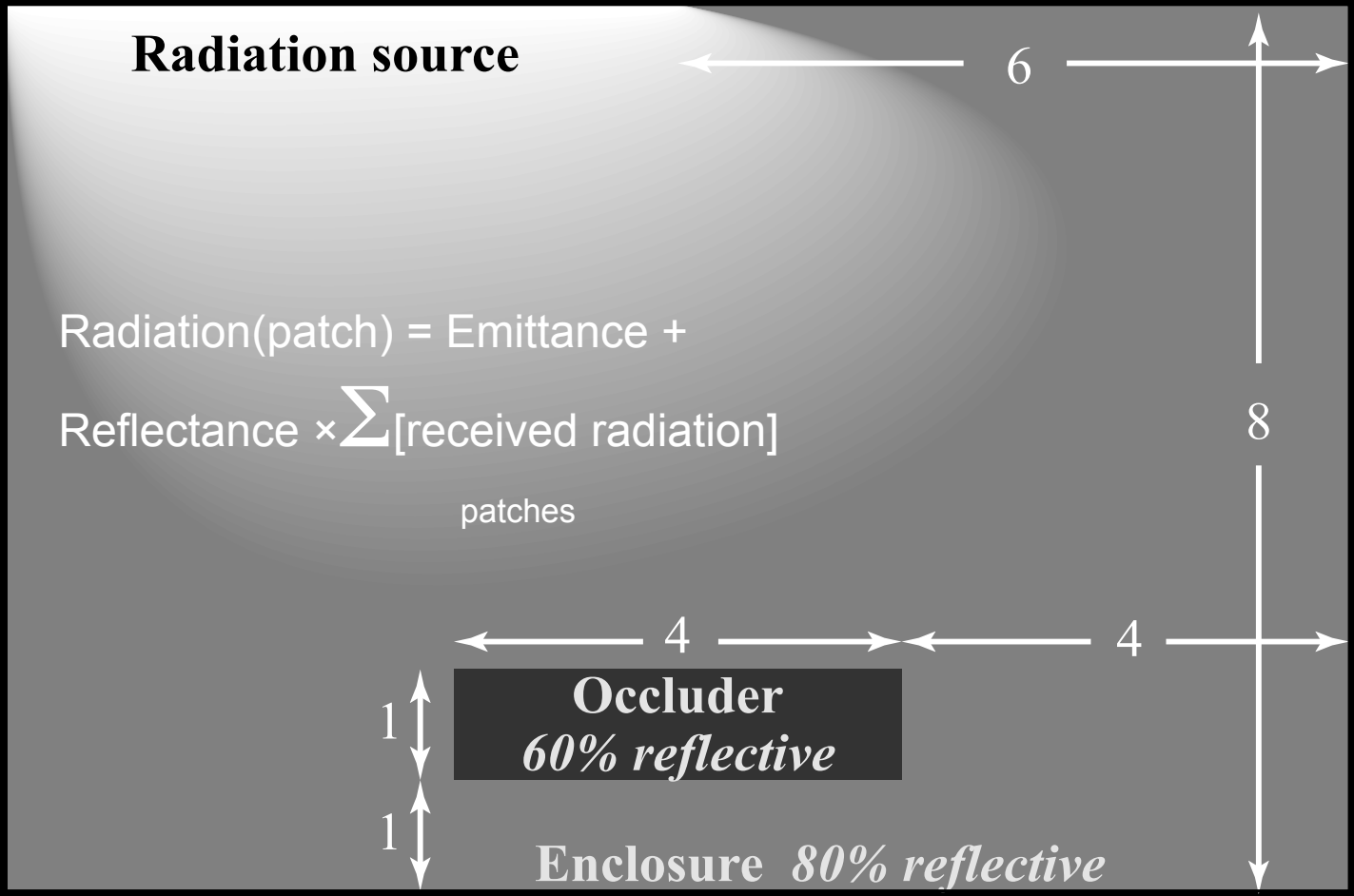
- For a physical simulation that computes $F(x, y, z, t)$ on a domain $D$, bound the result *rigorously* by $F^-$ and $F^+$. Then define total error $E$:

$$E = \int\int\int\int_D \left( F^+ - F^- \right) dx\,dy\,dz\,dt$$

- The answer quality is then $\boldsymbol{Q = 1 / E}$.
- Can do this for *n*-body problems, structural analysis, radiation transfer, and Laplace's Equation… Bounds can come from conservation laws, causality, etc.
- Or just add up the ubox volumes!

