Posit Arithmetic

John L. Gustafson
10 October 2017

1 Overview

Unums are for expressing real numbers and ranges of real numbers. There are two modes of operation, selectable by the user: *posit mode* and *valid mode*.

In *posit mode*, a unum behaves much like a floating-point number of fixed size, rounding to the nearest expressible value if the result of a calculation is not expressible exactly; however, the posit representation offers more accuracy and a larger dynamic range than floats with the same number of bits, as well as many other advantages. We can refer to these simply as *posits* for short, just as we refer to IEEE 754 Standard floating-point numbers as *floats*.

In *valid mode*, a unum represents a **range** of real numbers and can be used to rigorously bound answers much like interval arithmetic does, but with a number of improvements over traditional interval arithmetic. Valids will only be mentioned in passing here, and described in detail in a separate document.

This document focuses on the details of posit mode, which can be thought of as “beating floats at their own game.” It introduces a type of unum, Type III, that combines many of the advantages of Type I and Type II unums, but is less radical and is designed as a “drop-in” replacement for an IEEE 754 Standard float with no changes needed to source code at the application level. The following table may help clarify the terminology.

Unum	Date Intro–duced	IEEE 754 Compati–bility	Advantages	Disadvantages
Type I	March 2015	Yes; perfect superset	Most bit–efficient rigorous–bound representation	Variable width management needed; inherits IEEE 754 dis–advantages, such as redundant representations
Type II	January 2016	No; complete redesign	Maximum infor–mation per bit (can customize to a particular workload); perfect reciprocals (+ – × ÷ equally easy); extremely fast via ROM table lookup; allows decimal representations	Table look–up limits precision to ~20 bits or less; exact dot product is usually expensive and impractical
Type III	February 2017	Similar; conversion possible	Hardware–friendly; *posit* form is a drop–in replacement for IEEE floats (less radical change); Faster, more accurate, lower cost than float	Too new to have vendor support from vendors yet; perfect reciprocals only for 2^n, 0, and ±∞

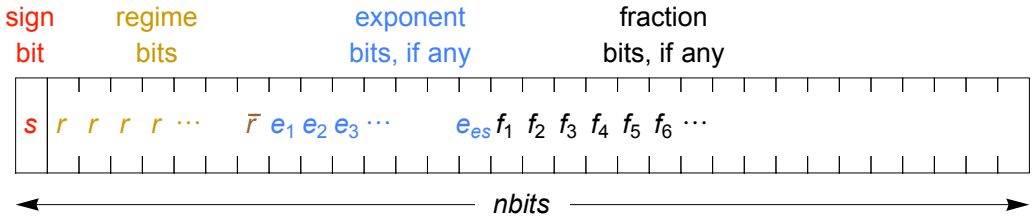
Type III posits use a fixed number of bits, though that number is very flexible, from as few as two bits up to many thousands. They are designed for simple hardware and software implementation. They make use of the same type of low-level circuit constructs that IEEE 754 floats use (integer adds, integer multiplies, shifts, etc.), and take less chip area since they are *simpler* than IEEE floats in many ways. Early FPGA experiments show markedly reduced latency for posits, compared to floats of the same precision.

As with all unum types, there is an *h*-layer of human-readable values, a *u*-layer that represents values with unums, and a *g*-layer that performs mathematical operations exactly and temporarily, for return to the *u*-layer. Here, the *h*-layer looks very much like standard decimal input and output, though we are more careful about communicating the difference between exact and rounded

values. The *u*-layer is posit-format numbers, which look very much like floats to a programmer. The *g*-layer represents temporary (scratch) values in a *quire*, which does exact operations that obey the distributive and associative laws of algebra and allow posits to achieve astonishingly high accuracy with fewer bits than floats. It's a lot to explain at once, but let's start with the *u*-layer since it is the closest cousin to floating-point arithmetic.

1.1 Brief description of the format for posit mode, Type III unums

A posit for Type III unums is designed to be hardware-friendly, and looks like the following:



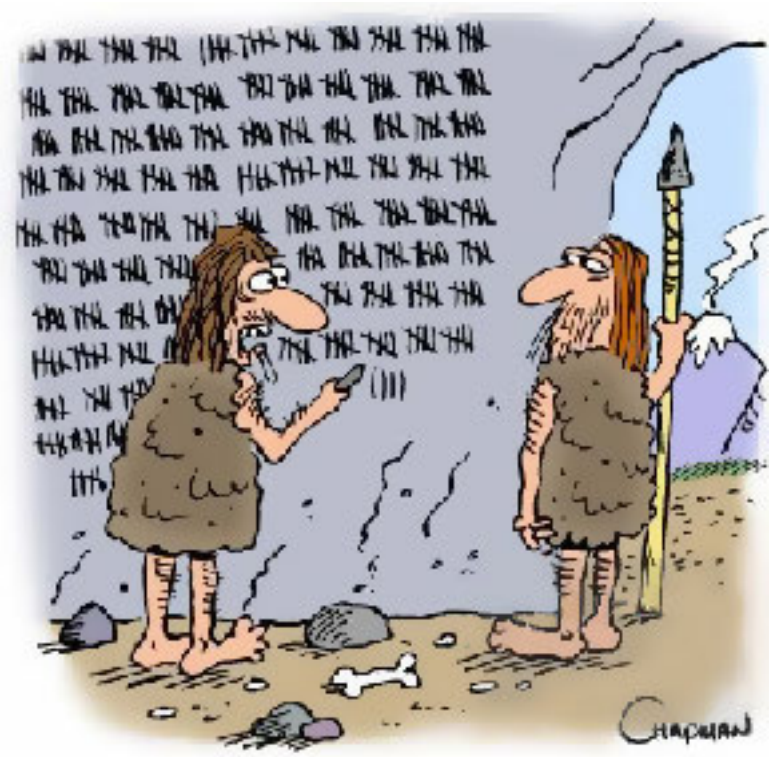
The boundary between regions is only shown after the sign bit, because the other boundaries vary with the size of the *regime.* The regime is the sequence of identical bits r, terminated by the opposite bit $\bar{r}$ or by the end of the posit. This way of representing integers is sometimes referred to as “unary arithmetic” but it is actually a bit more sophisticated. The most primitive way to record integers is with marks, where the number of marks is the number represented. Think of Roman numerals,

1 = I, 2 =II, 3 =III, but then it breaks down and uses a different system for 4 = IV.

Or in Chinese and Japanese,

1 = 一, 2 = 二, 3 = 三, but then the system breaks down and uses a different system for 4 = 四.

The situation we see this system going beyond 3 is *tally marks*, the way countless cartoons depict prisoners tracking how many days they’ve been in prison. But how can tally marks record both positive, zero, and negative integers? We can imagine early attempts to notate the concept:



"I've completely lost track. Is it 1 million or 2 million notches to A.D.?"

But we have the ability to use *two* kinds of "tick mark": zero and one. This means we can express both positive *and negative* integers by repeating the mark. Like, negative integers by repeating **0** bits and zero or positive integers by repeating **1** bits.

The number of exponent bits is *es*, but the number of those bits can be less than *es* if they are crowded off the end of the unum by the regime bits. For people used to IEEE floats, this is usually the part where confusion sets in. “Wait, exactly *where* are the exponent bits, and *how many* are there?” They do **not** work the same way as the exponent field of a float. They move around, and they can be clipped off. For now, think of the regime bits and exponent bits as together serving to represent an integer *k*, and the scaling of the number is 2^k.

The fraction bits are whatever bit locations are not used by the sign, regime, and exponent bits. They work exactly like the fraction bits for normalized floats. That's the main thing that makes Type III posits hardware-friendly. Just as with normalized (binary) floats, the representation boils down to

$$(-1)^{\text{sign}} \times 2^{\text{some integer power}} \times (1 \text{ plus a binary fraction})$$

Hardware designers will be happy to hear that there are no “subnormal” posits like there are with floats, that there is only one rounding mode, and there are only two exception values (0 and ±∞) that do not work like the above formula.

The reason for introducing regime bits is that they automatically and economically create *tapered accuracy*, where values with small exponents have more accuracy and very large or very small

numbers have less accuracy. (IEEE floats have a crude form of tapered accuracy called *gradual underflow* for small-magnitude numbers, but they have no equivalent for large-magnitude numbers.) The idea of tapered accuracy was first proposed by Morris in 1971 and a full implementation was created in 1989, but it used a separate field to indicate the number of bits in the exponent. Not only was that scheme wasteful of bits, it led to a plethora of bit patterns that all meant the same number and a hopelessly complicated mess when comparing $x < y$ and $x = y$. This does not happen with Type III posits: every bit string has a unique meaning, and comparisons are the same as comparing signed integers.

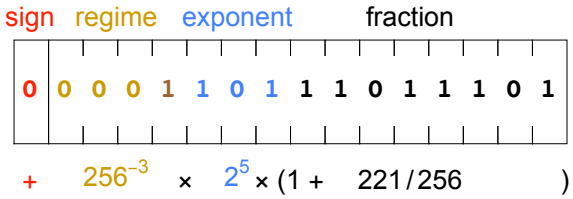
The posit environment is set by specifying two numbers: *nbits*, the total number of bits in the posit, and *es*, the maximum size of the exponent bit field. Again, the bit fields represent a numerical value similarly to how the bit fields in floats work, **except**:

- If the sign bit is 1, the unum bit string is negated (treating it as a standard 2's complement integer) before decoding the remaining fields. This eliminates the need for "negative zero" and all the complications of having two distinct bit patterns represent the same real value. (It is possible for hardware to decode the meaning of a posit bit string directly *without* first taking the absolute value, which makes the circuits faster and simpler. However, that makes the explanation more complicated, so here we will assume the 2's complement is used when working with a negative posit value.)
- A machine instruction that tests if two integers are equal will also serve to compare two posits, whereas float instructions need a additional exception tests for negative zero (equal even though bit patterns are different) and also a check for NaN values (not equal, even when bit patterns are identical).
- The number of identical regime bits *r* determines a positive or negative power of $2^{2^{es}}$ that scales the value, and the exponent bits scale the value by a power of 2 ranging from 2^0 to $2^{2^{es}-1}$. Like, with $es = 3$, the three exponent bits can represent binary integers from `000` $= 0$ to `111` $= 2^{2^3-1} = 7$. The regime bits determine the positive or negative power of $2^3 = 8$, so the regime and exponent together can express a signed integer *k* that works like the integer expressed by the exponent bits in a float.
- There are no subnormal (denormalized) numbers. The implied "hidden bit" before the fraction bits is *always* 1.
- Posits do not underflow to zero or overflow to infinity like floats do. Rounding is to the nearest number, where "nearest" is the smallest difference when rounding fraction bits, and smallest ratio when rounding exponent bits. There are no flags hidden in the state of the processor to indicate that something happened during the calculation; what you see is what you get. Debugging tools can provide a rich source of information regarding underflow, overflow, and other exceptions when a code is being developed, but they do not burden the processor during normal operation.

The other major component of the definition of posit arithmetic is the *quire*, the fixed-size set of bits used for scratch operations that are mathematically exact in the *g*-layer. A quire can be thought of as a dedicated register that permits dot products, sums, and other operations to be performed with rounding error deferred to the very end of the calculation. All computers use a hidden scratch area for temporary calculations. In posit arithmetic, the quire data type is accessible to the programmer, which is what makes possible for posits to follow the rules of algebra much more closely than floats do. The quire concept is incredibly powerful. It allows posits to "punch above their weight class." We will later show some examples where 16-bit posits can outperform 64-bit floats at both speed and accuracy!

1.2 A 16-Bit Example

People always ask to see an example right away instead of wading through a litany of careful definitions. So here is an example of how the bit fields work for a 16-bit posit. Pick *es* = 3, so the regime bits scale by negative and positive powers of $2^{2^3} = 256$. ***Note***: A "standard" 16-bit posit has an *es* size of 1, not 3, but 3 works better here for purposes of illustration.



The sign bit of **0** means the value is positive. The regime bits have a run of three **0**s terminated by the opposite bit **1**, which means the power of *useed* is −3. The scale factor from the regime is 256^{-3}. The exponent bits, **101**, represent 5 as an unsigned binary integer, and contribute another scale factor of 2^5. Lastly, the fraction bits **11011101** represent 221 as an unsigned binary integer, so the fraction is $1 + 221/256$. The expression written underneath the bits works out to $477 \times 2^{-27} \approx 3.55393 \times 10^{-6}$.

The regime bits may seem like a weird and artificial construct, but they actually arise from a natural and elegant geometric mapping of binary integers to the projective real numbers on a circle. The extraction of bit fields and their meaning will be discussed in more detail later.

Notice that the largest shift expressible by the exponent here is **111** = 7 bits, or 2^7. That means the exponent can scale the fraction by a factor of 1, 2, 4, 8, …, or 128. If you need to scale by a factor of 256, that's where the regime bits come in, because they express powers of 256.

1.3 The two exception values for posits

The above description of bit fields and their meanings has exceptions for just two posit bit strings:

- If all bits are **0**, the number represented is *zero*.
- If only the first bit is **1** and the rest are **0**, the value represented is ±∞, sometimes called "projective infinity" or "complex infinity" or "the point at infinity." If you do not have a way to type "±∞", `inf` can be used to mean the same thing. The values ±∞ and zero are reciprocals of each other, and the division of any nonzero number by zero does not trigger an error condition; it is simply ±∞. The diagrams in Section 2 will make this more clear.

There is only one rounding mode: *round to nearest, tie to nearest even*, the same as the default rounding mode for IEEE 754 floats. If a programmer or user feels the need for the other three modes that floats support (round down, round up, round toward zero), that means the application calls for *valids*, not posits, or perhaps just a decent debugging tool.

What "nearest" means in rounding is a bit more complicated than with floats, because the bits clipped off on the right are not necessarily fraction bits; part or all of the exponent field can be clipped off, too. When an exponent bit is clipped off and must be rounded, "nearest" means

"nearest exponent," and the tie point is the *geometric mean* between the two choices instead of the *arithmetic mean*. If that worries you, consider that it happens very rarely, and only when the calculation is about to hit the limits of the dynamic range so it's probably already off the rails and the algorithm needs to be debugged.

If a calculation exceeds the largest representable posit by some amount, should it overflow to infinity? Certainly not! That would turn some finite amount of error into an *infinitely wrong* answer. Which is what floats do. Similarly, we never allow underflow to zero since doing so means throwing away *all* information about an answer… even the sign. As with nonstandard rounding modes, a need to distinguish underflow, overflow, and NaN is generally an indication that an application is still in development and not yet ready for production use. To quote Donald Knuth: "It has unfortunately become customary in many instances to ignore exponent underflow and simply to set underflows to zero with no indication of error. This causes a serious loss of accuracy in most cases (indeed, it is the loss of *all* the significant digits), and the assumptions underlying floating point arithmetic have broken down, so the programmer really must bet old when underflow has occurred. Setting the result to zero is appropriate only in certain cases when the result is later to be added to a significantly larger quantity.

To see why this is, imagine there is some chance that an application produces an out-of-range number that is too large or too small. Posit arithmetic returns a number of magnitude *maxpos* or *minpos*, and it is **up to the programmer** what to do about that. Which means the handling of such situations is visible in the source code, like

```
if (x = minpos || x = -minpos, 0, x);
```

In other words, if a programmer prefers that a calculation underflow, then explicitly replace the small result with zero. It is rare for a programmer to want this. It is even more rare for a programmer to want a computer to underflow with the only indicator being a flag hidden in the processor that can only be made visible using assembly language! For over thirty years, the IEEE 754 Standard has forced hardware engineers to put flags into processor registers in the hope that popular computer languages would make them visible somehow. It hasn't happened.

Exception values in any data type add to hardware complexity, and also increase energy consumption. The posit format has far fewer exceptions than floats. In a 64-bit representation, floats have *quadrillions* of bit patterns that represent "Not a Number" (NaN), whereas posits use those bit patterns to represent numerical values. If the user asks for the square root of a negative number, zero divided by zero, etc. with posits, the computer language should catch it and report it if that is in the rules for the language. If something makes it through the protections of the language to actually attempting something like $0 \div 0$, the default behavior is to produce the posit for $\pm\infty$ and continue computing, what float arithmetic calls "Quiet NaN." This avoids the logical contradiction of declaring a result to be unrepresentable as a number, and then… *representing it as a number*. It also avoids any need for the hardware to test for a NaN when beginning an arithmetic operation, since NaN is not possible as an input. There is no bit string that represents it. The $\pm\infty$ value fulfills the role of a "quiet NaN." While a computer language should protect against the following situations,

$\pm\infty + \pm\infty$
$\pm\infty - \pm\infty$
$0 \times \infty$
$\infty \times 0$
$0 \div 0$
$\pm\infty \div \pm\infty$,

if it does not, we let $\pm\infty$ serve as the quiet NaN indicator. It is the dumping ground for all that goes wrong and we can consider it to be “rounding to infinity.” Remember, even very large numbers never round to infinity with posits, so getting $\pm\infty$ means you did one of the things your math teachers told you never to do, liked dividing by zero. This policy is useful in situations, say, where an array of numbers needs to be processed to completion and the cases that throw an exception can be discarded. Sometimes, input data fails to come in so an value is completely unknown, but we don’t want to halt computation because there was an occasional dropout. That’s when you need a quiet NaN.

Some may object, "But the exception for the square root of a negative number is different, and I need a different behavior for that!" Think about what this means. A program occasionally tries to compute the a real number as the square root of a negative value. Is this something the hardware should try to handle gracefully? Of course not. It's a *bug*. The programmer needs to stop that situation from happening in the first place. The offending line of the code

```
y = sqrt (x)
```

needs to be replaced with something like

```
y = if (x ≥ 0, (*then*)sqrt (x), (*else*) reportsquarerooterror (x))
```

or it may be that the definition of the **sqrt(x)** function is to produce an error message and halt, in a particular language and computing environment. If the handling of errors is entrusted to an interrupt that invokes the operating system, that means there will be *different behavior on different systems*. The next objection is usually “But conditional statements will slow down my code!” The answer to this is similarly to ask if hardware is supposed to handle programming bugs at full speed, or if the programmer should figure out the places where exceptions can happen, and protect just those places with explicit instructions about what to do. Obviously, the latter. Asking hardware to always be on the lookout for errors and to produce detailed info about what happened is like turning on debug mode when compiling a program, and then demanding that debug mode runs at full speed so that production codes can always be in debug mode with no performance penalty. Experienced programmers will recognize the foolishness of such a demand.

Valid mode has far more informative ways of representing the results of operations as *sets* (including the empty set), and can continue computing even when mathematicians would pronounce a result *indeterminate*. Posits are designed for speed, simplicity, and economy, which for many computer users (such as those who play video games, or who use numerical methods for which floats appear to be good enough in practice), for whom speed is paramount. For those

concerned with rigorous computing that puts bounds on results and tracks any loss of accuracy, or those still working out the numerical behavior of an algorithm to make sure it behaves, valid mode is the answer.

> The current IEEE 754 definition attempts a *mixture* of the two esthetics, speed and validity, which means **it achieves neither**. “Almost true” is mathematically the same as “false.”

For example, the variety of rounding modes is touted as a technique for checking the sensitivity of an algorithm to rounding error, but that technique still provides no *guarantees* of validity and it adds complexity to the circuitry. Similarly, "negative zero" is sometimes treated as some kind of indicator of a negative infinitesimal, and at other times it is simply zero, creating more complexity and gate delays for hardware designers, but infinitesimally little value to their customers.

2 Specifics: Converting a posit bit string to a value

2.1 Setting the posit environment

We defined the computing environment with *esizesize* and *fsizesize* in Type I unums. Instead, here, we define *nbits* and *es* as the global integer values we need for establishing the meaning of a posit. The value *nbits* is the total number of bits in the number and can be any integer 2 or greater; *es*, the number of exponent bits, can be any integer 0 or greater. The *nbits* and *es* values can be chosen independent of each other; it may seem strange, but you can have, say, a 4-bit posit with a maximum of 5 exponent bits. It does not break the math.

Similarly to Type I unums, the **setpositenv** routine uses *nbits* and *es* to compute values that characterize the posit environment. The number of possible bit patterns, *npat*, is 2^{nbits}. The value $2^{2^{es}}$ is called *useed* because it arises so often. Besides computing *npat* and *useed*, **setpositenv** finds the minimum and maximum positive numbers, *minpos* and *maxpos*. Those values determine the dynamic range, consistent with IEEE floats. The extremes are always exact reciprocals for posits; $\frac{1}{minpos} = maxpos$; the dynamic range is perfectly balanced about 1.0, and every power of 2 has an exact reciprocal. (These properties do not hold for IEEE floats.)

It is also necessary to set the size of the quire register, **qsize**, and the number of extra bits it has for worst-case summations that produce carry bits, **qextra**. The quire register will be explained later in a section on Fused Operations.

```
setpositenv[{n_Integer /; n ≥ 2, e_Integer /; e ≥ 0}] :=
 ({nbits, es} = {n, e};
  npat = 2^nbits;
  useed = 2^2^es;
  {minpos, maxpos} = {useed^-nbits+2, useed^nbits-2};
  qsize = 2^⌈Log[2, (nbits-2) 2^es+2+5]⌉;
  qextra = qsize - (nbits - 2) 2^es+2;)
```

For example, a 6-bit posit environment with a single exponent bit is established this way:

```
setpositenv[{6, 1}]
```

That command assigned these global variables:

```
{nbits, es, npat, useed, minpos, maxpos, qsize, qextra}
```

$$\left\{6, 1, 64, 4, \frac{1}{256}, 256, 64, 32\right\}$$

Most computer architectures have a strong preference for powers of two as the bit sizes for variables, or at least multiples of 8-bit bytes. If it is not a performance burden for a computer to use, say, 56-bit posits instead of 64-bit floats, then the 56-bit posits can often provide more accurate answers than 64-bit floats. The savings in memory and bandwidth needed for the smaller numbers may save time (and energy and power) on bandwidth-limited computers, depending on how much they require alignment of fetches and stores on larger power-of-two boundaries. Users may discover that 64-bit floats are overkill for their application, but that they need more accuracy or dynamic range than a 32-bit float provides; in some of those cases, a *32-bit posit* may suffice and produce a clear performance and storage advantage over 64-bit standard floats. Similarly, 16-bit posits can sometimes replace 32-bit floats, and we know of at least one application (neural network training) where 8-bit posits can replace 16-bit floats. The use of the quire register is a powerful way to create long sequences of arithmetic operations with only one rounding error at the end, and can even allow 16-bit posits to replace 64-bit floats in some situations.

Notice that the ratio of *maxpos* to *minpos* is $useed^{2\,nbits-4}$, which determines the dynamic range of the posits. Recall that *useed* is $2^{2^{es}}$. The posit format uses regime bits to raise *useed* to the power of any integer from $-nbits+1$ to $nbits-1$, so the dynamic range is an exponential of an exponential of an exponential. Hence, posits can create a larger dynamic range from fewer exponent bits than the system used for IEEE floats, leaving more fraction bits available to improve the accuracy of results. In other words, the *es* setting is powerful, so be careful with that thing. A value larger than 5 might tax your computer's memory; simply printing the exact value of *minpos* with $es = 5$ requires several gigabytes.

2.2 Extracting the sign bit

A posit bit string, if interpreted as a 2's complement integer, would range from $-npat$ to $npat-1$. *Mathematica* does not handle 2's complement integers as a native type, so this explanation will have to make do with integers between 0 and $npat-1$. The unsigned integer for $npat/2$ is the posit bit string that represents $\pm\infty$. If you have signed integers the posit bit string for $\pm\infty$ looks like $-npat/2$.

The **positQ** test returns **True** if an input integer (considered as a bit string) is a viable posit, and **False** otherwise.

```
positQ[p_Integer] := 0 ≤ p < npat
```

The following function will help deal with the lack of 2's complement integers in *Mathematica*.

```
twoscomp[sign_, p_] := Mod[If[sign > 0, p, npat - p], npat]
```

Extracting the sign bit of a posit is easy, It's the most significant bit, **0** for positive numbers (and zero) or **1** for negative numbers (and ±∞). The color-coding of bits, similar to that used for all types of unums, makes long binary strings easier to read. The sign bit is color-coded red (RGB components 1, 0.125, 0):

```
signbit[p_ /; positQ[p]] := IntegerDigits[p, 2, nbits][[1]]
```

Test it at the extreme points of possible posit values, including an illegal one for the {6, 1} environment we just set:

```
signbit[1]
signbit[npat - 1]
signbit[npat]
```

```
0

1

signbit[64]
```

Notice that the **positQ** function protects us from evaluating the **signbit** of the out-of-range integer **npat**, since the posit bit strings range from 0 to *npat* – 1 (when viewed as unsigned integers). In a C environment, the test that an input *p* is a signed integer representing a valid posit would be to require $-32 \le p < 32$, not $0 \le p < 64$. (When *Mathematica* is asked to perform an out-of-domain operation, it simply echoes the expression back to the user, like when it returns "`signbit[64]`" above.)

2.3 Extracting the regime bits

The *regime bits* are dictated by the length of the run of identical bits, either all **0**s or all **1**s. Identical regime bits are color-coded amber (RGB 0.8, 0.6, 0.2). A simple way to decode it is, if the posit has a sign bit of **1**, negate the bit string for the posit as a 2's complement signed integer (which means flip all the bits and add 1). In assembly language, many of the complicated-looking expressions in this function are single opcodes that execute in one clock cycle on most processors, like "Find First One" or "Count Leading Zeros." In the **regimebits** function we take the binary digits apart as a string, but real hardware (or a routine in a low-level language) could perform bit extraction much more quickly and simply.

The run of bits terminates when we reach the end of the string, or when the opposite bit happens.

The opposite bit that terminates a regime is color-coded brown (RGB 0.6, 0.4, 0.2).

```
regimebits[p_ /; positQ[p]] :=
 Module[{q = twoscomp[1 - signbit[p], p], bits, bit2, npower, tempbits},
  bits = IntegerDigits[q, 2, nbits];
  bit2 = bits[[2]]; (* Look for the run length after the sign bit. *)
  tempbits = Join[Drop[bits, 1], {1 - bit2}]; (* Drop the sign bit,
  but append a complement bit as a sure-fire way to end the run. *)
  npower = (Position[tempbits, 1 - bit2, 1, 1])[[1]] - 1;
  (* Find first opposite bit. *)
  Take[bits, {2, Min[npower + 1, nbits]}]]
```

The interpretation of the regime bits is a lot like clothing sizes; L is Large, XL is Extra Large, XXL is Extra Extra Large, and so on. Count the number of X characters. The function **regimevalue** uses a similar principle, and it's a one-liner function. It could have been the direct output of **regimebits**, but for clarity, we calculate the integer represented by the regime bits in two stages.

```
regimevalue[bits_] := If[bits[[1]] == 1, Length[bits] - 1, -Length[bits]]
```

Here are two examples. **Notice that for positive exponents, the regime value is one less than the number of bits in the run, since we need to be able to represent a value of zero**. Again, hardware would have no trouble doing this kind of thing very quickly; it looks a lot like 2's complement logic.

First, a small positive number (remember, we are still in a 6-bit posit environment):

```
regimebits[2^^000011]
regimevalue[%]
```

```
{0, 0, 0}
```

```
-3
```

Next, try a negative number, so that it gets negated 2's complement style, and the run of **0** bits turns into a run of **1** bits. Note: the 2's complement of **100001** is **011110** + **1** = **011111**; that is, flip all the bits and add **1**.

```
regimebits[2^^100001]
regimevalue[%]
```

```
{1, 1, 1, 1, 1}
```

```
4
```

2.4 Extracting the exponent bits

The next field is the exponent bits. Even if *es* is greater than zero, there might not *be* any exponent bits, because the regime bits can crowd some or all of them off the right end of the number! For example, **011111** has no exponent bits, and neither does **011110**. Exponent bits start to appear for shorter runs of regime bits, like **011100** and **011101**.

In the following code, the run of regime bits is terminated by the opposite bit, so we skip over that termination bit and look at the next *es* bits or however many are left, whichever is smaller. The result can be anything from the empty set, {}, to a sequence of bits of length *es*.

```
exponentbits[p_ /; positQ[p]] :=
 Module[{q = twoscomp[1 - signbit[p], p], bits, startbit},
  startbit = Length[regimebits[q]] + 3;
  bits = IntegerDigits[q, 2, nbits];
  If[startbit > nbits, {},
   Take[bits, {startbit, Min[startbit + es - 1, nbits]}]]]
```

There is no "bias" in these exponent bit strings as there is with floats, so the power of 2 they express is simply an unsigned integer. Exponent bits are color-coded blue (RGB value 0.25, 0.5, 1). If the exponent bits are `110` = 6, that means 2^6 is the scale factor contributed by the exponent bits. If there are no exponent bits, then the exponent is zero and the scale factor contributed by the exponent bits is $2^0 = 1$.

2.5 Extracting the fraction bits

Once we have the fraction bits, we have all four parts of the posit bit string and can directly compute the value it represents. The fraction bits are all the bits to the right of the exponent, if they haven't been crowded off the right end by the other bits. (If they have, the value of the fraction is 1. In other words, the hidden bit is always `1`, even if there is no place to hide it!) The **`fractionbits`** function differs from **`exponentbits`** only in using a value for **`startbit`** that is *es* bits farther to the right, and in taking *all* remaining bits instead of stopping at a set maximum number. This is how posits have variable accuracy depending on the scale factor. If the scale factor is small, there will be plenty of bits to express a highly-accurate fraction. The bigger or smaller the magnitude of a posit, the fewer bits remain for accuracy.

```
fractionbits[p_ /; positQ[p]] :=
 Module[{q = twoscomp[1 - signbit[p], p], bits, startbit},
  startbit = Length[regimebits[q]] + 3 + es;
  bits = IntegerDigits[q, 2, nbits];
  If[startbit > nbits, {}, Take[bits, {startbit, nbits}]]]
```

The color-coding of the fraction bits is simply to leave them **black**.

2.6 Assembling the pieces: the `p2x` function

Following the naming conventions of the Type I unum prototype environment, we call the function that changes a posit bit string *p* into a mathematical value *x* the **`p2x`** function. It puts the pieces together as follows:

$$\text{Posit value represented by signed integer } p = \begin{cases} p = 0, & 0, \\ p = -npat/2, & \pm\infty, \\ \text{all other } p, & (-1)^s \times useed^k \times 2^e \times f \end{cases}$$

where *s* = the sign bit, *k* is the integer represented by the regime bits, *e* is the integer represented

by the exponent bits, and *f* is the fraction (including the hidden bit, which is always **1**.) The following routine extracts *s*, *k*, *e*, and *f*, and then applies the above formula.

```
p2x[p_ /; positQ[p]] :=
 Module[{s = (-1)^signbit[p], k = regimevalue[regimebits[p]],
   e = exponentbits[p], f = fractionbits[p]},
  e = Join[e, Table[0, es - Length[e]]];
  (* Pad with 0s on the right if they are clipped off. *)
  e = FromDigits[e, 2];
  If[f == {}, f = 1, f = 1 + FromDigits[f, 2] × 2^-Length[f]];
  Which[
   p == 0, 0,
   p == npat / 2, ComplexInfinity, (* The two exception values, 0 and ±∞ *)
   True, s × useed^k × 2^e × f]]
```

(Since *Mathematica* doesn't support 2's complement integers, the test for ±∞ is if *p* is *npat*/2 instead of –*npat*/2. And we use **ComplexInfinity** as *Mathematica*'s equivalent for ±∞.)

The "**colorcodep**" function makes the posit binary strings easier to read by first showing the original bits, and then coloring the bit fields for either that number or its 2's complement if it is on the left half of the projective real circle.

```
colorcodep[p_ /; positQ[p]] := Module[{s = signbit[p],
    r = regimebits[p], e = exponentbits[p], f = fractionbits[p]},
  Row[{IntegerString[p, 2, nbits], "→",
    Style[If[s == 0, "+", "-"], ■],
    Style[Row[r], ■],
    If[Length[r] ≤ nbits - 2, Style[1 - r[[1]], ■], ""],
    If[Length[e] > 0, Style[Row[e], ■], ""],
    If[Length[f] > 0, Row[f], ""]}]]
```

We need at least two exponent bits to demonstrate how the exponent bits can be crowded out by the regime bits, so we will change the environment to {6, 2}, and create a table of every possible bit string and what it means. The table starts with the bit string representing the smallest signed integer (**100000** is –32), counts up to zero, and keeps going to the largest signed integer (**011111** is +32):

```
setpositenv[{6, 2}];
Table[Row[{colorcodep[u + j], "   ", p2x[u + j], "          "}],
  {u, 0, 31}, {j, 32, 0, -32}] // TableForm
```

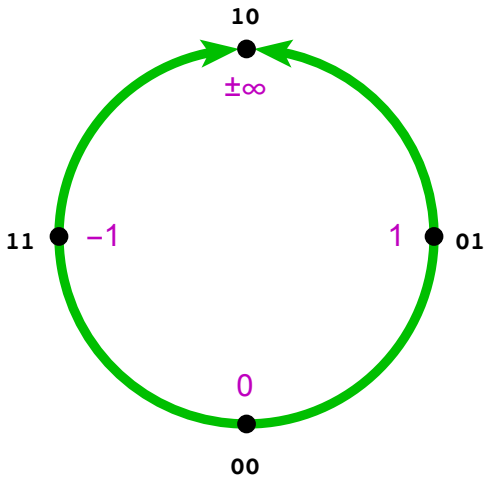
100000→-00000	ComplexInfinity	000000→+00000	0
100001→-11111	-65 536	000001→+00001	$\frac{1}{65\,536}$
100010→-11110	-4096	000010→+00010	$\frac{1}{4096}$
100011→-11101	-1024	000011→+00011	$\frac{1}{1024}$
100100→-11100	-256	000100→+00100	$\frac{1}{256}$
100101→-11011	-128	000101→+00101	$\frac{1}{128}$
100110→-11010	-64	000110→+00110	$\frac{1}{64}$
100111→-11001	-32	000111→+00111	$\frac{1}{32}$
101000→-11000	-16	001000→+01000	$\frac{1}{16}$
101001→-10111	-12	001001→+01001	$\frac{3}{32}$
101010→-10110	-8	001010→+01010	$\frac{1}{8}$
101011→-10101	-6	001011→+01011	$\frac{3}{16}$
101100→-10100	-4	001100→+01100	$\frac{1}{4}$
101101→-10011	-3	001101→+01101	$\frac{3}{8}$
101110→-10010	-2	001110→+01110	$\frac{1}{2}$
101111→-10001	$-\frac{3}{2}$	001111→+01111	$\frac{3}{4}$
110000→-10000	-1	010000→+10000	1
110001→-01111	$-\frac{3}{4}$	010001→+10001	$\frac{3}{2}$
110010→-01110	$-\frac{1}{2}$	010010→+10010	2
110011→-01101	$-\frac{3}{8}$	010011→+10011	3
110100→-01100	$-\frac{1}{4}$	010100→+10100	4
110101→-01011	$-\frac{3}{16}$	010101→+10101	6
110110→-01010	$-\frac{1}{8}$	010110→+10110	8
110111→-01001	$-\frac{3}{32}$	010111→+10111	12
111000→-01000	$-\frac{1}{16}$	011000→+11000	16
111001→-00111	$-\frac{1}{32}$	011001→+11001	32
111010→-00110	$-\frac{1}{64}$	011010→+11010	64
111011→-00101	$-\frac{1}{128}$	011011→+11011	128
111100→-00100	$-\frac{1}{256}$	011100→+11100	256
111101→-00011	$-\frac{1}{1024}$	011101→+11101	1024
111110→-00010	$-\frac{1}{4096}$	011110→+11110	4096
111111→-00001	$-\frac{1}{65\,536}$	011111→+11111	65 536

3 Visualizing the Projective Reals

3.1 The smallest posit size with a *useed*

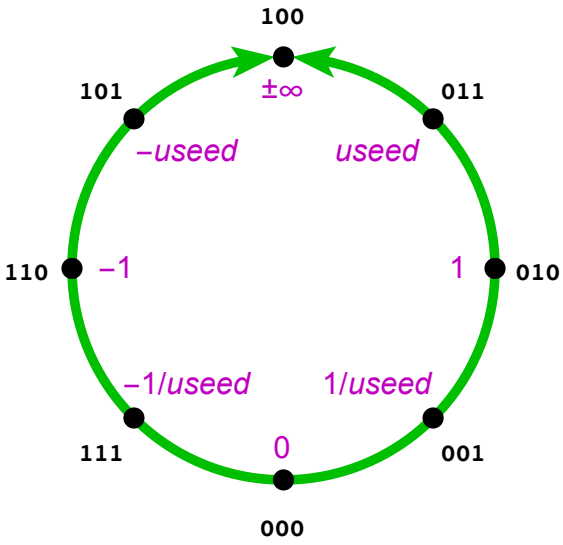
Unlike the real number line, the *projective reals* wrap the line onto a circle so that negative and positive infinity meet at the top.

Sometimes the way things were invented is not the easiest way to understand them, which is why we first showed the bit fields in Section 2, and only now show their geometrical derivation. Type III posit arithmetic is derived from Type II unums that also map binary integers to the projective reals, but in Type III we relax the requirement that *all* values have an exact reciprocal. Both Type II and Type III unums start with this two-bit template:



The 2's complement signed integers around the outside of the ring wrap from positive to negative at exactly the same point the real numbers do. This eliminates "negative zero"; it unfortunately takes away –∞ and +∞ as distinct quantities, but the workarounds for that are much simpler than having to deal with two forms of zero that are sometimes considered equal and sometimes treated differently, which is what IEEE floats do.

The above ring is only two bits, yet it represents a number system. We move to three bits by inserting a value between 1 and ±∞. It could be real number greater than 1; it could be 2, or 10, or π, or 1.00003, or 10^{100}. The choice "seeds" the way the rest of the ring of unums gets populated, so we call it the *useed*. We follow Type II rules and make sure negation reflects about the vertical axis and reciprocation reflects about the horizontal axis.

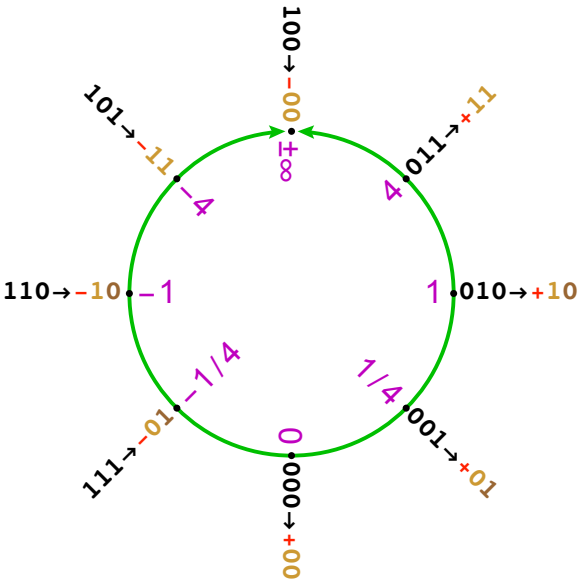


For the next step, what should we put between *useed* and ±∞? Well, $useed^2$ certainly works. We then could get an elegant new symmetry by putting $useed^{1/2} = \sqrt{useed}$ between 1 and *useed*, especially if $\sqrt{useed}$ is an integer that fits computer hardware nicely. If $\sqrt{\sqrt{useed}}$ is an integer, we can repeat the process.

The *useed* values that are computer hardware-friendly are 2, 2^2, $(2^2)^2$, etc., repeatedly squaring to make sure we can repeatedly take the square root and get back to 2. The number series is 2, 4, 16, 256, 65536,…. Between 1 and 2 we can populate the ring *linearly*, that is, {1, 1.5, 2} if we have one more bit, or {1, 1.25, 1.5, 1.75, 2} if you have two more bits, and so on, exactly the way a floating-point fraction works. The number of times you square to get *useed*, starting with the value 2, is *es*. This is why $useed = 2^{2^{es}}$.

The **`ringplot`** routine shows posits representing projective real numbers arranged on a ring, with their mathematical form on the inside and the color-coded binary form on the outside. It uses whatever *nbits* and *es* environment settings are in place. We switch to rotated text so we can cram more values into the space, even though it makes it slightly peculiar to read. Try it for *nbits* = 3 and *es* = 1:

```
setpositenv[{3, 1}]; ringplot
```



If the ring were the points of a compass, +1 is always east, –1 is west, 0 is south, and ±∞ is north. The value of *useed* is always at the northeast point of the compass. Reflecting about the vertical axis always gives the negative of every number. Reflecting about the horizontal axis gives the exact reciprocal for 0, ±∞, and all powers of two; it gives the *approximate* reciprocal for values *between* powers of two; the value represented by the horizontal reflection of a posit value x ranges from $\frac{1}{x}$ to $\frac{1.125}{x}$ in all cases.