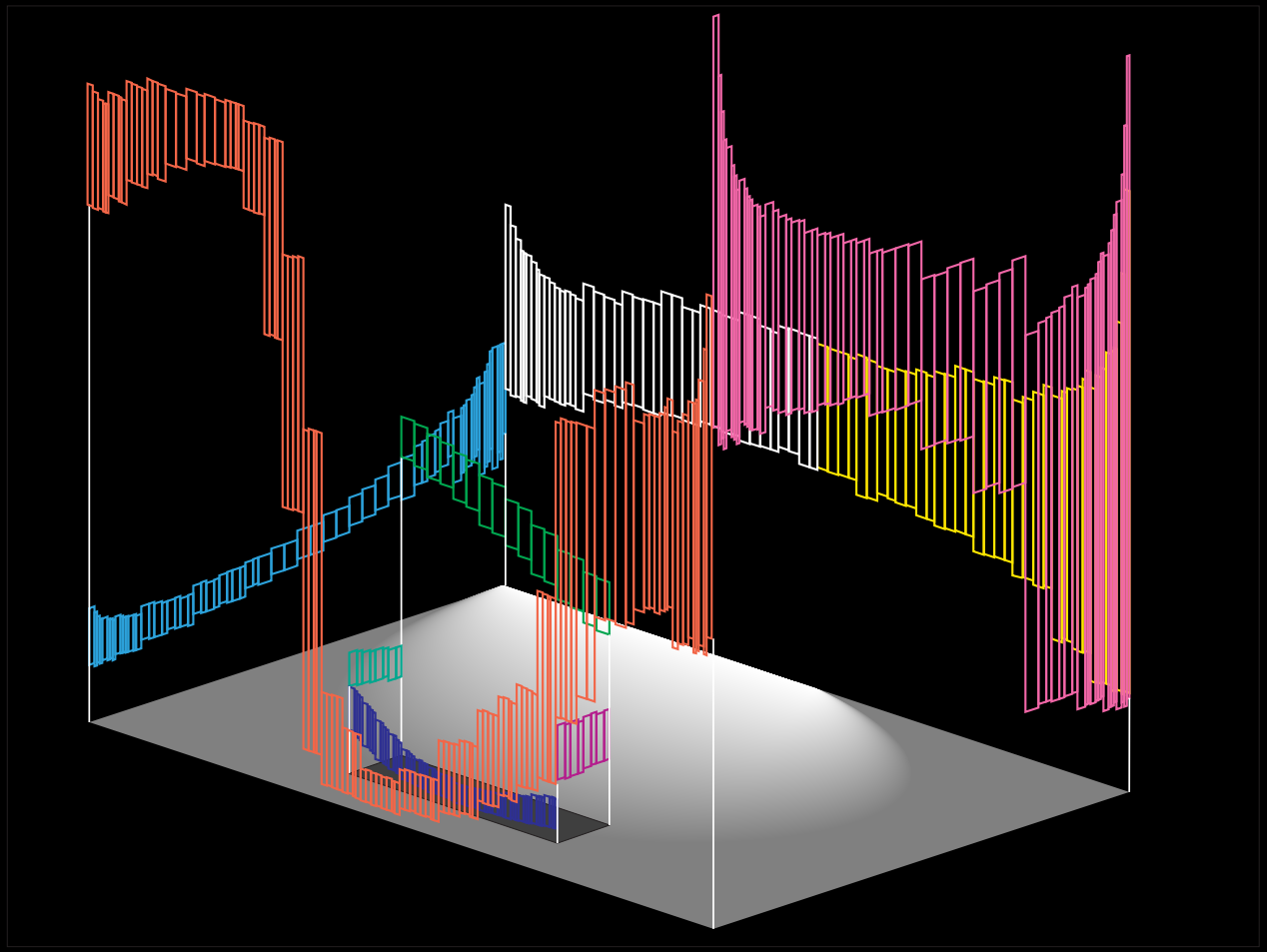
# Radiation Transfer Example

- Use a 2D problem for easier visualization:



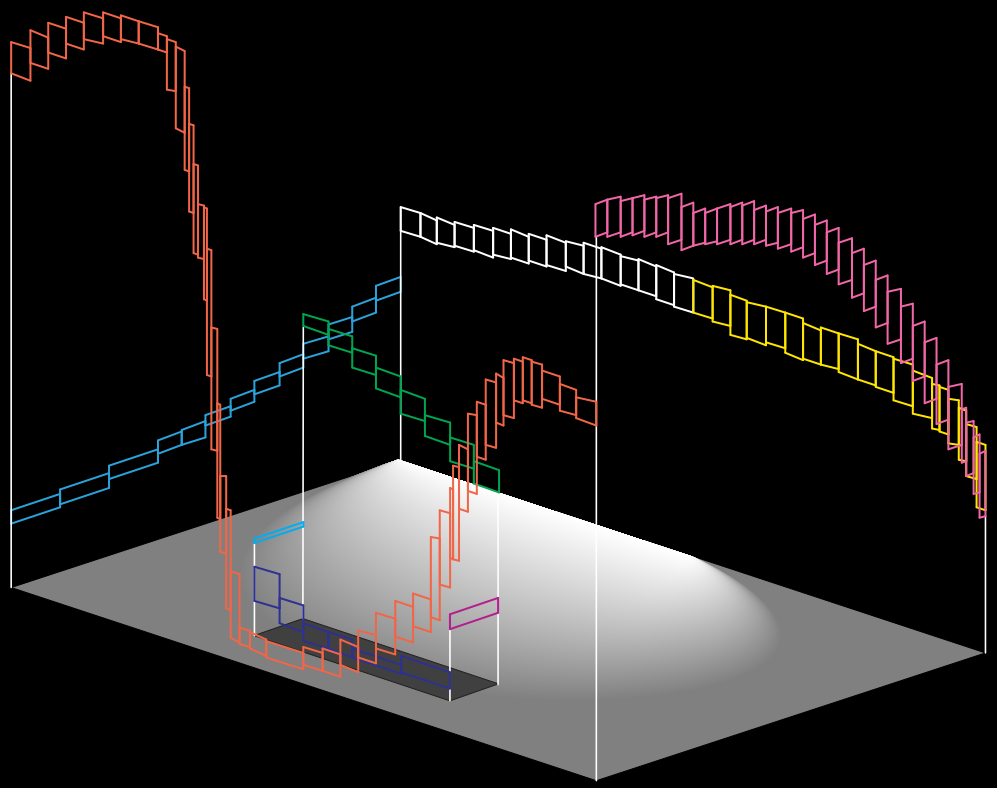
# *Low-Quality Solution*

- Third dimension is $F^+$ and $F^-$ for each surface

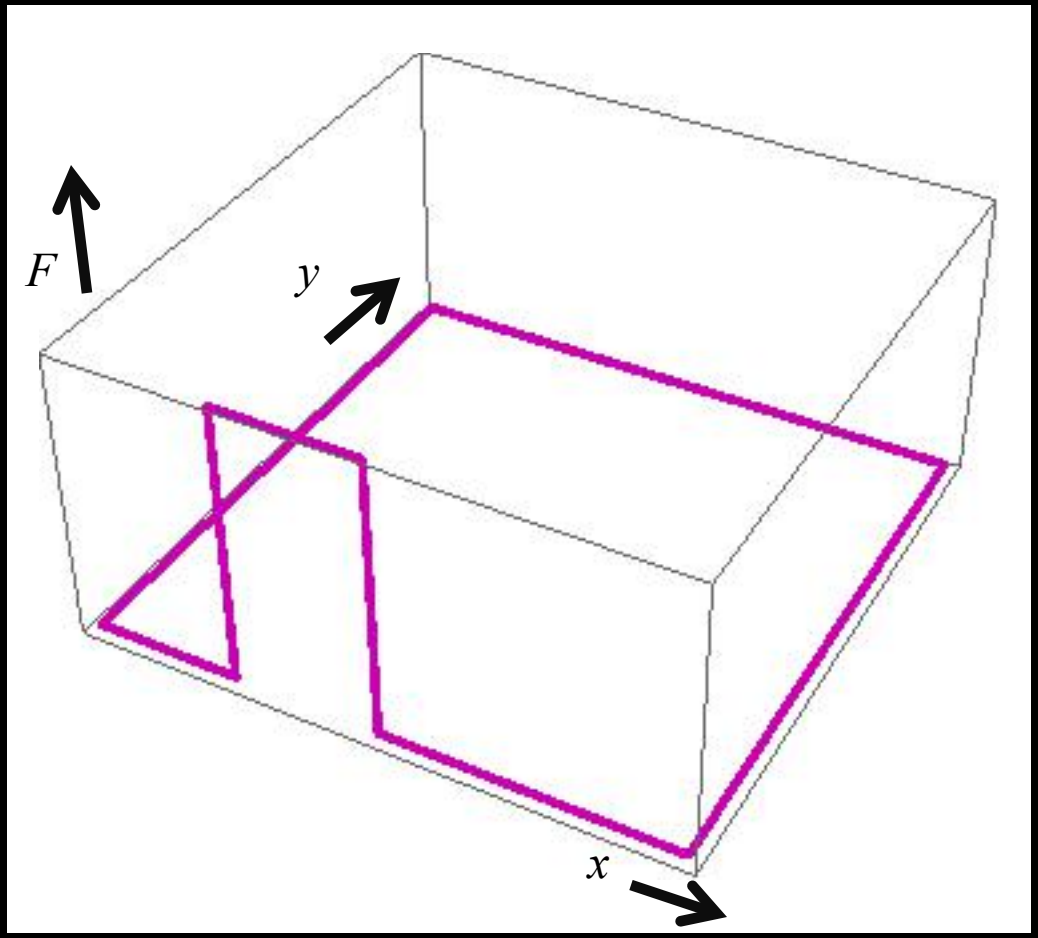