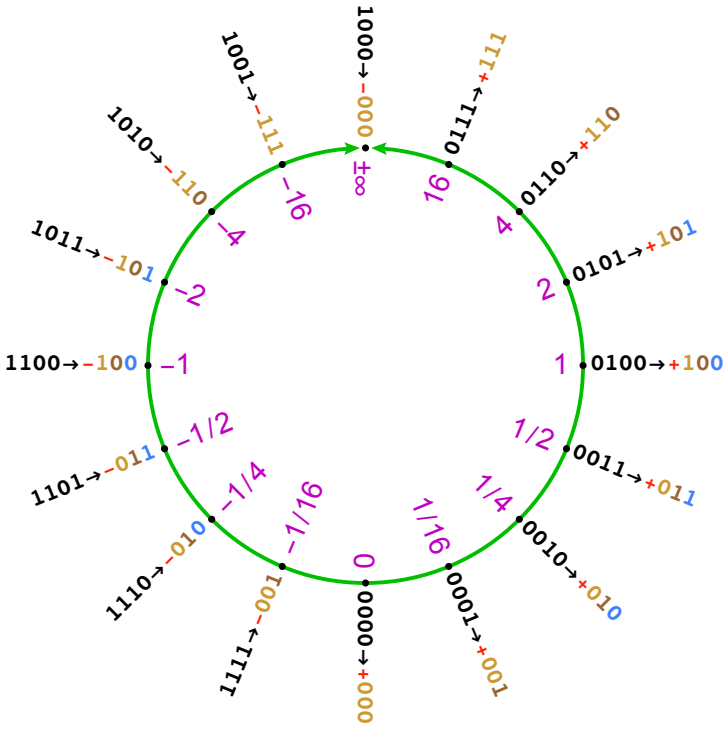
These unums have a recursive definition. In appending a bit by increasing *nbits*, appending **0** does not change the value; appending **1** creates a new point on the circle midway between existing values, with the following rules for what value it represents:

- If it is next to the bottom of the circle between 0 and $\pm 2^j$, the new value is **smaller** than $\pm 2^j$ by a ratio of *useed*. (The added bit is a **regime** bit.)
- If it is next to the top of the circle between $\pm 2^j$ and ±∞, the new value is **larger** than $\pm 2^j$ by a ratio of *useed*. (The added bit is a **regime** bit.)
- If it is between two values of magnitude 2^i and 2^j where integers i and j differ by more than 1, the new value has magnitude $2^{(i+j)/2}$. (The added bit is an **exponent** bit.)
- If it is between any other adjacent points x and y, it represents $(x+y)/2$. (The added bit is a **fraction** bit.)

3.2 Introducing exponent bits

The system for adding new points on the ring is simpler than it sounds. Check the color coding for a 4-bit posit; the exponent bits start to show up in the east and the west parts of the ring:

```
setpositenv[{4, 1}]; ringplot
```



Notice that every value in the previous ring plot stays in place, with a **0** appended to its representation. The new values are between the previous ring plot values, and end in a **1** bit. This is one of the beautiful properties of these numbers; they stay put when increasing precision by appending a **0** bit, and the in-between points created by appending a **1** bit automatically increase both dynamic range (top and bottom) *and* accuracy (left and right).

Another thing you can notice is that the first two bits always determine the *quadrant* of the circle where the number resides. As with Type II unums, think of them as the *sign-reciprocal* bits. The first two bits are

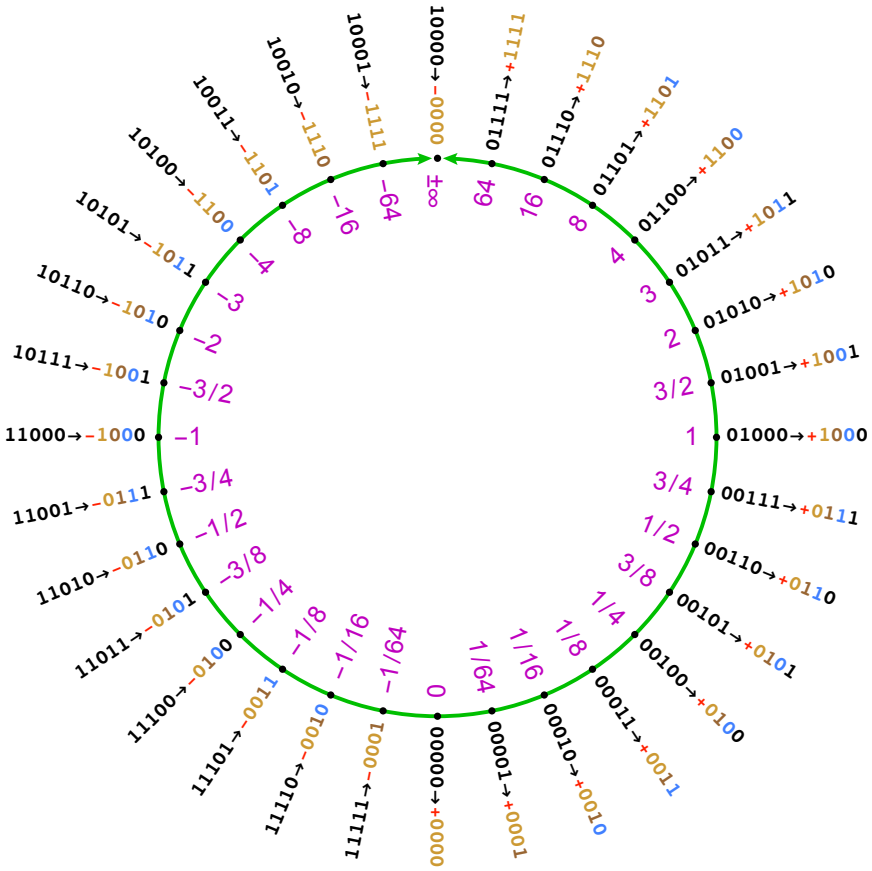
00 in the southeast, $0 \le x < 1$
01 in the northeast, $1 \le x < \infty$,
10 in the northwest, $-\infty < x < -1$,
11 in the southwest, $-1 \le x < 0$.

That also might help explain how hardware can decode the regime bits without taking the 2's complement of a number first.

3.3 Fraction bits appear

The next step takes us to five bits, and then the complete pattern emerges. At the top and bottom of the circle, adding a bit always extends the dynamic range by a factor of *useed* larger and smaller. At the right and the left sides of the circle, adding a bit always now adds one bit of precision to the fraction. With five bits, we now see fraction bits appear in the eastern and western parts of the projective real circle.

```
setpositenv[{5, 1}]; ringplot
```



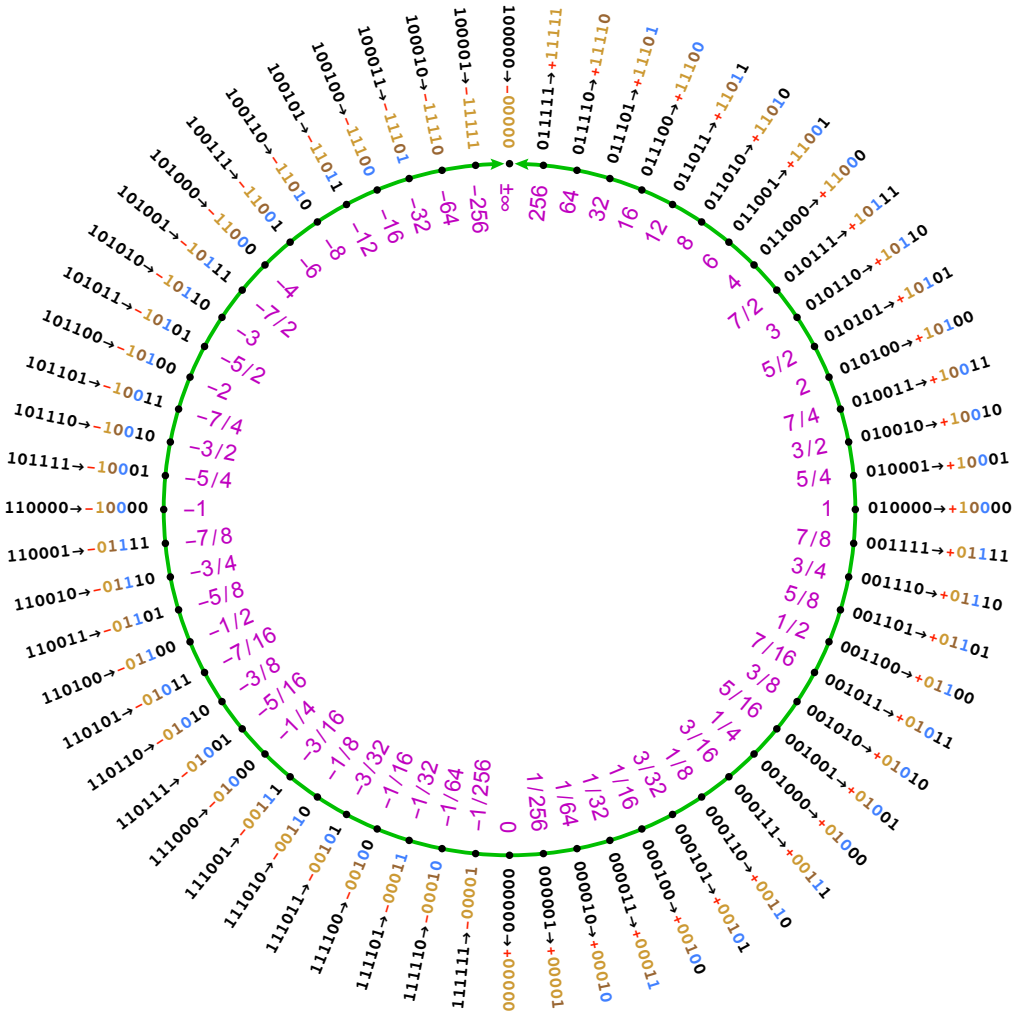
If you increase the number of bits in an IEEE float, you have to decide how many bits to increase in the exponent and how many in the fraction. Changing the number of IEEE exponent bits also means changing the *bias* in the exponent, so in general it is a complicated thing to convert from one IEEE precision to another. That is also true for Type I unums. With Type III posits, it is trivial: you just add bits to the right. Adding bits to the end of a posit increases dynamic range at the north and south parts of the ring, and accuracy at the east and west parts of the ring. This suggests that a calculation can very easily change the environment settings *as it progresses*.

In the above plot, notice that numbers ending with fraction bits do not have perfect reciprocals in their reflected quantity. Close, but not quite. The reflection of 3/2 is 3/4, not 2/3. The reflection of 3 is 3/8, not 1/3. The reflected posit value is still an excellent starting point for an iterative method for computing the reciprocal. Relaxing the requirement that reciprocals be perfect for every number is what makes posits simple and hardware-friendly, by restricting all finite numbers to the form $2^m \times n$ where m and n are integers.

3.4 A six-bit ringplot

If we attempt 6 bits, the ringplot is still readable.

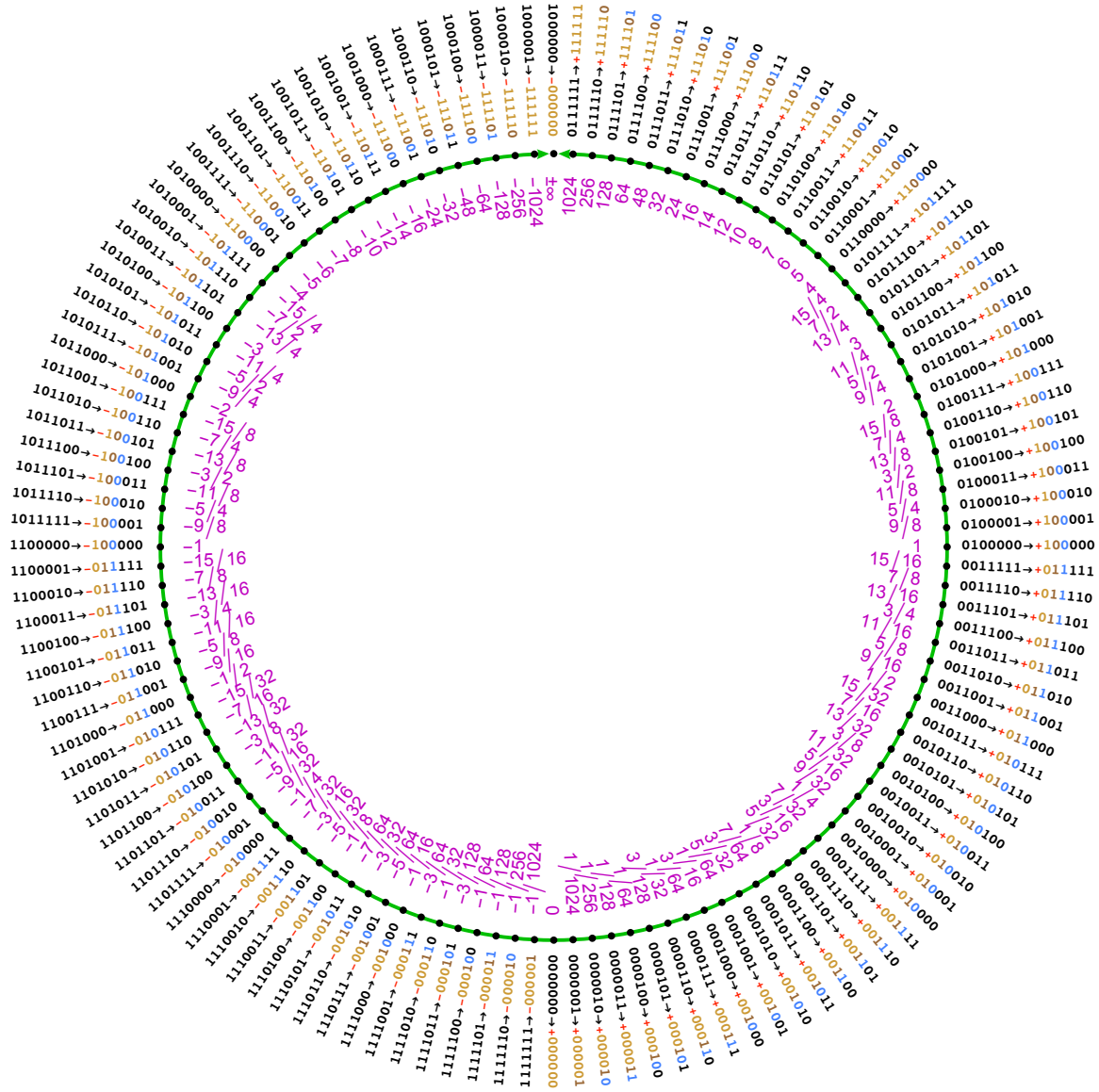
```
setpositenv[{6, 1}]; ringplot
```



3.5 A seven-bit ringplot shows a million-to-one dynamic range

Pushing the graphics to the point where the values almost overlap but are still legible, a 7-bit posit with a 1-bit exponent field can have a dynamic range of six orders of magnitude, yet also have more than one decimal of relative accuracy for points near ±1.

```
setpositenv[{7, 1}]; ringplot
```

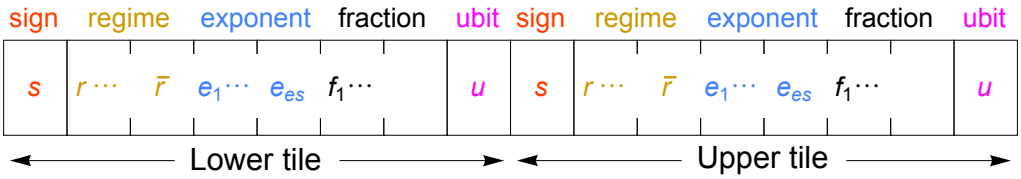


Perhaps this is a good place to point out how rounding is subtly different from what it is for floats. If the rounded bit is a fraction bit, it's *exactly* the same as for floats. Round to the nearest value, and if there's a tie, pick the number ending in a **0** bit. But suppose a calculation landed between 64 and 256; what is the midpoint where we decide if we round up or down? Notice that the rounded bit there is an *exponent* bit, so you are rounding to the *nearest exponent*. For the hardware, it's exactly the same algorithm as when rounding fraction bits. The midpoint between 64 and 256, if we had one more bit of precision, is 128. Numbers between 64 and 128 round to 64; numbers between 128 and 256 round to 256; the exact value 128 is a tie, and 64 is the posit ending in a **0** bit, so 128 rounds to 64.

The rounding rules are therefore consistent, except for results near the exception values 0 and ±∞. Never round toward those extremes. Use *–minpos*, *minpos*, *–maxpos*, or *maxpos* instead. At the very least, that preserves the correct *sign* of the answer.

3.6 Sneak Preview: Valid Arithmetic

One reason for carrying ringplots all the way to seven bits is that the one above forms an excellent basis for 16-bit valids. As a preview to the world of powerful, guess-free arithmetic, valids are pairs of posits that each have an uncertainty bit, or *ubit* appended to the fraction, forming a *tile*. Ubits are color-coded in magenta (RGB 1, 0, 1). The ubit is `0` for all the values shown above, and `1` for all the *open intervals between the values* shown above. A possible 16-bit valid format is a pair of 7-bit posits with ubits at the end of each one, like this:



(Again we use a general *es* value for the purpose of illustration. The *es* value for such a small number of bits would almost certainly be 0. In the ringplot above, *es* is 1.)

Each tile can be any value shown in the above ring of 7-bit posits, followed by a `0` ubit or a `1` ubit. Conventional interval arithmetic is in the form of closed intervals [*a*, *b*] where *a* and *b* are floats, and $a \le b$. In the case of valids *The endpoints need not be ordered*. The lower tile says where to start on the ring; the interval represented includes all tiles counterclockwise from the lower tile until you reach the upper tile, which is also included in the set; it wraps around the circle, crossing ±∞ if necessary! This allows representation of closed, open, and half-open intervals, and because of the use of the projective reals, contiguous intervals **remain contiguous** under addition, subtraction, multiplication, **and division**. For example, the half-open interval (–1/4, 1/16] in the {8, 1} environment of the above ringplot is represented by the pair of tiles

`11100011`→`–00111011`, representing the tile (–1/4, –7/32), and
`00100000`→`–00100000`, representing the tile 1/16.

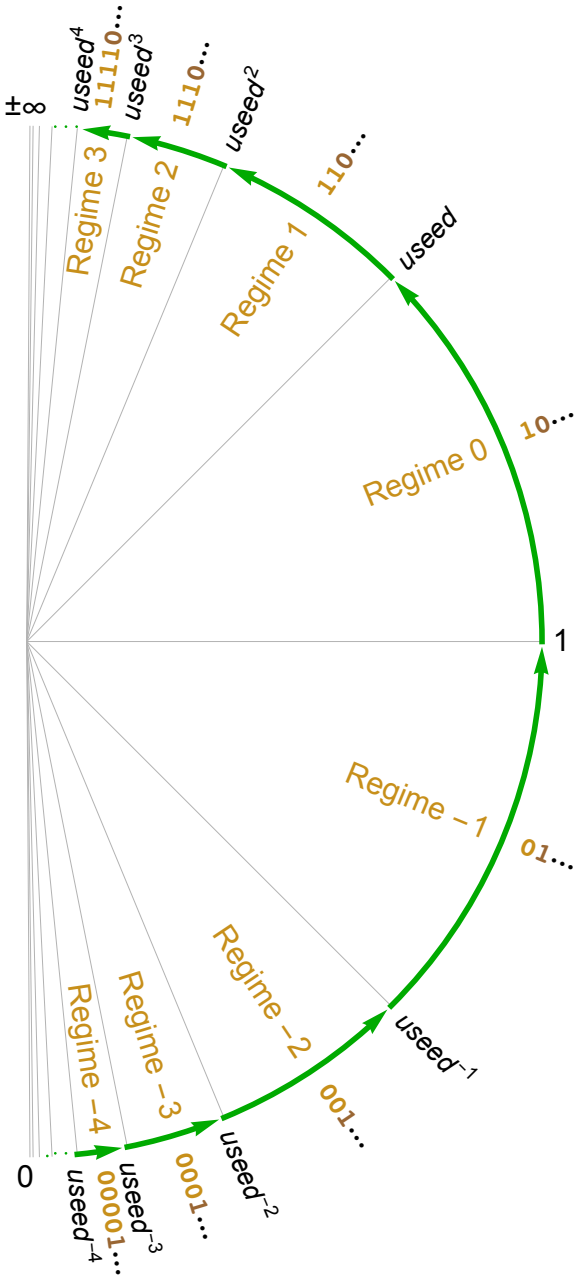
That's a compact way to say the set of all tiles between the two points:

$$(-1/4, -7/32),\ -7/32,\ (-7/32, -3/16),\ -3/16,\ \ldots,\ 3/64,\ (3/64, 1/16),\ 1/16.$$

To reiterate: *posits* are for where floats are good enough, and the algorithms have been shown reliable enough to satisfy user requirements. In contrast, *valids* are for where you need a *provable bound on the answer*. Or when you are still developing an algorithm and debugging its numerical behavior. Or where you want to describe *sets* of real numbers and not just point values. The algorithms for valids are often quite different from the ones for floats, and vice versa. Valids can express a rich source of exception conditions that are useful for debugging, such as the empty set, the entire set of real numbers, or trigger a halt when the interval has gotten too wide (as set by the user) to be useful. Details about the use of valids are discussed elsewhere, but for now the interested reader is referred to *The End of Error: Unum Arithmetic* and its discussion of "ubounds."

3.7 Visualizing the regime bits on the projective real ring

Now that we have shown how reals and integers map geometrically onto a ring, we can visualize the regime bits in particular, by looking at the right half of the ring and showing how the powers of *useed* crowd together at the top and bottom of the ring:



It takes some getting used to, but the regime-exponent pair describes a scale factor of a power of 2 the same way the exponent (minus the *bias*) does for IEEE Standard floats.

If *es* is 1 or more, we could put $useed^{0.5}$ between 1 and *useed*, and $useed^{1.5}$ between *useed* and $useed^2$, and so on. If *es* is 2 or more, we get equal-spaced values $useed^{0.25}$, $useed^{0.5}$ and

useed$^{0.75}$ between 1 and *useed*. In other words, the exponent bits are the fraction bits of the power of *useed* represented by the regime bits.

4 Converting values into posits

4.1 Building the `x2p` conversion function

The method for converting any real value into posit representation is very similar to the way you could convert a number of any type into a float. After checking for exceptions cases (which for posits is simply 0 and ±∞), the number is divided by two or multiplied by two until it is in the interval [1, 2), which then determines the fraction. In the case of posits, the *useed* is a kind of "batched" form of powers of two, so we first divide by *useed* or multiply by *useed* until it is in the interval [1, *useed*). Then the value is repeatedly divided by 2 until it is in the interval [1, 2), and that determines the exponent. The exponent will always be nonnegative, eliminating the need for a bias. The fraction always has a leading `1` bit to the left of the binary point, eliminating the need to handle subnormal exception values that have a `0` bit to the left of the binary point.

The approach could be used to convert a fixed point number, an integer, a float, or any other type that can be repeatedly multiplied or divided by two until it is in the range [1, 2). In *Mathematica*, values are generic "numeric" values, and we need only include values of infinite magnitude in the set of allowed numeric types to assure that it is possible to turn it into a posit. The **`positableQ`** function returns `True` if its argument can be turned into a posit, and `False` otherwise.

```
positableQ[x_] := (Abs[x] == ∞ ⋁ x ∈ Reals)
```

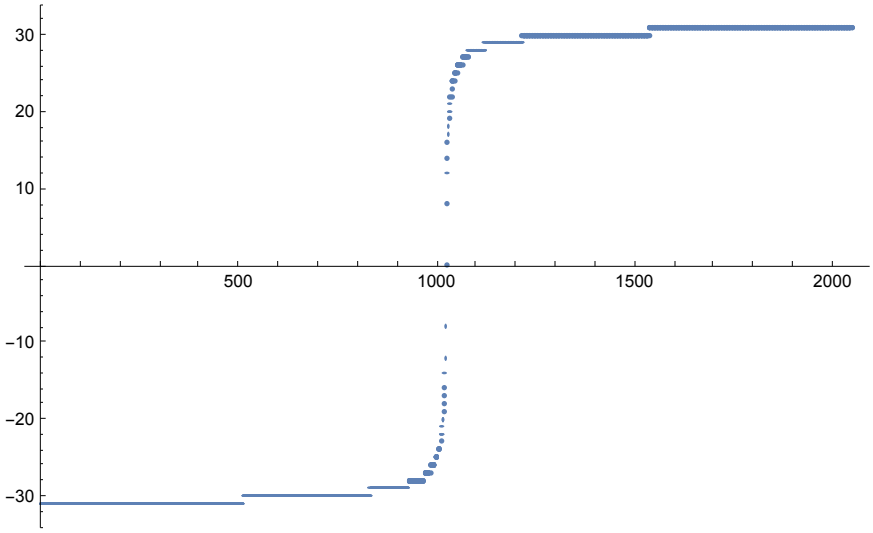
Finally, here is the function that runs a real number into its posit form.

```
x2p[x_ /; positableQ[x]] := Module[{i, p, e = 2^(es-1), y = Abs[x]},
  Which[(* First, take care of the two exception values: *)
   y == 0, 0, (* all 0 bits s *)
   y == ∞, BitShiftLeft[1, nbits - 1], (* 1 followed by all 0 bits *)
   True, If[y ≥ 1, (* Northeast quadrant: *)
    p = 1;
    i = 2; (* Shift in 1s from the right and scale down. *)
    While[y ≥ useed ⋀ i < nbits, {p, y, i} = {2 p + 1, y / useed, i + 1}];
    p = 2 p; i++, (* Else, southeast quadrant: *)
    p = 0;
    i = 1; (* Shift in 0s from the right and scale up. *)
    While[y < 1 ⋀ i ≤ nbits, {y, i} = {y * useed, i + 1}];
    If[i ≥ nbits, p = 2;
     i = nbits + 1, p = 1;
     i++]];
   (* Extract exponent bits: *)
   While[e > 1 / 2 ⋀ i ≤ nbits, p = 2 p;
    If[y ≥ 2^e, y /= 2^e;
     p++];
    e /= 2; i++];
   y--; (* Fraction bits; subtract the hidden bit *)
   While[y > 0 ⋀ i ≤ nbits, y = 2 y; p = 2 p + ⌊y⌋; y -= ⌊y⌋; i++];
   p *= 2^(nbits+1-i);
   i++;
   (* Round to nearest; tie goes to even *)
   i = BitAnd[p, 1]; p = ⌊p / 2⌋;
   p = Which[
     i == 0, p, (* closer to lower value *)
     y == 1 ⋁ y == 0, p + BitAnd[p, 1], (* tie goes to nearest even *)
     True, p + 1 (* closer to upper value *)];
   Mod[If[x < 0, npat - p, p], npat (* Simulate 2's complement *)]
  ]
 ]
```

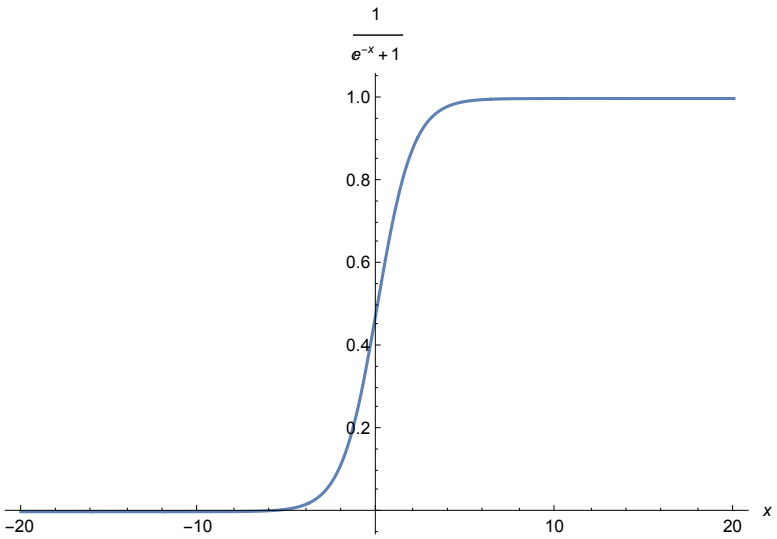
The **x2p** function is mostly written with a vocabulary designed to translate easily into low-level operations in C, or to chip-level circuitry. When you see a line like "**p = 2 p+1**", that would be best expressed as `p=(p<<1)||1` in C; in *Mathematica*, **p=BitOr[BitShiftLeft[p,1],1]** seems too verbose, so we write it with arithmetic that accomplishes the intended bit logic.

4.2 The sigmoid function

Here are some visualizations of the **x2p** function that also help test its correctness. First, a plot of the two's complement integer produced as the argument ranges from –*maxpos* to *maxpos*:

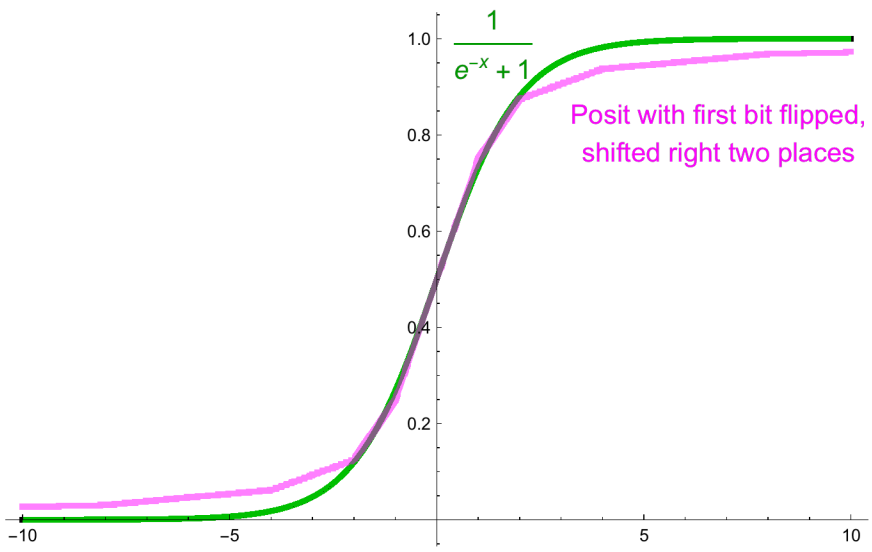


As Isaac Yonemoto has observed, this plot is similar to the kind of “sigmoid” function that Machine Learning requires for deep learning neural networks. It simply needs to be scaled and shifted to range from 0 to 1. This mapping is trivially achieved by flipping the first bit of the posit and shifting it right two places, requiring perhaps four transistors of circuitry and easily fitting into a single clock cycle for a processor. The Machine Learning community presently has two options: It can calculate exponentials like $\frac{1}{1+e^{-x}}$ for this, requiring quite a few clock cycles:



…or, it can use a piecewise linear approximation, which can lead to problems with the convergence of the training.

Here is the one-cycle low-precision posit function plotted in magenta, and the $1/(e^{-x}+1)$ function in green. The slopes match at $x = 0$.



4.3 Restricting to the posit vocabulary with the overbar operator

The easiest way to prototype what a posit environment will do is to let *Mathematica* perform the operations, but always restrict the numerical vocabulary to what the posit environment supports. We can define the overbar function to do the restriction for us. Convert the real to the closest posit, then convert that back to a real to accomplish this; also, we make it a function that can work on a list of numbers, not just a single number, which will save a lot of code when we do linear algebra.

```
x_ := p2x[x2p[x]];
SetAttributes[OverBar, Listable]
```

For example, here is how we can find the nearest posit value of π in a {10, 1} environment:

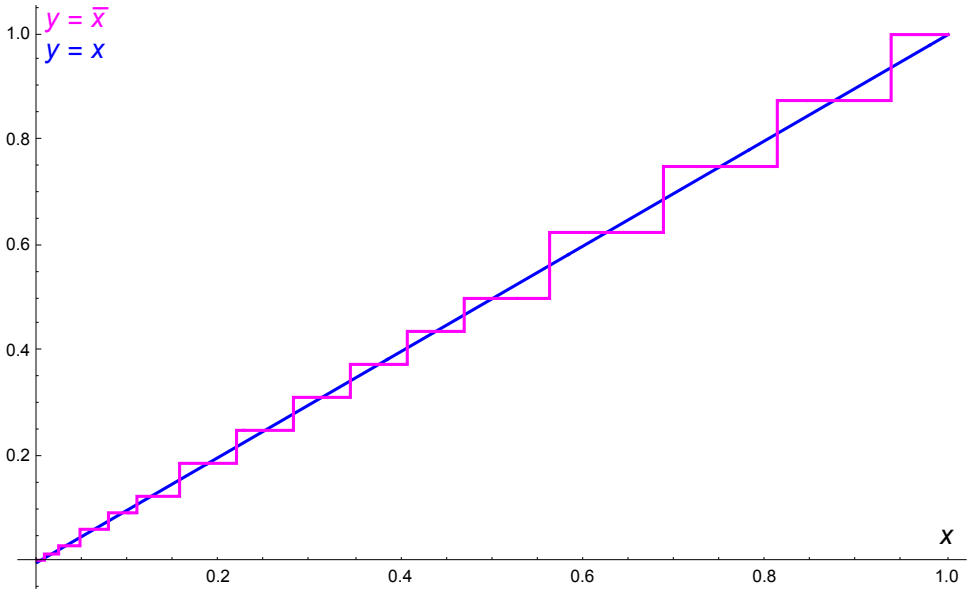
```
setpositenv[{10, 1}]
π
N[%]
```

$$\frac{101}{32}$$

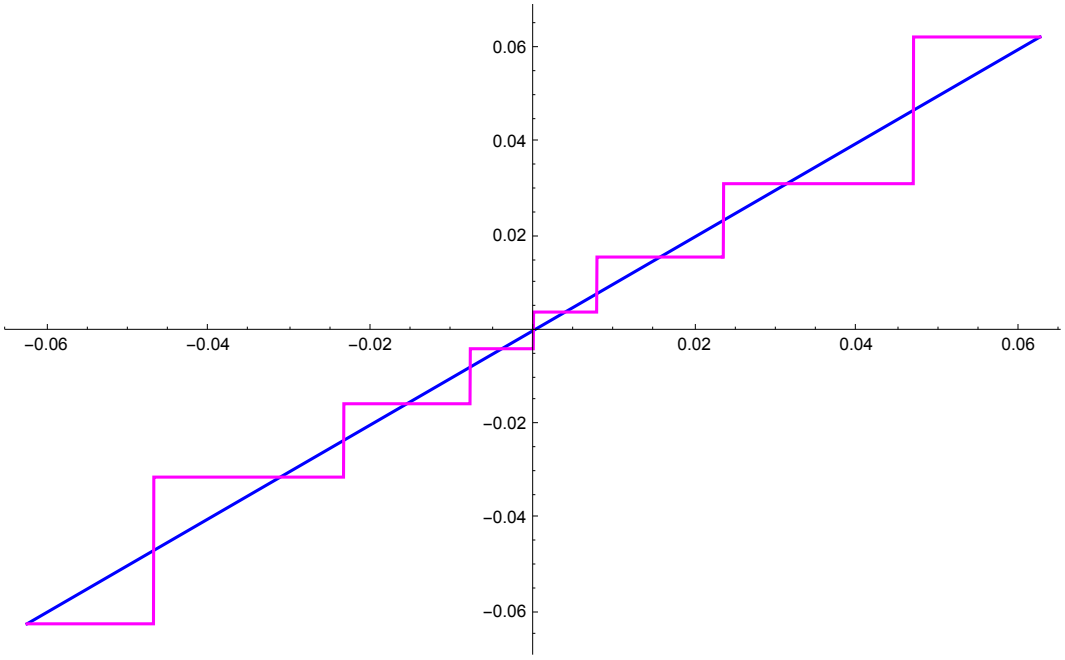
```
3.15625
```

Think of *Mathematica* as the "*g*-layer" where there are no rounding errors, and the overbar operator as what brings us back to the "*u*-layer" with unum representation of limited accuracy. This system will prove quite convenient as a rapid prototype for an actual posit computing environment.

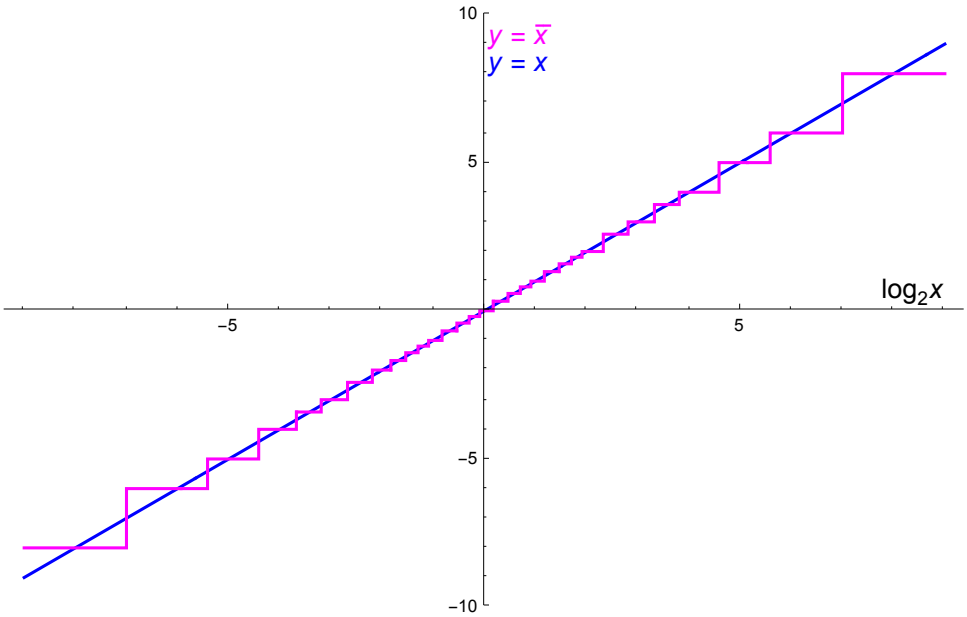
Since **x2p** and **p2x** are approximately inverse functions, we can compare plots of $y = x$ with $y = \overline{x}$ and get a stair-step function that criss-crosses the $y = x$ line. It looks pretty good on a linear plot from zero to one:



Test the range around $-minpos$ to $minpos$, to make sure that exceptions near zero are handled correctly. Notice that there is no "round to zero" for small numbers, or underflow:



For the range from $-maxpos$ to $maxpos$, a log-log plot is used to make the graph easier to read. Let's actually go from $-2\,maxpos$ to $2\,maxpos$, to check that values never "round" to infinity (overflow).



5 Creating an IEEE 754 float environment

5.1 The two parameters that specify a float environment

We can do something very similar for IEEE-type floats that we did to set up the posit environment. However, unlike IEEE floats, we can have very flexible sizes for the total number of bits, *nfbits*, and the number of bits in just the exponent, *esize*. The number of bits in the fraction, *fsize*, is always a constant since there is simply a sign bit, exponent bits, and fraction bits totaling to *nfbits*. The values we need for easy number conversion are things like the bias in the exponent, the smallest subnormal float, the smallest normalized float, the largest finite float, and the largest finite value that rounds down to the largest finite float instead of overflowing to infinity. (The smallest finite value that rounds up to the smallest subnormal is half the smallest subnormal; smaller values underflow to zero.)

```
setfloatenv[{n_Integer /; n ≥ 4, e_Integer /; e ≥ 2}] :=
 (
  {nfbits, esize, fsize} = {n, e, n - e - 1};
  bias = 2^(esize-1) - 1;
  smallsubnormal = 2^(1-bias-fsize);
  smallnormal = 2^(1-bias);
  maxfloat = 2^bias (1 + (2^fsize - 1)/2^fsize);
  minroundable = smallsubnormal / 2;
  maxroundable = 2^bias (1 + (2^fsize - 1 / 2)/2^fsize);
 )
```

For example, set an environment for 6 bits per number and 2 exponent bits:

```
setfloatenv[{6, 2}]
```

Here are the values set by that environment setting:

```
{nfbits, esize, fsize, bias, smallsubnormal,
 smallnormal, maxfloat, minroundable, maxroundable}
```

$$\left\{6, 2, 3, 1, \frac{1}{8}, 1, \frac{15}{4}, \frac{1}{16}, \frac{31}{8}\right\}$$

Notice that we can independently specify a posit environment and a float environment. The variable names do not overlap. This allows us to do side-by-side benchmarking of various algorithms with floats and with posits.