# *Better Quality Solution*

Note: All processors can update these *asynchronously*



# *Example: Laplace's Equation*

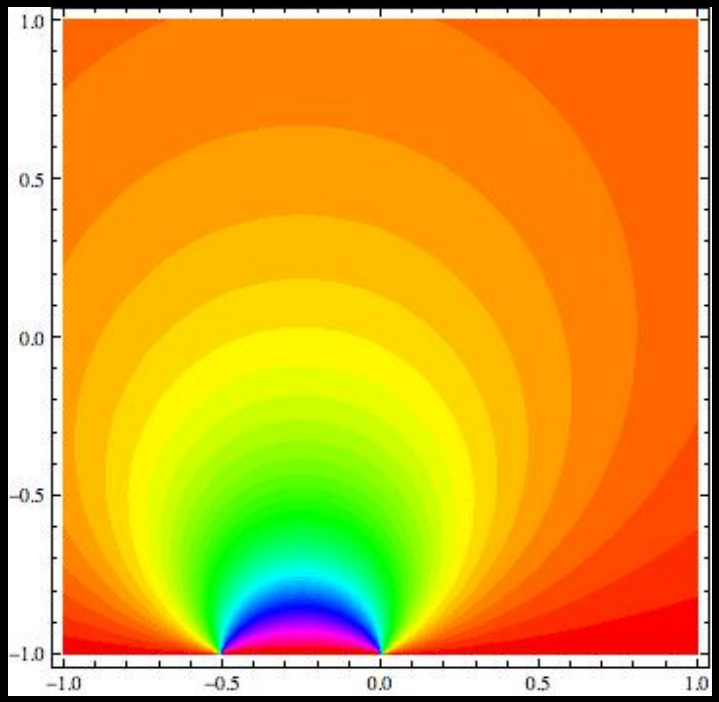



- Magenta line specifies boundary condition.