When floats have only one exponent bit, they act a lot like fixed-point numbers, because all the finite values are subnormal. That creates difficulties with the IEEE rules; for example, *maxfloat* becomes smaller than *smallnormal*, but the bit pattern for *smallnormal* is the one IEEE says must be used to represent infinity! The easiest fix to such craziness is simply to require that *esize* be 2 or greater. We also need the sign bit, and at least one fraction bit (otherwise there is no distinction between infinity values and NaN values). So the smallest possible value for *nfbits* is 4. Here are the values represented by the 16 possible bit patterns for *nfbits* = 4, just for fun:

0000	0	"Positive zero"
0001	$\frac{1}{2}$	*smallsubnormal*
0010	1	*smallnormal*
0011	$\frac{3}{2}$	
0100	2	
0101	3	*maxfloat*
0110	∞	Infinity
0111	NaN	Quiet NaN
1000	0	"Negative zero"
1001	$-\frac{1}{2}$	*–smallsubnormal*
1010	–1	*–smallnormal*
1011	$-\frac{3}{2}$	
1100	–2	
1101	–3	*–maxfloat*
1110	–∞	Minus infinity
1111	NaN	Signalling NaN

5.2 The function that converts a float to its numerical value: `f2x`

Now we need the equivalent of **p2x**, which we can call **f2x**, that takes a bit string representing a float (the first column in the above table) and returns its mathematical value (the second column). The routine is more complicated than for posits, because of having five exception cases, which is one reasons floats consume more circuitry than posits:

$$\text{Float value} = \begin{cases} e = \text{all } \mathbf{1} \text{ bits}, & \begin{cases} f = 0, & (-1)^s \infty, \\ f \neq 0, & \text{NaN of some kind}, \end{cases} \\ e = \text{all } \mathbf{0} \text{ bits}, & \begin{cases} f = 0, & \begin{cases} s = 0, & \text{"Positive zero"}, \\ s = 1, & \text{"Negative zero"}, \end{cases} \\ f \neq 0, & (-1)^s \times 2^{1-bias} \times f, \end{cases} \\ \text{all other } e, & (-1)^s \times 2^{e-bias} \times (1 + f). \end{cases}$$

where s = the sign bit, e is the integer represented by the exponent bits as an unsigned integer, and f is the fraction, *not* including the hidden bit. The following routine extracts the bit fields and then applies the above formula.

We cannot return “positive zero” and “negative zero” because they are not mathematical quantities. By IEEE rules, their reciprocals are positive infinity and negative infinity, respectively, yet they are supposed to test as *equal* to each other! Other quirky personality traits have been assigned to the two kinds of zero, such as requiring that the square root of “negative zero” be “negative zero.” It is little wonder that the handling of these two non-mathematical quantities are such a rich source of hardware design errors for the floating-point processing unit (FPU) in a processor. Here, we just assign zero to both cases (the mathematical kind of zero), but later on when figuring out closure plots, we again have to accommodate IEEE rules for the two concepts that are… zero. Sort of.

```
f2x[f_Integer /; 0 ≤ f < 2^nfbits] :=
 Module[{emask = FromDigits[Table[1, esize], 2],
   exp, fmask = FromDigits[Table[1, fsize], 2],
   frac, sgn, smask = BitShiftLeft[1, nfbits - 1]},
  sgn = BitShiftRight[BitAnd[smask, f], nfbits - 1];
  exp = BitAnd[BitShiftRight[f, fsize], emask];
  frac = BitAnd[fmask, f];
  Which[
   exp == emask, If[frac == 0, (-1)^sgn × ∞, Indeterminate],
   exp == 0, (-1)^sgn × 2^(1-bias) × (frac / 2^fsize),
   True, (-1)^sgn × 2^(exp-bias) × (1 + frac / 2^fsize)]]
```

5.3 Converting numbers into floats

We’re almost done; the final step is to create **x2f**, so that we have a way of converting a value (possibly NaN or infinite) into the IEEE-style float. A “floatable” value is a real number, or a signed infinity, or a NaN (which *Mathematica* calls **Indeterminate**). Here’s the test for that:

```
floatableQ[x_] := (x === Indeterminate ⋁ x === ∞ ⋁ x === -∞ ⋁ x ∈ Reals)
```

We can use that to guard the conversion function from trying to do something it shouldn't attempt.

```
x2f[x_ /; floatableQ[x]] := Module[{e = 0, f, sgn, y},
  Which[
   x === Indeterminate ⋁ ¬ floatableQ[x],
   FromDigits[Table[1, nfbits], 2], (* NaN exceptions *)
   y = Abs[x];
   sgn = BitShiftLeft[Boole[x < 0], nfbits - 1];
   y == 0, 0, (* Zero exception *)
   y ≥ maxroundable,
   BitOr[sgn, BitShiftLeft[FromDigits[Table[1, esize], 2], fsize]],
   (* Infinity and overflow exceptions *)
   y < smallnormal, BitOr[sgn, Round[y / smallsubnormal]],
   (* subnormal exceptions *)
   True, (* Else x is normal. Find the exponent and fraction fields. *)
   (* At most one of the next two While loops will execute. *)
   While[y ≥ 2, y /= 2; e++];
   While[y < 1, y *= 2; e--];
   (* We now have 1 ≤ y < 2. *)
   f = Round[(y - 1) * 2^fsize];
   (* This may round up to 2, so we add f, instead of ORing it. *)
   BitOr[sgn, BitShiftLeft[bias + e, fsize]] + f]]
```

It may become handy to have a **colorcodef** operator to see the bit fields of a float, just as we did with posits.

```
colorcodef[f_ /; floatableQ[f]] :=
 Module[{fbits = IntegerDigits[f, 2, nfbits]},
  Row[{ Style[fbits[[1]], ■], "", Style[Row[Take[fbits, {2, esize + 1}]], ■],
    "", Row[Take[fbits, -nfbits + esize + 1]]}]]
```

We last left the **setfloatenv** setting at {6, 2}. Let's test the color-coding of a float represented with **001101**, which as a float in this tiny environment represents 1.625:

```
colorcodef[2^^001101]
N[f2x[2^^001101]]
```

```
001101

1.625
```

5.4 Restriction to a float vocabulary: the underbar operator

Just as we defined an overbar operator to restrict the numerical vocabulary to posits, we can

define the underbar to restrict values to floats:

```
x_ := f2x[x2f[x]]
SetAttributes[UnderBar, Listable]
```

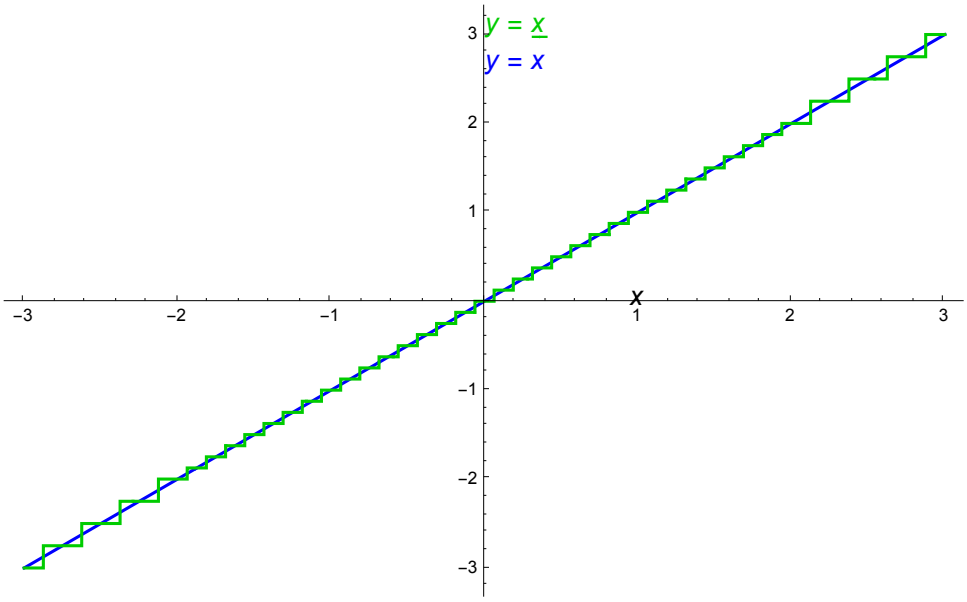
Again use π as an example, within the current {6, 2} float environment:

```
π
N[%]
```

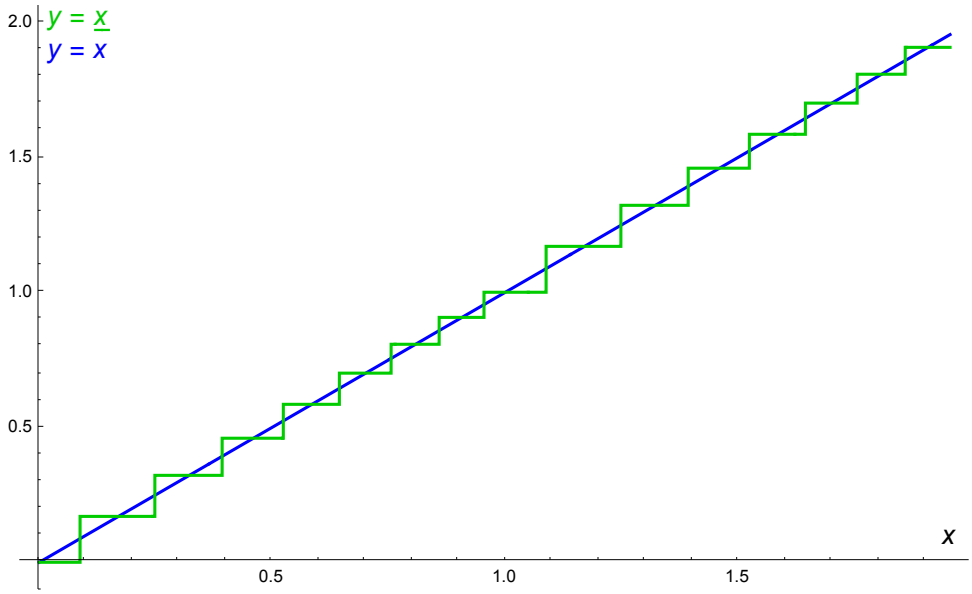
$$\frac{13}{4}$$

```
3.25
```

Let's do two quick graphics tests like the ones done for posits. First, the graph near zero, including all the subnormal values:



And now the graph from 1 to *maxfloat*, where we use a log-log plot to better show the zig-zag over a changing scale:



It is worth noting that for posits, it is always true the **x2p[p2x[*p*]]** returns *p*, whereas for floats, it is *not* possible to achieve such a perfect inverse with **x2f[f2x[*f*]]** and *f*. Exceptions occur for "negative zero" and for the multiple ways of representing NaN.

6 Floats vs. Posits Preview: Accuracy on a 32-bit budget

Before we go through a litany of comparisons of posits and floats, let's have a pre-game match. If we evaluate the expression

$$\left(\frac{27/10-e}{\pi-\left(\sqrt{2}+\sqrt{3}\right)}\right)^{67/16}$$

correct to ten decimals, it is 302.8827196⋯. We can compare the accuracy we get with 32-bit standard IEEE floats and 32-bit posits that have at least as large a dynamic range.

First, try the single-precision float environment, which by the IEEE Standard uses 8 bits for the exponent:

```
setfloatenv[{32, 8}]
```

IEEE 32-bit floats have an unbalanced dynamic range of about 83 decades:

```
N[{smallsubnormal, maxfloat}]
Log[10., maxfloat / smallsubnormal]
```

$\{1.4013 \times 10^{-45}, 3.40282 \times 10^{38}\}$

83.3853

For each step in the calculation, the *Mathematica*-type real values are turned into floats, which means in general they are rounded. The **accbenchf** and **accbenchp** functions do not take an argument, since they rely only on the environment settings. Notice how the underscore and overbar notations let us write normal-looking equations, and explicitly round every result to a number within the float or posit vocabulary.

```
accbenchf := Module[{te = e, t2710 = 27 / 10, tpi = π,
    tr2 = √2, tr3 = √3, tnum, t1, tden, t2, t6716, tans},
  tnum = t2710 - te;
  t1 = tr2 + tr3;
  tden = tpi - t1;
  t2 = tnum / tden;
  t6716 = 67 / 16;
  tans = t2^t6716;
  Print[Style[Row[{"Float answer to ", nfbits - esize, " decimals: "}],
    "Text"], N[tans, nfbits - esize]];
  Print[Style["Rounding error in answer: ", "Text"],
   N[((27 / 10 - e) / (π - (√2 + √3)))^(67/16) - tans]]]
```

Try it out on IEEE Standard 32-bit floats, single precision:

```
accbenchf
```

Float answer to 24 decimals: 302.912414550781250000000

Rounding error in answer: -0.0296949

IEEE single precision gives us the answer 302.912⋯, which is off by about 0.0297. We use the convention of coloring the decimal digits orange that differ from the decimal digits representing the correct answer. Although single precision floats have a nominal accuracy of about seven decimals, that accuracy quickly erodes away in this simple expression, and we are left with only three correct decimals.

Now try posits. The underscores turn to overscores, and the calculation of the denominator **tden** has rounding deferred, since posits can sum long lists of numbers in the quire (more about that later).

```
accbenchp := Module[{te = e, t2710 = 27 / 10, tpi = π,
    tr2 = √2 , tr3 = √3 , tnum, t1, tden, t2, t6716, tans},
  tnum = t2710 - te;
  tden = tpi - (tr2 + tr3); (* Posits allow this kind of fused operation *)
  t2 = tnum / tden;
  t6716 = 67 / 16;
  tans = t2^t6716;
  Print[Style[Row[{"Posit answer with es = ", es, ": "}], "Text"],
   N[tans, nbits - es - 9]];
  Print[Style["Rounding error in answer: ", "Text"],
   N[((27 / 10 - e) / (π - (√2 + √3)))^(67/16) - tans], "\n"]]
```

Let's try a **range** of *es* values to see where the sweet spot is.

```
For[i = 0, i ≤ 5, i++, setpositenv[{32, i}]; accbenchp]
```

Posit answer with *es* = 0: 302.88171386718750000000
Rounding error in answer: 0.00100579

Posit answer with *es* = 1: 302.8827819824218750000
Rounding error in answer: -0.0000623271

Posit answer with *es* = 2: 302.883300781250000000
Rounding error in answer: -0.000581126

Posit answer with *es* = 3: 302.88231658935546875
Rounding error in answer: 0.000403066

Posit answer with *es* = 4: 302.8883361816406250
Rounding error in answer: -0.00561653

Posit answer with *es* = 5: 302.872161865234375
Rounding error in answer: 0.0105578

All of these answers are much more accurate than the float answer! When *es* = 1, the posit answer has four more correct decimals for the same number of bits, 302.882781⋯. The error is smaller than the float error by a factor of almost *500*. When *es* = 3, for which the dynamic range is almost twice that of floats (144 decades instead of 83), the posits are more accurate by about a factor of 74.

This "Accuracy on a 32-Bit Budget" example gives a preview of why switching from floats to posits might be worth the trouble. There may be many applications where 32-bit floats are *not quite*

accurate enough, forcing programmers to jump all the way to 64-bit floats instead. If 32-bit posits can achieve more decimals of accuracy, they may allow the use of 32-bit variables and thereby provide a 2x savings in bandwidth, storage, and the energy and power needed to move the data.

The next section will create two complete number sets with float rules and posit rules, and compare properties of their complete tables for basic arithmetic.

7 Posits versus IEEE-type Floats: A Metric-Based Study

7.1 Why Regime Bits increase *Accuracy*

It is obvious that regime bits are a sort of “super-exponent” that amplify the dynamic range of posits compared to floats. What is less obvious is that they also *increase accuracy* by allowing more bits “left over” to express the fraction part of a number.

As an example, suppose we have 16 bits and wish to express the number 10 000. Half-precision IEEE floats (5-bit exponent field) can do that without rounding:

```
setfloatenv[{16, 5}]
10 000
```

```
10 000
```

As a float, $10\,000 = 2^{13} \times \left(1 + \frac{113}{512}\right)$. That means we need an exponent that ranges from 2^{-14} to 2^{14} for normalized floats, which requires an exponent with at least five bits. If we instead represent the number as a 16-bit posit, consider how 2^{13} would be expressed with different values of *es*, leaving off the all-zero fraction bits for clarity:

es	2^{13}
0	1111111111111110
1	11111101
2	1111001
3	110101
4	101101
5	1001101
6	10001101
7	100001101

etc.

Those are all ways of expressing the integer 13 as the bit shift (scaling by a power of 2). The number of regime bits needed to express that integer repeatedly drops by about a factor of two initially, and then when it hits the minimum two-bit 10 pattern, the exponent bits take over the job and the total number of exponent bits ramps up linearly. Hence, there is a "sweet spot" if we intend to be dealing a lot with quantities near 10 000 in magnitude. Only $es = 3$ or $es = 4$ allow enough fraction bits to let 10000 be represented exactly:

es = 0	0111111111111110→+111111111111110	closest value to 10000: 8192
es = 1	0111111101001110→+111111101001110	closest value to 10000: 9984
es = 2	0111100100111000→+111100100111000	closest value to 10000: 9984
es = 3	0110101001110001→+110101001110001	closest value to 10000: 10000
es = 4	0101101001110001→+101101001110001	closest value to 10000: 10000
es = 5	0100110100111000→+100110100111000	closest value to 10000: 9984
es = 6	0100011010011100→+100011010011100	closest value to 10000: 9984
es = 7	0100001101001110→+100001101001110	closest value to 10000: 9984
es = 8	0100000110100111→+100000110100111	closest value to 10000: 9984

For *es* = 3, the dynamic range goes far beyond what 16-bit floats can express and is probably overkill for most low-precision applications:

```
setpositenv[{16, 3}];
N[{minpos, maxpos}]
Log[10., maxpos / minpos]
```

$\{1.92593 \times 10^{-34}, 5.1923 \times 10^{33}\}$

```
67.4307
```

That's enough to do a decent job of representing Avogadro's number, in case anyone wants to do low-accuracy but very quick chemistry calculations on molar quantities!

```
N[Overscript[6.022 × 10^23, _], 3]
```

6.04×10^{23}

If someone is using posit arithmetic in a field-programmable gate array (FPGA) for a particular embedded application where it need not be compatible with a standard storage format, such as signal processing or machine learning, there is no reason why a posit cannot have, say, 23 bits with an *es* of 6. However, we want to make the transition away from the outdated 1985 float format as painless as possible, which means standardizing the *es* values for 16-, 32-, 64-, 128-, and 256-bit posits. An ambitious hardware engineer might make the *es* value configurable, but then there is the issue of language support for the many different data types. Ideally, a C program written many years ago that uses `float` and `double` keywords could simply be recompiled to use 32-bit and 64-bit posits, with a different math library (the `libm.h` include file), and everything should just work. And produce more accurate answers.

7.2 Should we match IEEE float dynamic ranges?

In the current (2008) version of the IEEE 754 standard, there are five binary float sizes: 16, 32, 64, 128, and 256 bits. For some users, it *might* be important that posits do as well or better than floats for dynamic range, lest they appear to be sacrificing dynamic range in favor of accuracy.

Here is a table of the IEEE floats and the same size posits, with *es* chosen to make the dynamic range better for sizes 16 and 32 bits, and very nearly as large for 64, 128, and 256 bits. It may be

important to *improve* the dynamic ranges for 16-bit and 32-bit posits if they are to be used as replacements, say, for 32-bit and 64-bit floats, respectively.

Size, bits	Float exponent size	Float dynamic range	Posit es value	Posit dynamic range
16	5	$3.\times10^{-8}$ to $7.\times10^{4}$	1	$9.\times10^{-10}$ to $1.\times10^{9}$
32	8	$7.\times10^{-46}$ to $3.\times10^{38}$	3	$2.\times10^{-75}$ to $5.\times10^{74}$
64	11	$2.\times10^{-324}$ to $2.\times10^{308}$	4	$4.\times10^{-304}$ to $3.\times10^{303}$
128	15	$3.\times10^{-4966}$ to $1.\times10^{4932}$	7	$3.\times10^{-4894}$ to $3.\times10^{4893}$
256	19	$1.\times10^{-78\,984}$ to $2.\times10^{78\,913}$	10	$1.\times10^{-78\,605}$ to $9.\times10^{78\,604}$

These exponent sizes for floats do not follow any mathematical pattern, but reflect intensely-argued compromises by the IEEE committee. Trying to match the 1985-era choices of the IEEE committee results in an equally inexplicable set of *es* values for the posits. Frankly, the reason for such enormous dynamic ranges is that they were trying to save transistors instead of provide what users really need. Multiplying and dividing floats only requires integer addition and subtraction of the exponent field, but the fraction field needs an integer multiplier, and the cost of that can grow almost as the square of the number of fraction bits. So while almost no one strays outside the range 10^{-13} to 10^{13} in real applications, the IEEE 754 Standard proudly lets you go from about $10^{-78\,984}$ to $10^{78\,913}$. Even astrophysicists don't know what to do with dynamic ranges that huge. Ironically, whatever those decision-makers saved in the size of the integer multiplier, they more than made up for it with a long list of burdensome exception cases that consume plenty of logic (like negative zero, subnormal exceptions, NaN and infinity bit patterns, multiple rounding modes, and the requirement of processor flags for overflow, underflow, and inexact results).

For the moment, suppose the IEEE choices are justified. The posit method of expressing the power-of-two scaling factor frees up more bits for the fraction over a wide range, giving a greater maximum accuracy for the above choices of *es*:

Size, bits	Float maximum accuracy, bits	Posit maximum accuracy, bits	Posit advantage, bits	Range where posit accuracy is ≥ float accuracy
16	11	13	2	$\frac{1}{64}$ to 64
32	24	27	3	$2.\times10^{-10}$ to $4.\times10^{9}$
64	53	58	5	$1.\times10^{-29}$ to $8.\times10^{28}$
128	113	119	6	$2.\times10^{-270}$ to $5.\times10^{269}$
256	237	244	7	9×10^{-2467} to 1.1×10^{2466}

So even if we buy into the transistor-pinching choices of IEEE 754, posits easily beat floats for accuracy.

What if we take a more reasoned approach, and simply **increment** the *es* value every time we double the number of bits of precision? Every such increment results in a fourfold increase in the dynamic range, which certainly should be ample. For purposes of comparison with 8-bit posits, let's imagine a "quarter-precision" IEEE float with 3 exponent bits so we can include that size in the table.

Size, bits	Float exponent size	Float dynamic range	Posit *es* value	Posit dynamic range
8	3	0.008 to $2.\times10^{1}$	0	0.008 to $1.\times10^{2}$
16	5	$3.\times10^{-8}$ to $7.\times10^{4}$	1	$9.\times10^{-10}$ to $1.\times10^{9}$
32	8	$7.\times10^{-46}$ to $3.\times10^{38}$	2	$5.\times10^{-38}$ to $2.\times10^{37}$
64	11	$2.\times10^{-324}$ to $2.\times10^{308}$	3	$2.\times10^{-152}$ to $5.\times10^{151}$
128	15	$3.\times10^{-4966}$ to $1.\times10^{4932}$	4	$2.\times10^{-612}$ to $5.\times10^{611}$
256	19	$1.\times10^{-78\,984}$ to $2.\times10^{78\,913}$	5	$4.\times10^{-2457}$ to $3.\times10^{2456}$

Here is the posit accuracy advantage if we do *not* try to match the oversized dynamic ranges of IEEE and simply use 0-1-2-3-4-5 as the *es* sizes.

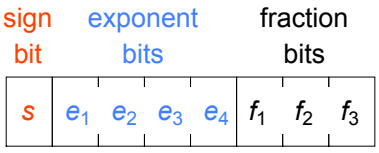
Size, bits	Float maximum accuracy, bits	Posit maximum accuracy, bits	Posit advantage, bits	Range where posit accuracy is ≥ float accuracy
8	5	6	1	$\frac{1}{4}$ to 4
16	11	13	2	$\frac{1}{64}$ to 64
32	24	28	4	$1.\times10^{-6}$ to $1.\times10^{6}$
64	53	59	6	$1.\times10^{-17}$ to $7.\times10^{16}$
128	113	122	9	$7.\times10^{-49}$ to $1.\times10^{48}$
256	237	249	12	$6.\times10^{-126}$ to $2.\times10^{125}$

Somehow, that looks a lot more sane. Notice that for a 32-bit posit, the four extra accuracy bits would make them comparable to an old-fashioned 36-bit float from the days before IBM introduced its System 360 in the 1960s. It will take some time to get sufficient feedback from the HPC community to decide this issue, but right now my vote is for the second set of *es* settings. The *es* value should simply be

$$es = \log_2(nbits) - 3$$

7.3 Constructing the low-precision sets to compare

Let's construct some eight-bit "quarter precision" floats that follow IEEE rules, and then an eight-bit posit with a comparable dynamic range. We use such low precision to make it practical to find the entire set of values and work with tables formed from pairs of them, which will have 65 536 entries. Here is what a quarter-precision IEEE-style float might look like (with a 4-bit exponent, instead of the 3-bit exponent used in the tables in the previous section):



Set the float environment to 8-bit values with 4-bit exponent fields:

```
setfloatenv[{8, 4}]
```

The **floatfix** function turns counting integers into two sets of integers that puts the floats repre-

sented by the bit strings into increasing order.

```
floatfix[i_Integer] := If[i < 2^nfbits-1, 2^nfbits - i - 1, i - 2^nfbits-1]
```

The **float8** list is the set of floats represented by every possible bit pattern. Printing the list with TableForm is a trick to make the fractions automatically typeset with a smaller font size than the integers.

```
float8 = Table[f2x[floatfix[f]], {f, 0, 2^nfbits - 1}];
Table[TableForm[{float8[[i]]}], {i, 1, Length[float8]}]
```

$$\{\text{Indeterminate}, \text{Indeterminate}, \text{Indeterminate}, \text{Indeterminate}, \text{Indeterminate}, \text{Indeterminate}, \text{Indeterminate}, -\infty, -240, -224, -208, -192, -176, -160, -144, -128, -120, -112, -104, -96, -88, -80, -72, -64, -60, -56, -52, -48, -44, -40, -36, -32, -30, -28, -26, -24, -22, -20, -18, -16, -15, -14, -13, -12, -11, -10, -9, -8, -\frac{15}{2}, -7, -\frac{13}{2}, -6, -\frac{11}{2}, -5, -\frac{9}{2}, -4, -\frac{15}{4}, -\frac{7}{2}, -\frac{13}{4}, -3, -\frac{11}{4}, -\frac{5}{2}, -\frac{9}{4}, -2, -\frac{15}{8}, -\frac{7}{4}, -\frac{13}{8}, -\frac{3}{2}, -\frac{11}{8}, -\frac{5}{4}, -\frac{9}{8}, -1, -\frac{15}{16}, -\frac{7}{8}, -\frac{13}{16}, -\frac{3}{4}, -\frac{11}{16}, -\frac{5}{8}, -\frac{9}{16}, -\frac{1}{2}, -\frac{15}{32}, -\frac{7}{16}, -\frac{13}{32}, -\frac{3}{8}, -\frac{11}{32}, -\frac{5}{16}, -\frac{9}{32}, -\frac{1}{4}, -\frac{15}{64}, -\frac{7}{32}, -\frac{13}{64}, -\frac{3}{16}, -\frac{11}{64}, -\frac{5}{32}, -\frac{9}{64}, -\frac{1}{8}, -\frac{15}{128}, -\frac{7}{64}, -\frac{13}{128}, -\frac{3}{32}, -\frac{11}{128}, -\frac{5}{64}, -\frac{9}{128}, -\frac{1}{16}, -\frac{15}{256}, -\frac{7}{128}, -\frac{13}{256}, -\frac{3}{64}, -\frac{11}{256}, -\frac{5}{128}, -\frac{9}{256}, -\frac{1}{32}, -\frac{15}{512}, -\frac{7}{256}, -\frac{13}{512}, -\frac{3}{128}, -\frac{11}{512}, -\frac{5}{256}, -\frac{9}{512}, -\frac{1}{64}, -\frac{7}{512}, -\frac{3}{256}, -\frac{5}{512}, -\frac{1}{128}, -\frac{3}{512}, -\frac{1}{256}, -\frac{1}{512}, 0, 0, \frac{1}{512}, \frac{1}{256}, \frac{3}{512}, \frac{1}{128}, \frac{5}{512}, \frac{3}{256}, \frac{7}{512}, \frac{1}{64}, \frac{9}{512}, \frac{5}{256}, \frac{11}{512}, \frac{3}{128}, \frac{13}{512}, \frac{7}{256}, \frac{15}{512}, \frac{1}{32}, \frac{9}{256}, \frac{5}{128}, \frac{11}{256}, \frac{3}{64}, \frac{13}{256}, \frac{7}{128}, \frac{15}{256}, \frac{1}{16}, \frac{9}{128}, \frac{5}{64}, \frac{11}{128}, \frac{3}{32}, \frac{13}{128}, \frac{7}{64}, \frac{15}{128}, \frac{1}{8}, \frac{9}{64}, \frac{5}{32}, \frac{11}{64}, \frac{3}{16}, \frac{13}{64}, \frac{7}{32}, \frac{15}{64}, \frac{1}{4}, \frac{9}{32}, \frac{5}{16}, \frac{11}{32}, \frac{3}{8}, \frac{13}{32}, \frac{7}{16}, \frac{15}{32}, \frac{1}{2}, \frac{9}{16}, \frac{5}{8}, \frac{11}{16}, \frac{3}{4}, \frac{13}{16}, \frac{7}{8}, \frac{15}{16}, 1, \frac{9}{8}, \frac{5}{4}, \frac{11}{8}, \frac{3}{2}, \frac{13}{8}, \frac{7}{4}, \frac{15}{8}, 2, \frac{9}{4}, \frac{5}{2}, \frac{11}{4}, 3, \frac{13}{4}, \frac{7}{2}, \frac{15}{4}, 4, \frac{9}{2}, 5, \frac{11}{2}, 6, \frac{13}{2}, 7, \frac{15}{2}, 8, 9, 10, 11, 12, 13, 14, 15, 16, 18, 20, 22, 24, 26, 28, 30, 32, 36, 40, 44, 48, 52, 56, 60, 64, 72, 80, 88, 96, 104, 112, 120, 128, 144, 160, 176, 192, 208, 224, 240, \infty, \text{Indeterminate}, \text{Indeterminate}, \text{Indeterminate}, \text{Indeterminate}, \text{Indeterminate}, \text{Indeterminate}, \text{Indeterminate}\}$$

That set has 256 elements in it, but 14 of the elements express NaN (`Indeterminate`). Also, notice that the value 0 occurs twice in the above set because of “negative zero.” The smallest positive value for the floats is 1/512 and the largest value is 240, giving them a dynamic range of about five decades.

Now construct posits with *nbits* = 8. If we choose *es* = 1, notice the values of *minpos* and *maxpos* exceed the dynamic range of the floats:

```
setpositenv[{8, 1}]
{minpos, maxpos}
```

$$\left\{\frac{1}{4096}, 4096\right\}$$

Again generate the table as if we had 2's complement integers, from most negative to most positive. The **positfix** function makes an adjustment like that made by **floatfix**, to put the posits into increasing order.

```
positfix[i_Integer] := If[i < npat / 2, i + npat / 2, i - npat / 2]
posit8 = Table[p2x[positfix[j]], {j, 0, npat - 1}];
Table[TableForm[{posit8[[i]]}], {i, 1, npat}]
```

$$\{\text{ComplexInfinity}, -4096, -1024, -512, -256, -192, -128, -96, -64,$$
$$-56, -48, -40, -32, -28, -24, -20, -16, -15, -14, -13, -12,$$
$$-11, -10, -9, -8, -\frac{15}{2}, -7, -\frac{13}{2}, -6, -\frac{11}{2}, -5, -\frac{9}{2}, -4, -\frac{31}{8},$$
$$-\frac{15}{4}, -\frac{29}{8}, -\frac{7}{2}, -\frac{27}{8}, -\frac{13}{4}, -\frac{25}{8}, -3, -\frac{23}{8}, -\frac{11}{4}, -\frac{21}{8}, -\frac{5}{2}, -\frac{19}{8}, -\frac{9}{4},$$
$$-\frac{17}{8}, -2, -\frac{31}{16}, -\frac{15}{8}, -\frac{29}{16}, -\frac{7}{4}, -\frac{27}{16}, -\frac{13}{8}, -\frac{25}{16}, -\frac{3}{2}, -\frac{23}{16}, -\frac{11}{8}, -\frac{21}{16},$$
$$-\frac{5}{4}, -\frac{19}{16}, -\frac{9}{8}, -\frac{17}{16}, -1, -\frac{31}{32}, -\frac{15}{16}, -\frac{29}{32}, -\frac{7}{8}, -\frac{27}{32}, -\frac{13}{16}, -\frac{25}{32}, -\frac{3}{4},$$
$$-\frac{23}{32}, -\frac{11}{16}, -\frac{21}{32}, -\frac{5}{8}, -\frac{19}{32}, -\frac{9}{16}, -\frac{17}{32}, -\frac{1}{2}, -\frac{31}{64}, -\frac{15}{32}, -\frac{29}{64}, -\frac{7}{16}, -\frac{27}{64},$$
$$-\frac{13}{32}, -\frac{25}{64}, -\frac{3}{8}, -\frac{23}{64}, -\frac{11}{32}, -\frac{21}{64}, -\frac{5}{16}, -\frac{19}{64}, -\frac{9}{32}, -\frac{17}{64}, -\frac{1}{4}, -\frac{15}{64}, -\frac{7}{32},$$
$$-\frac{13}{64}, -\frac{3}{16}, -\frac{11}{64}, -\frac{5}{32}, -\frac{9}{64}, -\frac{1}{8}, -\frac{15}{128}, -\frac{7}{64}, -\frac{13}{128}, -\frac{3}{32}, -\frac{11}{128}, -\frac{5}{64},$$
$$-\frac{9}{128}, -\frac{1}{16}, -\frac{7}{128}, -\frac{3}{64}, -\frac{5}{128}, -\frac{1}{32}, -\frac{7}{256}, -\frac{3}{128}, -\frac{5}{256}, -\frac{1}{64}, -\frac{3}{256},$$
$$-\frac{1}{128}, -\frac{3}{512}, -\frac{1}{256}, -\frac{1}{512}, -\frac{1}{1024}, -\frac{1}{4096}, 0, \frac{1}{4096}, \frac{1}{1024}, \frac{1}{512}, \frac{1}{256}, \frac{3}{512},$$
$$\frac{1}{128}, \frac{3}{256}, \frac{1}{64}, \frac{5}{256}, \frac{3}{128}, \frac{7}{256}, \frac{1}{32}, \frac{5}{128}, \frac{3}{64}, \frac{7}{128}, \frac{1}{16}, \frac{9}{128}, \frac{5}{64}, \frac{11}{128},$$
$$\frac{3}{32}, \frac{13}{128}, \frac{7}{64}, \frac{15}{128}, \frac{1}{8}, \frac{9}{64}, \frac{5}{32}, \frac{11}{64}, \frac{3}{16}, \frac{13}{64}, \frac{7}{32}, \frac{15}{64}, \frac{1}{4}, \frac{17}{64}, \frac{9}{32},$$
$$\frac{19}{64}, \frac{5}{16}, \frac{21}{64}, \frac{11}{32}, \frac{23}{64}, \frac{3}{8}, \frac{25}{64}, \frac{13}{32}, \frac{27}{64}, \frac{7}{16}, \frac{29}{64}, \frac{15}{32}, \frac{31}{64}, \frac{1}{2}, \frac{17}{32}, \frac{9}{16},$$
$$\frac{19}{32}, \frac{5}{8}, \frac{21}{32}, \frac{11}{16}, \frac{23}{32}, \frac{3}{4}, \frac{25}{32}, \frac{13}{16}, \frac{27}{32}, \frac{7}{8}, \frac{29}{32}, \frac{15}{16}, \frac{31}{32}, 1, \frac{17}{16}, \frac{9}{8},$$
$$\frac{19}{16}, \frac{5}{4}, \frac{21}{16}, \frac{11}{8}, \frac{23}{16}, \frac{3}{2}, \frac{25}{16}, \frac{13}{8}, \frac{27}{16}, \frac{7}{4}, \frac{29}{16}, \frac{15}{8}, \frac{31}{16}, 2, \frac{17}{8}, \frac{9}{4},$$
$$\frac{19}{8}, \frac{5}{2}, \frac{21}{8}, \frac{11}{4}, \frac{23}{8}, 3, \frac{25}{8}, \frac{13}{4}, \frac{27}{8}, \frac{7}{2}, \frac{29}{8}, \frac{15}{4}, \frac{31}{8}, 4, \frac{9}{2}, 5,$$
$$\frac{11}{2}, 6, \frac{13}{2}, 7, \frac{15}{2}, 8, 9, 10, 11, 12, 13, 14, 15, 16, 20, 24,$$
$$28, 32, 40, 48, 56, 64, 96, 128, 192, 256, 512, 1024, 4096\}$$

For posits, there are *no* wasted cases. All bit patterns represent unique mathematical quantities. This relates to another violation of mathematics committed by IEEE floats. If $a = b$, then for any function f we expect $f(a) = f(b)$. But the IEEE rules send us down the rabbit hole by declaring positive and negative zero to be numerically equal, yet $1/x$ is $-\infty$ for "negative zero" and $+\infty$ for "positive zero." Which implies that negative infinity is the same as positive infinity. Gulp.

For posits, there are *no* wasted cases.

7.4 A careful definition of "decimal accuracy"

Like so many things in numerical analysis, we have gotten accustomed to some concepts that are widely accepted but rather sloppy in their logic. The way we measure "error" of various types is such a concept. For instance, this is a widely-accepted definition:

$$\textit{absolute error} = \left| x_{\text{computed}} - x_{\text{exact}} \right|$$

That seems reasonable at first glance; if the values are identical, their difference is zero and there is no error. But if a calculation produces 315 instead of 314, doesn't the error look very similar to returning 3.15 instead of 3.14? A number system designed for real numbers usually spans many decades of dynamic range, but simply subtracting numbers looks more like an integer or fixed-point way of looking at inaccuracy.

In an attempt to repair this contradiction, the *relative error* of a computed value is usually defined as

$$\textit{relative error} = \left| \frac{x_{\text{computed}} - x_{\text{exact}}}{x_{\text{exact}}} \right|$$

Here are a couple reasons you should be dissatisfied with this definition. For one, if you compute –1 when the correct answer is 100, then the relative error would only be 1.01. If you don't even know which half of the projective real circle a result is on, then you know essentially *nothing* about the answer since the sign is the most significant part of a number. An error formula should refuse to function in such cases. Even declaring the relative error to be infinite is too flattering.

For another, the formula is quite different for numbers and their inverses. Suppose $x_{\text{computed}} = 0.001$ but $x_{\text{exact}} = 0.0001$. Then the relative error is 9. But if we were instead computing the reciprocals of the numbers, the relative error would take $x_{\text{computed}} = 1000$ and $x_{\text{exact}} = 10\,000$ as inputs, and the above formula would reassure us that the relative error is only 0.9. It makes no sense that an answer can be made to look more accurate simply by taking the reciprocal. If you knew the miles per gallon of a car with a relative error of 0.1, would it not bother you that you know the gallons per mile of the same car with a relative error of 0.2? That's why it is more common to say something is accurate to some percentage, like "within five percent" which is certainly better because it invokes ratios instead of differences.

Engineers have long had a solution when comparing numbers with ratios, which is to use *decibels*. A ratio of 10 is 10 decibels (dB). A ratio of 1 dB means the ratio is $10^{1/10} = 1.2589\cdots$. As with decibels, we should be looking at the difference of the *logarithms* of the numbers, which is the same as the logarithm of their ratio:

$$\textit{decimal error} \equiv \left| \log_{10}\left(x_{\text{computed}}\right) - \log_{10}\left(x_{\text{exact}}\right) \right| = \left| \log_{10}\left(\frac{x_{\text{computed}}}{x_{\text{exact}}} \right) \right|.$$

Notice that the absolute value makes x_{computed} and x_{exact} interchangeable in the above definition. Also notice that it produces the same result whether you use x_{computed} and x_{exact} as inputs, or $1/x_{\text{computed}}$ and $1/x_{\text{exact}}$. So *that* looks like a mathematically sound definition.

We could achieve those properties with any base logarithm, but we choose base 10 because it measures the error in *decades*, the same human-friendly way we measure dynamic range. Recall the above example of $x_{\text{computed}} = 0.001$ but $x_{\text{exact}} = 0.0001$; the decimal error is 1. That means it's a decade off.