- Inside the unit square,

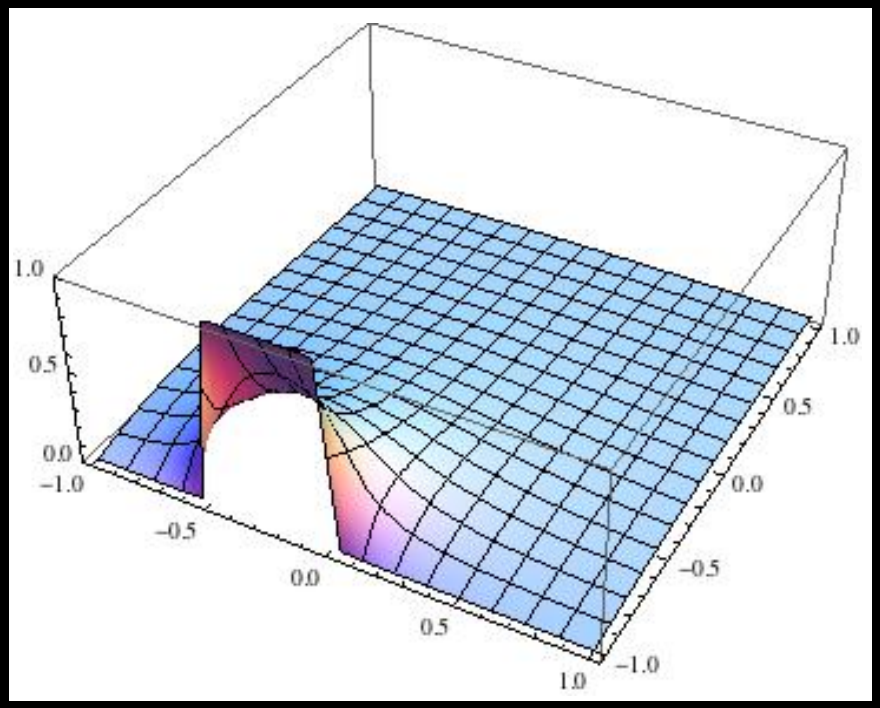
$$\nabla^2 F = 0$$

- (Classic problem for relaxation methods, but multigrid has lowest arithmetic complexity.)

# *Laplace's Solvers: Which is Better?*



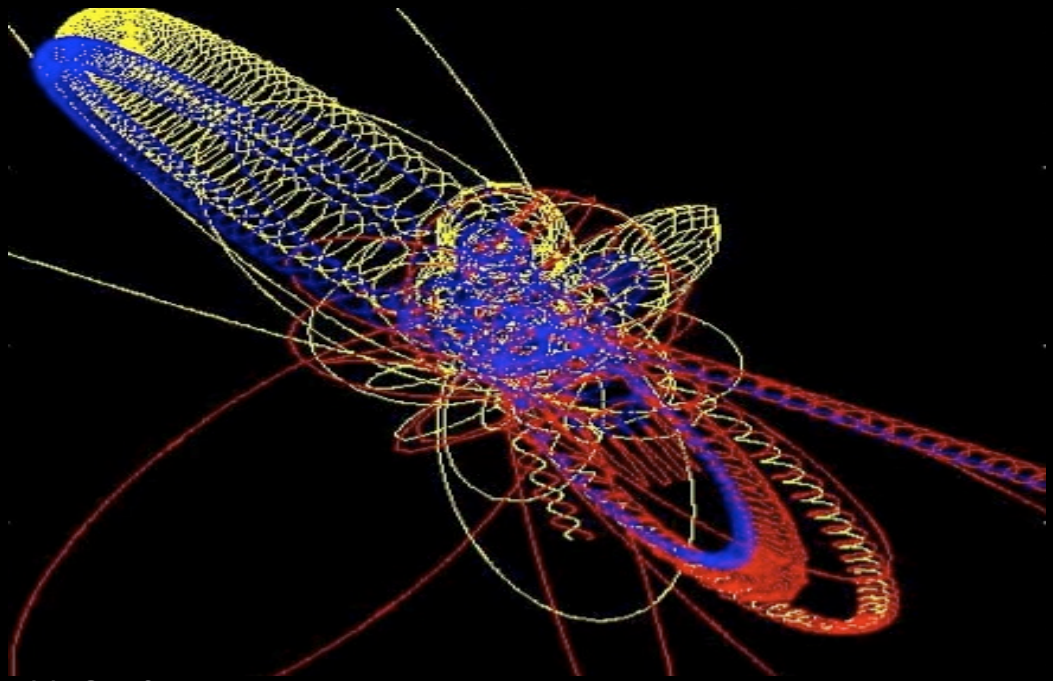

64-bit multigrid floating point method *seems* to have converged. 15 decimals, some of them probably correct. Mostly.




Ubox arithmetic provably bounds answer to 4 decimals, uses half the storage and bandwidth and energy.

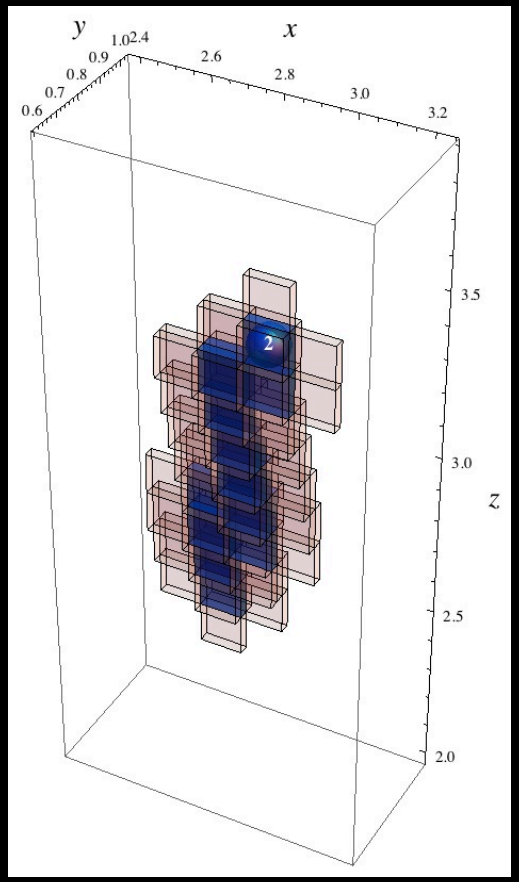
# *Even the 3-Body Problem is Massively Parallel*

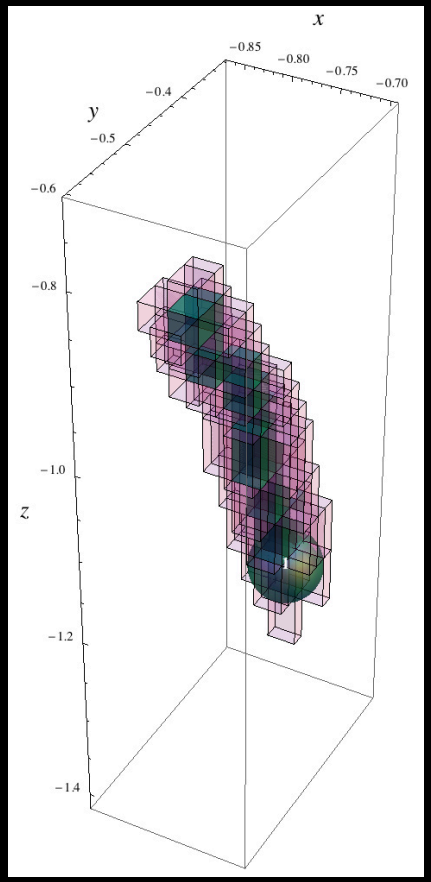
- Appears “Embarrassingly Serial” with only 18 variables, yet simulation involves a huge number of serial steps.
- However: each step produces an irregular containment set. *Use all available cores to track members of the set.*
- Far more ops per data point. Billions of cores usefully employed. Provable bounds on the answer.