The decimal error can be used to define *decimal accuracy*. Accuracy is the inverse of error. If we want to know the number of decimals of accuracy, we again take the log base 10.

$$\textit{decimal accuracy} \equiv \log_{10}\left(\frac{1}{\text{decimal error}}\right) = -\log_{10}\left|\log_{10}\left(\frac{x_{\text{computed}}}{x_{\text{exact}}}\right)\right|.$$

Here is the *Mathematica* function, which uses the above formula but also handles exceptional input values. If either input is NaN, then the accuracy is NaN. It also produces NaN if the numbers are of opposite sign. Otherwise, if the input values are identical, the accuracy is ∞. If only one input is ∞ or only one input is 0, the accuracy is –∞, because on a logarithmic scale, both 0 and ∞ are *infinitely* far away from any nonzero real number. This is why it is a disaster to "round" large results to infinity (overflow) or small results to zero (underflow).

```
decacc[x_, y_] := Which[
  x === Indeterminate ⋁ y === Indeterminate, Indeterminate,
  x === y, ∞,
  x === ComplexInfinity ⋁ y === ComplexInfinity, Indeterminate,
  (x < 0 ⋀ y > 0) ⋁ (x > 0 ⋀ y < 0), Indeterminate,
  True, N[-Log[10, Abs[Log[10, x / y]]]]]
```

If the numbers are a decade apart, the accuracy is zero; we don't even know the order of magnitude of the result:

```
decacc[23, 230]
```

```
0.
```

Engineers should like this property of the definition: Here is the decimal accuracy if an answer is off by 1 dB:

```
decacc[1, 10^(1/10)]
```

```
1.
```

If an answer is off by 0.1 dB, then we have 2 decimals of accuracy; off by 0.01 dB means 3 decimals of accuracy, and so on.

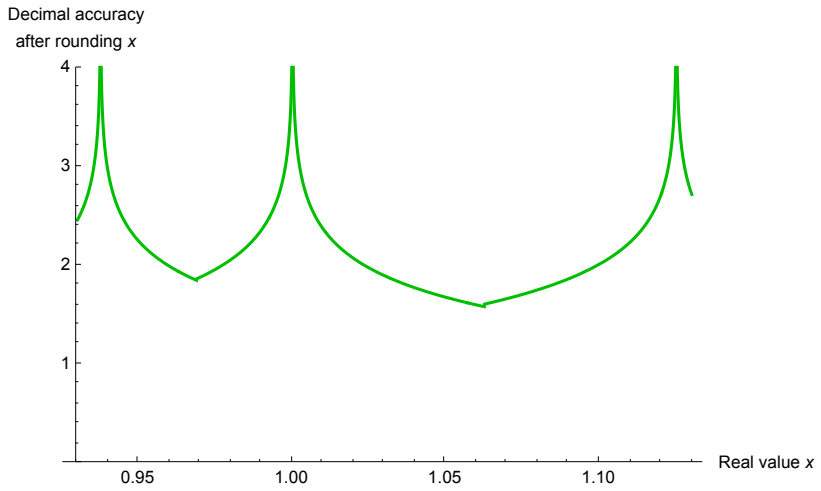
We can also analyze the accuracy of a **number system**. Let's just study three consecutive values in the **`float8`** set: {15/16, 1, 9/8}. We know that

Values in [15/16, 31/32) round to 15/16.

Values in [31/32,17/16] round to 1.

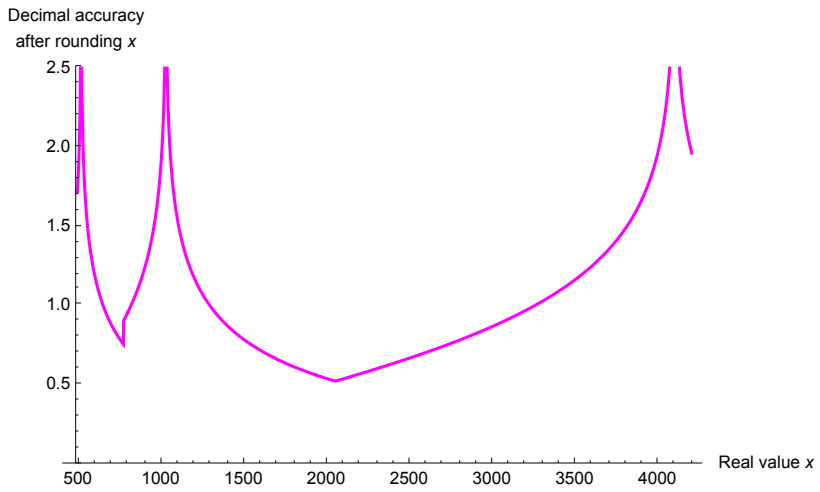
Values in (17/16,9/8] round to 9/8.

If we happen to have a value that is exactly 15/16, 1, or 9/8, then the decimal accuracy is infinite. In between those values, decimal accuracy dips to a minimum.

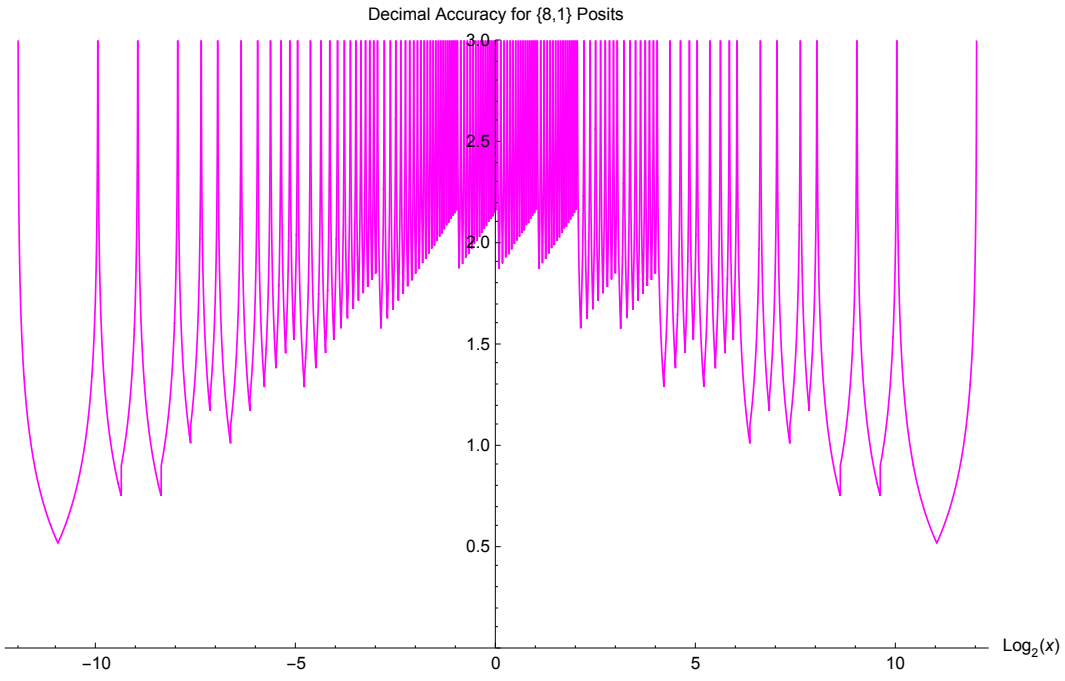


Notice the slight discontinuity at the two minima; that's because of the difference between the *geometric* mean and the *arithmetic* mean. The cusps are at the halfway points, $\frac{15/16+1}{2} = 0.96875$ and $\frac{1+9/8}{2} = 1.0625$, but those are *not* the places where the decimal error, defined as a logarithmic distance, is minimized. That point would happen at $\sqrt{(15/16)\times 1} = 0.96824\cdots$ and $\sqrt{1\times(9/8)} = 1.0606\cdots$, which at this extremely low precision is a large enough discontinuity to see in the graph, barely. In 16-bit precision or greater, you really wouldn't be able to see it. If it weren't so computationally expensive, it would be a tiny bit better if all rounding were done according to which side of the geometric mean a result falls, not the arithmetic mean.

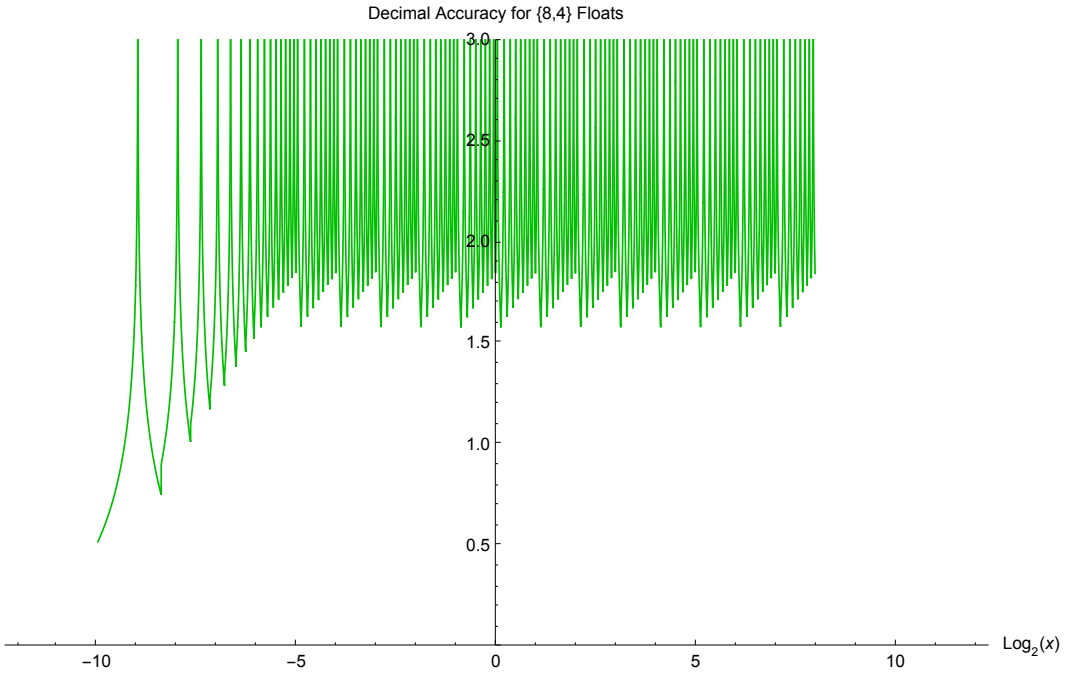
Notice, though, that posits *do* round using the geometric mean, **if the rounded bits are exponent bits**. That only happens at the extremes of the range, but it helps increase decimal accuracy. For instance, in the 8-bit posit set under examination here, the three largest positive reals are 512, 1024, and 4096. Here is what posit decimal accuracy looks like for *x* in that range:



The geometric mean of 512 and 1024 is 724.077⋯ but normal linear rounding of the fraction part uses the midpoint, 768, as the switchover point, creating a subtle but visible discontinuity. The geometric mean between 1024 and 4096, on the other hand, is exactly 2048 and that is also what posits use for the switchover point. Here is a decimal accuracy plot for the positive posits:

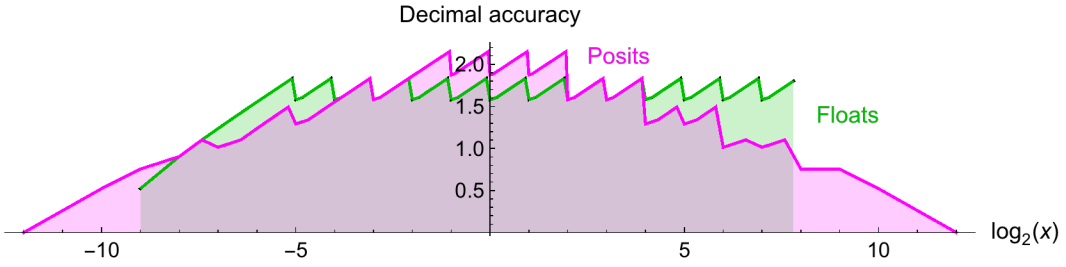


The tapered precision is very clear. Worst-case precision is highest where the most common numbers are, in the center of the range of possible exponents. Here is the equivalent plot for the 8-bit floats we are testing here:



The dynamic range is visibly smaller than for posits, and accuracy is tapered only on the left where the denormalized float exceptions happen.

Those graphs are thorough in showing the accuracy of the entire number system over the dynamic range, but the vertical spikes at the exact points make it difficult to compare the two systems. Let's instead be pessimistic and look at the *worst case decimal accuracy* for each point in the number set. The following graph shows just the bottom of each trough, allowing us to show both posits and floats on the same plot:



At higher precisions, the jaggedness is less pronounced, and the posit decimal accuracy is a symmetrical triangular outline that peaks in the center; the float pattern always forms a rectangle with a ramp on the left for the denormalized floats. The posits have higher accuracy than floats in the center, and less near the underflow and overflow regions. The larger the *es* value, the broader the triangle that describes posit accuracy, and at some point it becomes so broad that the entire accuracy curve has less accuracy than the floats.

For the 8-bit posits and floats, posits have superior accuracy for numbers with absolute value between 1/4 and 4, and equal or superior accuracy for numbers with absolute value between 1/16 and 16.

7.5 The Morris Floats

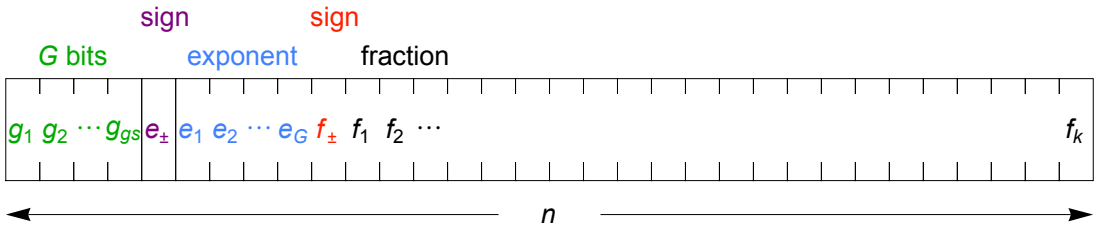
We now have enough mathematical machinery to show what happened to the 1971 tapered precision scheme. Almost half a century ago, Robert Morris proposed it in his paper, "Tapered Floating Point: A New Floating-Point Representation." It is worth studying his proposal and comparing it with posit arithmetic using the tools presented so far, particularly since those familiar with the literature on alternatives to floats remember that there were attempts to create tapered precision early on. We know that the Morris approach never caught on or influenced the IEEE 754 Standard committee. Perhaps we can figure out why it didn't.

Morris suggests an additional field, the *G* field, that describes how many bits are in the exponent. The bits in the *G* field represent an integer to which we add an offset, so two *G* bits could represent, say, an exponent of size 1, 2, 3, or 4. Or it could represent 3, 4, 5, or 6. He describes the need for

- The *G* field for the number of exponent bits
- The sign bit of the exponent
- The *E* field for the exponent bits
- The sign bit of the fraction
- The *F* field for the remaining bits, a fraction field where the hidden bit is always 1 (no subnormals).

The claim in the 1971 paper is that this allows more accuracy bits (fraction bits) for values with small exponents, yet more dynamic range than a system that only has a signed exponent and a signed fraction. That sounds familiar! The author notes that there can be multiple ways to represent a particular value with this system (redundant bit patterns) and he proposes the convention that the representation with the smallest value stored in *G* is the one to use; the other bit patterns become meaningless under the system.

A Morris float looks like the following, with five sub-fields:



Notice that we only show black dividing lines between the *G* bits and the exponent sign bit, and between the exponent sign bit and the exponent bits. That's to remind us that the other partition locations vary according to the contents of *G*.

We are using *nfbit* for the number of bits in a standard float, so let's use *nmbit* for the number in a Morris float. We can use *gs* as the size of the *G* field, and define a Morris float environment with **`setmorrisenv`**:

```
setmorrisenv[{n_Integer /; n ≥ 4, gsize_Integer /; gsize ≥ 1}] :=
 (
  {nmbit, gs} = {n, gsize};
  minmorris = 2^-(2^2^gs-1);
  maxmorris = 2^(2^2^gs-1) (1 + (2^(n-2^gs-gs-2) - 1)/2^(n-2^gs-gs-2));
 )
```

For example, set an environment for 8 bits per number and 1 *gs* bit:

```
setmorrisenv[{8, 1}]
{nmbit, gs, minmorris, maxmorris}
```

$$\left\{8,\ 1,\ \frac{1}{8},\ 15\right\}$$

There's an early warning about this scheme. The dynamic range is awfully small. In fact, the dynamic range would have been larger if we had used the 8 bits to represent unsigned integers from −128 to 127.

Now we need the equivalent of **f2x**, which we can call **m2x**, that follows the rules specified by Morris.

```
m2x[m_Integer /; 0 ≤ m < 2^nmbit] :=
 Module[{emask, esgn, exp, fmask, frac, fsgn, fs, g},
  g = BitShiftRight[m, nmbit - gs];
  esgn = BitAnd[1, BitShiftRight[m, nmbit - gs - 1]];
  emask = FromDigits[Table[1, g + 1], 2];
  fs = nmbit - gs - g - 3;
  fsgn = BitAnd[1, BitShiftRight[m, fs]];
  exp = BitAnd[BitShiftRight[m, fs + 1], emask];
  frac = BitAnd[FromDigits[Table[1, fs], 2], m];
  If[m == 0, 0, (-1)^fsgn × 2^((-1)^esgn×exp) × (1 + frac / 2^fs)]]
```

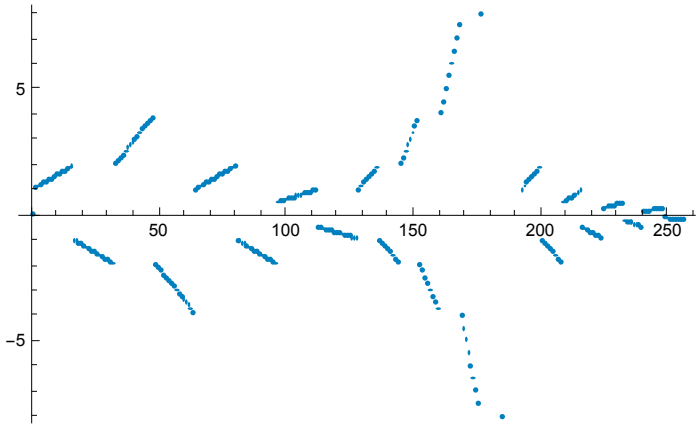
Apologies to the reader for what follows, but it is important to see the kind of chaos that results from simply adding a field that describes the exponent size:

```
morris8 = Table[m2x[i], {i, 0, 2^nmbit - 1}]
```