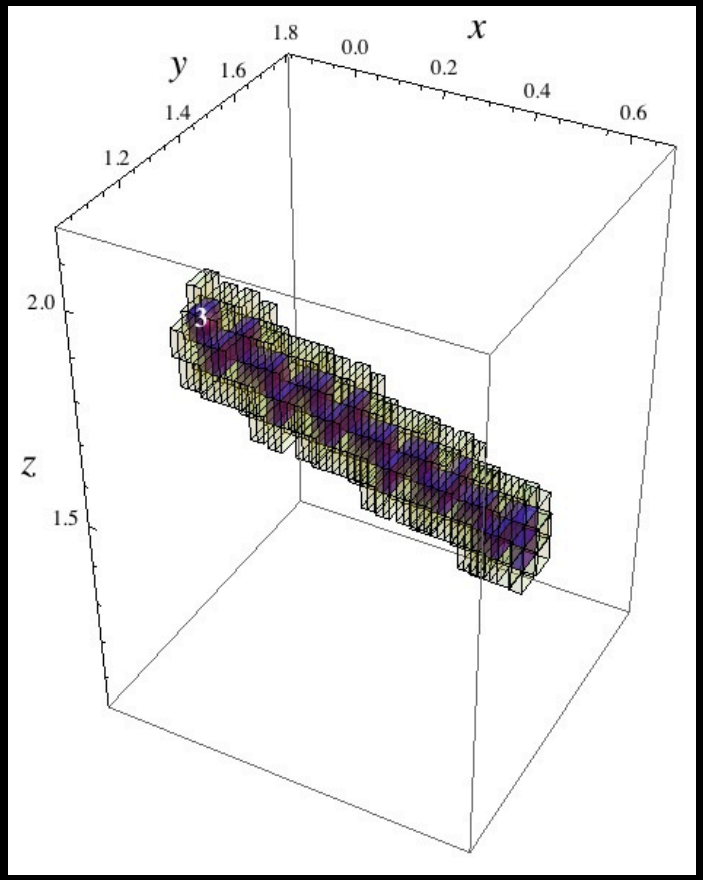


# *A rigorous, bounded 3-body simulator is possible using 8-bit floating-point*

- Sign-bit, 3-bit exponent, 4-bit mantissa; IEEE rules
- Coded in Mathematica for now. Need a fast, native version…



# *Summary*

- A range of techniques exist for reducing bandwidth/storage/ energy/power requirements of numerical storage.
- The most dramatic savings come from *changing the numerical format itself*. Unums often solve the associativity problem of floats, and ubox sets often solve the correlation problem of intervals.
- Combining async design with unum flexible precision could give over 4x efficiency improvement, maybe the key to getting an exascale computer built earlier than 2020.
- The unum approach could almost eliminate the need for numerical analysis in many algorithms. Let's make computer math much smarter than what it was in 1914!

# *If you like this stuff, here's where to learn more:*

Web-based information that is accessible for non-experts, and even entertaining to read:

http://floating-point-gui.de/

http://en.wikipedia.org/wiki/Kahan_summation_algorithm

http://www.cs.berkeley.edu/~wkahan/Mindless.pdf

http://ta.twi.tudelft.nl/users/vuik/wi211/disasters.html

http://en.wikipedia.org/wiki/IEEE_754-2008