$$\left\{0, \frac{17}{16}, \frac{9}{8}, \frac{19}{16}, \frac{5}{4}, \frac{21}{16}, \frac{11}{8}, \frac{23}{16}, \frac{3}{2}, \frac{25}{16}, \frac{13}{8}, \frac{27}{16}, \frac{7}{4}, \frac{29}{16}, \frac{15}{8}, \frac{31}{16}, -1, -\frac{17}{16},\right.$$
$$-\frac{9}{8}, -\frac{19}{16}, -\frac{5}{4}, -\frac{21}{16}, -\frac{11}{8}, -\frac{23}{16}, -\frac{3}{2}, -\frac{25}{16}, -\frac{13}{8}, -\frac{27}{16}, -\frac{7}{4}, -\frac{29}{16}, -\frac{15}{8},$$
$$-\frac{31}{16}, 2, \frac{17}{8}, \frac{9}{4}, \frac{19}{8}, \frac{5}{2}, \frac{21}{8}, \frac{11}{4}, \frac{23}{8}, 3, \frac{25}{8}, \frac{13}{4}, \frac{27}{8}, \frac{7}{2}, \frac{29}{8}, \frac{15}{4}, \frac{31}{8}, -2,$$
$$-\frac{17}{8}, -\frac{9}{4}, -\frac{19}{8}, -\frac{5}{2}, -\frac{21}{8}, -\frac{11}{4}, -\frac{23}{8}, -3, -\frac{25}{8}, -\frac{13}{4}, -\frac{27}{8}, -\frac{7}{2}, -\frac{29}{8}, -\frac{15}{4},$$
$$-\frac{31}{8}, 1, \frac{17}{16}, \frac{9}{8}, \frac{19}{16}, \frac{5}{4}, \frac{21}{16}, \frac{11}{8}, \frac{23}{16}, \frac{3}{2}, \frac{25}{16}, \frac{13}{8}, \frac{27}{16}, \frac{7}{4}, \frac{29}{16}, \frac{15}{8}, \frac{31}{16},$$
$$-1, -\frac{17}{16}, -\frac{9}{8}, -\frac{19}{16}, -\frac{5}{4}, -\frac{21}{16}, -\frac{11}{8}, -\frac{23}{16}, -\frac{3}{2}, -\frac{25}{16}, -\frac{13}{8}, -\frac{27}{16}, -\frac{7}{4}, -\frac{29}{16},$$
$$-\frac{15}{8}, -\frac{31}{16}, \frac{1}{2}, \frac{17}{32}, \frac{9}{16}, \frac{19}{32}, \frac{5}{8}, \frac{21}{32}, \frac{11}{16}, \frac{23}{32}, \frac{3}{4}, \frac{25}{32}, \frac{13}{16}, \frac{27}{32}, \frac{7}{8}, \frac{29}{32}, \frac{15}{16},$$
$$\frac{31}{32}, -\frac{1}{2}, -\frac{17}{32}, -\frac{9}{16}, -\frac{19}{32}, -\frac{5}{8}, -\frac{21}{32}, -\frac{11}{16}, -\frac{23}{32}, -\frac{3}{4}, -\frac{25}{32}, -\frac{13}{16}, -\frac{27}{32}, -\frac{7}{8},$$
$$-\frac{29}{32}, -\frac{15}{16}, -\frac{31}{32}, 1, \frac{9}{8}, \frac{5}{4}, \frac{11}{8}, \frac{3}{2}, \frac{13}{8}, \frac{7}{4}, \frac{15}{8}, -1, -\frac{9}{8}, -\frac{5}{4}, -\frac{11}{8}, -\frac{3}{2}, -\frac{13}{8},$$
$$-\frac{7}{4}, -\frac{15}{8}, 2, \frac{9}{4}, \frac{5}{2}, \frac{11}{4}, 3, \frac{13}{4}, \frac{7}{2}, \frac{15}{4}, -2, -\frac{9}{4}, -\frac{5}{2}, -\frac{11}{4}, -3, -\frac{13}{4}, -\frac{7}{2},$$
$$-\frac{15}{4}, 4, \frac{9}{2}, 5, \frac{11}{2}, 6, \frac{13}{2}, 7, \frac{15}{2}, -4, -\frac{9}{2}, -5, -\frac{11}{2}, -6, -\frac{13}{2}, -7, -\frac{15}{2}, 8,$$
$$9, 10, 11, 12, 13, 14, 15, -8, -9, -10, -11, -12, -13, -14, -15, 1, \frac{9}{8}, \frac{5}{4},$$
$$\frac{11}{8}, \frac{3}{2}, \frac{13}{8}, \frac{7}{4}, \frac{15}{8}, -1, -\frac{9}{8}, -\frac{5}{4}, -\frac{11}{8}, -\frac{3}{2}, -\frac{13}{8}, -\frac{7}{4}, -\frac{15}{8}, \frac{1}{2}, \frac{9}{16}, \frac{5}{8},$$
$$\frac{11}{16}, \frac{3}{4}, \frac{13}{16}, \frac{7}{8}, \frac{15}{16}, -\frac{1}{2}, -\frac{9}{16}, -\frac{5}{8}, -\frac{11}{16}, -\frac{3}{4}, -\frac{13}{16}, -\frac{7}{8}, -\frac{15}{16}, \frac{1}{4}, \frac{9}{32}, \frac{5}{16},$$
$$\frac{11}{32}, \frac{3}{8}, \frac{13}{32}, \frac{7}{16}, \frac{15}{32}, -\frac{1}{4}, -\frac{9}{32}, -\frac{5}{16}, -\frac{11}{32}, -\frac{3}{8}, -\frac{13}{32}, -\frac{7}{16}, -\frac{15}{32}, \frac{1}{8}, \frac{9}{64},$$
$$\left.\frac{5}{32}, \frac{11}{64}, \frac{3}{16}, \frac{13}{64}, \frac{7}{32}, \frac{15}{64}, -\frac{1}{8}, -\frac{9}{64}, -\frac{5}{32}, -\frac{11}{64}, -\frac{3}{16}, -\frac{13}{64}, -\frac{7}{32}, -\frac{15}{64}\right\}$$

A discrete plot of the values represented by ordered bit strings gives us the second clue about the shortcomings of the Morris proposal:

```
ListPlot[morris8, PlotStyle → ■]
```



What a crazy ordering this format produces! But here is the crucial failure of the Morris approach to tapered precision. Sort the values and get rid of the many redundant representations:

```
morris8s = Union[morris8]
```

$$\left\{-15, -14, -13, -12, -11, -10, -9, -8, -\frac{15}{2}, -7, -\frac{13}{2}, -6, -\frac{11}{2}, -5, -\frac{9}{2}, -4,\right.$$
$$-\frac{31}{8}, -\frac{15}{4}, -\frac{29}{8}, -\frac{7}{2}, -\frac{27}{8}, -\frac{13}{4}, -\frac{25}{8}, -3, -\frac{23}{8}, -\frac{11}{4}, -\frac{21}{8}, -\frac{5}{2}, -\frac{19}{8}, -\frac{9}{4},$$
$$-\frac{17}{8}, -2, -\frac{31}{16}, -\frac{15}{8}, -\frac{29}{16}, -\frac{7}{4}, -\frac{27}{16}, -\frac{13}{8}, -\frac{25}{16}, -\frac{3}{2}, -\frac{23}{16}, -\frac{11}{8}, -\frac{21}{16}, -\frac{5}{4},$$
$$-\frac{19}{16}, -\frac{9}{8}, -\frac{17}{16}, -1, -\frac{31}{32}, -\frac{15}{16}, -\frac{29}{32}, -\frac{7}{8}, -\frac{27}{32}, -\frac{13}{16}, -\frac{25}{32}, -\frac{3}{4}, -\frac{23}{32}, -\frac{11}{16},$$
$$-\frac{21}{32}, -\frac{5}{8}, -\frac{19}{32}, -\frac{9}{16}, -\frac{17}{32}, -\frac{1}{2}, -\frac{15}{32}, -\frac{7}{16}, -\frac{13}{32}, -\frac{3}{8}, -\frac{11}{32}, -\frac{5}{16}, -\frac{9}{32}, -\frac{1}{4},$$
$$-\frac{15}{64}, -\frac{7}{32}, -\frac{13}{64}, -\frac{3}{16}, -\frac{11}{64}, -\frac{5}{32}, -\frac{9}{64}, -\frac{1}{8}, 0, \frac{1}{8}, \frac{9}{64}, \frac{5}{32}, \frac{11}{64}, \frac{3}{16}, \frac{13}{64},$$
$$\frac{7}{32}, \frac{15}{64}, \frac{1}{4}, \frac{9}{32}, \frac{5}{16}, \frac{11}{32}, \frac{3}{8}, \frac{13}{32}, \frac{7}{16}, \frac{15}{32}, \frac{1}{2}, \frac{17}{32}, \frac{9}{16}, \frac{19}{32}, \frac{5}{8}, \frac{21}{32}, \frac{11}{16}, \frac{23}{32},$$
$$\frac{3}{4}, \frac{25}{32}, \frac{13}{16}, \frac{27}{32}, \frac{7}{8}, \frac{29}{32}, \frac{15}{16}, \frac{31}{32}, 1, \frac{17}{16}, \frac{9}{8}, \frac{19}{16}, \frac{5}{4}, \frac{21}{16}, \frac{11}{8}, \frac{23}{16}, \frac{3}{2}, \frac{25}{16},$$
$$\frac{13}{8}, \frac{27}{16}, \frac{7}{4}, \frac{29}{16}, \frac{15}{8}, \frac{31}{16}, 2, \frac{17}{8}, \frac{9}{4}, \frac{19}{8}, \frac{5}{2}, \frac{21}{8}, \frac{11}{4}, \frac{23}{8}, 3, \frac{25}{8}, \frac{13}{4}, \frac{27}{8},$$
$$\left.\frac{7}{2}, \frac{29}{8}, \frac{15}{4}, \frac{31}{8}, 4, \frac{9}{2}, 5, \frac{11}{2}, 6, \frac{13}{2}, 7, \frac{15}{2}, 8, 9, 10, 11, 12, 13, 14, 15\right\}$$

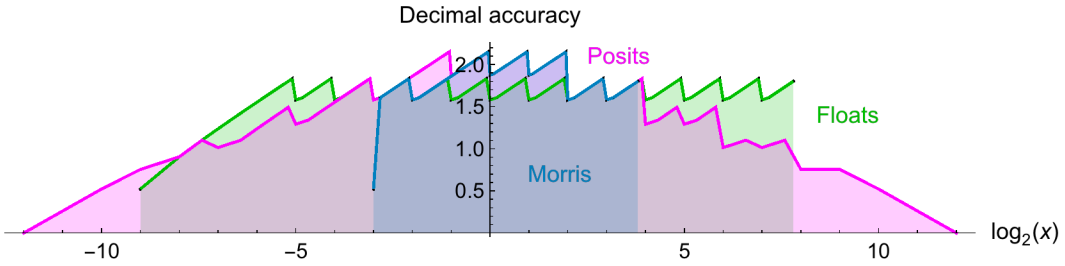
Eight bits can produce $2^8 = 256$ distinct bit patterns. How many mathematical values are in the above list?

```
Length[morris8s]
```

```
161
```

There should be 256 distinct values… but we only have 161. The Morris system is littered with redundant ways to express the same value, wasting over *one-third* of the bit patterns.

With that as the number system, we can plot the decimal accuracy alongside that for posits and conventional floats:



At their best, Morris floats *match* posit accuracy, but have a much smaller dynamic range. Let's at least show that by allowing two bits in *G* instead of one, Morris floats *can* have a larger dynamic range (at the expense of very low accuracy everywhere):

```
setmorrisenv[{8, 2}]
{nmbit, gs, minmorris, maxmorris}
```

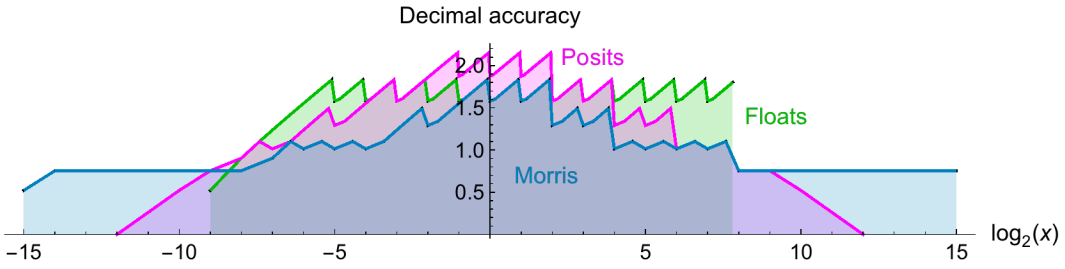
$$\left\{8,\ 2,\ \frac{1}{32\,768},\ 32\,768\right\}$$

Here is the set that results, after removing all the redundancies:

```
morris8s = Union[Table[m2x[i], {i, 0, 2^nmbit - 1}]]
```

$$\left\{-32\,768, -16\,384, -8192, -4096, -2048, -1024, -512, -256, -192, -128, -96, -64, -48, -32, -24, -16, -14, -12, -10, -8, -7, -6, -5, -4, -\frac{15}{4}, -\frac{7}{2}, -\frac{13}{4}, -3, -\frac{11}{4}, -\frac{5}{2}, -\frac{9}{4}, -2, -\frac{15}{8}, -\frac{7}{4}, -\frac{13}{8}, -\frac{3}{2}, -\frac{11}{8}, -\frac{5}{4}, -\frac{9}{8}, -1, -\frac{15}{16}, -\frac{7}{8}, -\frac{13}{16}, -\frac{3}{4}, -\frac{11}{16}, -\frac{5}{8}, -\frac{9}{16}, -\frac{1}{2}, -\frac{7}{16}, -\frac{3}{8}, -\frac{5}{16}, -\frac{1}{4}, -\frac{7}{32}, -\frac{3}{16}, -\frac{5}{32}, -\frac{1}{8}, -\frac{3}{32}, -\frac{1}{16}, -\frac{3}{64}, -\frac{1}{32}, -\frac{3}{128}, -\frac{1}{64}, -\frac{3}{256}, -\frac{1}{128}, -\frac{1}{256}, -\frac{1}{512}, -\frac{1}{1024}, -\frac{1}{2048}, -\frac{1}{4096}, -\frac{1}{8192}, -\frac{1}{16\,384}, -\frac{1}{32\,768}, 0, \frac{1}{32\,768}, \frac{1}{16\,384}, \frac{1}{8192}, \frac{1}{4096}, \frac{1}{2048}, \frac{1}{1024}, \frac{1}{512}, \frac{1}{256}, \frac{1}{128}, \frac{3}{256}, \frac{1}{64}, \frac{3}{128}, \frac{1}{32}, \frac{3}{64}, \frac{1}{16}, \frac{3}{32}, \frac{1}{8}, \frac{5}{32}, \frac{3}{16}, \frac{7}{32}, \frac{1}{4}, \frac{5}{16}, \frac{3}{8}, \frac{7}{16}, \frac{1}{2}, \frac{9}{16}, \frac{5}{8}, \frac{11}{16}, \frac{3}{4}, \frac{13}{16}, \frac{7}{8}, \frac{15}{16}, 1, \frac{9}{8}, \frac{5}{4}, \frac{11}{8}, \frac{3}{2}, \frac{13}{8}, \frac{7}{4}, \frac{15}{8}, 2, \frac{9}{4}, \frac{5}{2}, \frac{11}{4}, 3, \frac{13}{4}, \frac{7}{2}, \frac{15}{4}, 4, 5, 6, 7, 8, 10, 12, 14, 16, 24, 32, 48, 64, 96, 128, 192, 256, 512, 1024, 2048, 4096, 8192, 16\,384, 32\,768\right\}$$

Now there is even more waste of bit patterns, since there are only *145* distinct values.

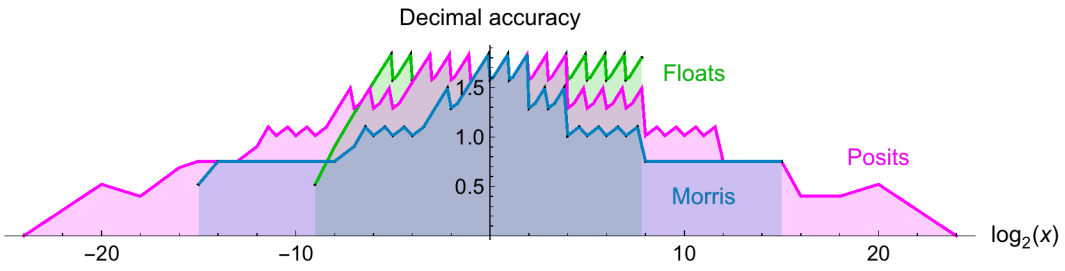


At least the dynamic range now goes from 2^{-15} to 2^{15}. But the accuracy is *worse* than floats for the full range where floats are defined, except for a small interval for magnitudes between 1/2 and 4, where it manages to eke out the same accuracy. Hmm… what would happen if we also gave *posits* another bit for dynamic range?

```
setpositenv[{8, 2}]
posit8 = Table[p2x[positfix[j]], {j, 0, npat - 1}]
```

$$\{\text{ComplexInfinity}, -16\,777\,216, -1\,048\,576, -262\,144, -65\,536, -32\,768, -16\,384,$$
$$-8192, -4096, -3072, -2048, -1536, -1024, -768, -512, -384, -256, -224,$$
$$-192, -160, -128, -112, -96, -80, -64, -56, -48, -40, -32, -28, -24, -20,$$
$$-16, -15, -14, -13, -12, -11, -10, -9, -8, -\frac{15}{2}, -7, -\frac{13}{2}, -6, -\frac{11}{2}, -5, -\frac{9}{2},$$
$$-4, -\frac{15}{4}, -\frac{7}{2}, -\frac{13}{4}, -3, -\frac{11}{4}, -\frac{5}{2}, -\frac{9}{4}, -2, -\frac{15}{8}, -\frac{7}{4}, -\frac{13}{8}, -\frac{3}{2}, -\frac{11}{8}, -\frac{5}{4}, -\frac{9}{8},$$
$$-1, -\frac{15}{16}, -\frac{7}{8}, -\frac{13}{16}, -\frac{3}{4}, -\frac{11}{16}, -\frac{5}{8}, -\frac{9}{16}, -\frac{1}{2}, -\frac{15}{32}, -\frac{7}{16}, -\frac{13}{32}, -\frac{3}{8}, -\frac{11}{32}, -\frac{5}{16},$$
$$-\frac{9}{32}, -\frac{1}{4}, -\frac{15}{64}, -\frac{7}{32}, -\frac{13}{64}, -\frac{3}{16}, -\frac{11}{64}, -\frac{5}{32}, -\frac{9}{64}, -\frac{1}{8}, -\frac{15}{128}, -\frac{7}{64}, -\frac{13}{128},$$
$$-\frac{3}{32}, -\frac{11}{128}, -\frac{5}{64}, -\frac{9}{128}, -\frac{1}{16}, -\frac{7}{128}, -\frac{3}{64}, -\frac{5}{128}, -\frac{1}{32}, -\frac{7}{256}, -\frac{3}{128}, -\frac{5}{256},$$
$$-\frac{1}{64}, -\frac{7}{512}, -\frac{3}{256}, -\frac{5}{512}, -\frac{1}{128}, -\frac{7}{1024}, -\frac{3}{512}, -\frac{5}{1024}, -\frac{1}{256}, -\frac{3}{1024}, -\frac{1}{512},$$
$$-\frac{3}{2048}, -\frac{1}{1024}, -\frac{3}{4096}, -\frac{1}{2048}, -\frac{3}{8192}, -\frac{1}{4096}, -\frac{1}{8192}, -\frac{1}{16\,384}, -\frac{1}{32\,768},$$
$$-\frac{1}{65\,536}, -\frac{1}{262\,144}, -\frac{1}{1\,048\,576}, -\frac{1}{16\,777\,216}, 0, \frac{1}{16\,777\,216}, \frac{1}{1\,048\,576}, \frac{1}{262\,144},$$
$$\frac{1}{65\,536}, \frac{1}{32\,768}, \frac{1}{16\,384}, \frac{1}{8192}, \frac{1}{4096}, \frac{3}{8192}, \frac{1}{2048}, \frac{3}{4096}, \frac{1}{1024}, \frac{3}{2048}, \frac{1}{512},$$
$$\frac{3}{1024}, \frac{1}{256}, \frac{5}{1024}, \frac{3}{512}, \frac{7}{1024}, \frac{1}{128}, \frac{5}{512}, \frac{3}{256}, \frac{7}{512}, \frac{1}{64}, \frac{5}{256}, \frac{3}{128}, \frac{7}{256},$$
$$\frac{1}{32}, \frac{5}{128}, \frac{3}{64}, \frac{7}{128}, \frac{1}{16}, \frac{9}{128}, \frac{5}{64}, \frac{11}{128}, \frac{3}{32}, \frac{13}{128}, \frac{7}{64}, \frac{15}{128}, \frac{1}{8}, \frac{9}{64}, \frac{5}{32},$$
$$\frac{11}{64}, \frac{3}{16}, \frac{13}{64}, \frac{7}{32}, \frac{15}{64}, \frac{1}{4}, \frac{9}{32}, \frac{5}{16}, \frac{11}{32}, \frac{3}{8}, \frac{13}{32}, \frac{7}{16}, \frac{15}{32}, \frac{1}{2}, \frac{9}{16}, \frac{5}{8}, \frac{11}{16}, \frac{3}{4},$$
$$\frac{13}{16}, \frac{7}{8}, \frac{15}{16}, 1, \frac{9}{8}, \frac{5}{4}, \frac{11}{8}, \frac{3}{2}, \frac{13}{8}, \frac{7}{4}, \frac{15}{8}, 2, \frac{9}{4}, \frac{5}{2}, \frac{11}{4}, 3, \frac{13}{4}, \frac{7}{2}, \frac{15}{4}, 4,$$
$$\frac{9}{2}, 5, \frac{11}{2}, 6, \frac{13}{2}, 7, \frac{15}{2}, 8, 9, 10, 11, 12, 13, 14, 15, 16, 20, 24, 28, 32, 40,$$
$$48, 56, 64, 80, 96, 112, 128, 160, 192, 224, 256, 384, 512, 768, 1024, 1536,$$
$$2048, 3072, 4096, 8192, 16\,384, 32\,768, 65\,536, 262\,144, 1\,048\,576, 16\,777\,216\}$$

Again, here are all three number systems on the same graph.



That's a massive dynamic range from a byte-sized number! Posits match or outperform Morris floats everywhere. Posits also have the same accuracy as IEEE floats except for the regions close to where they overflow or underflow.

This shows the value of designing a number system to make the best possible use of **every bit pattern**. Trying proposed systems out with low precision is a quick way to expose shortcomings that may be masked when using 32-bit or 64-bit representation.

That was fun, but for the upcoming sections, we need to go back to an {8, 1} posit environment for comparison with the 8-bit floats:

```
setpositenv[{8, 1}]
posit8 = Table[p2x[positfix[j]], {j, 0, npat - 1}];
```

7.6 Comparing floats with posits performing *unary* operations

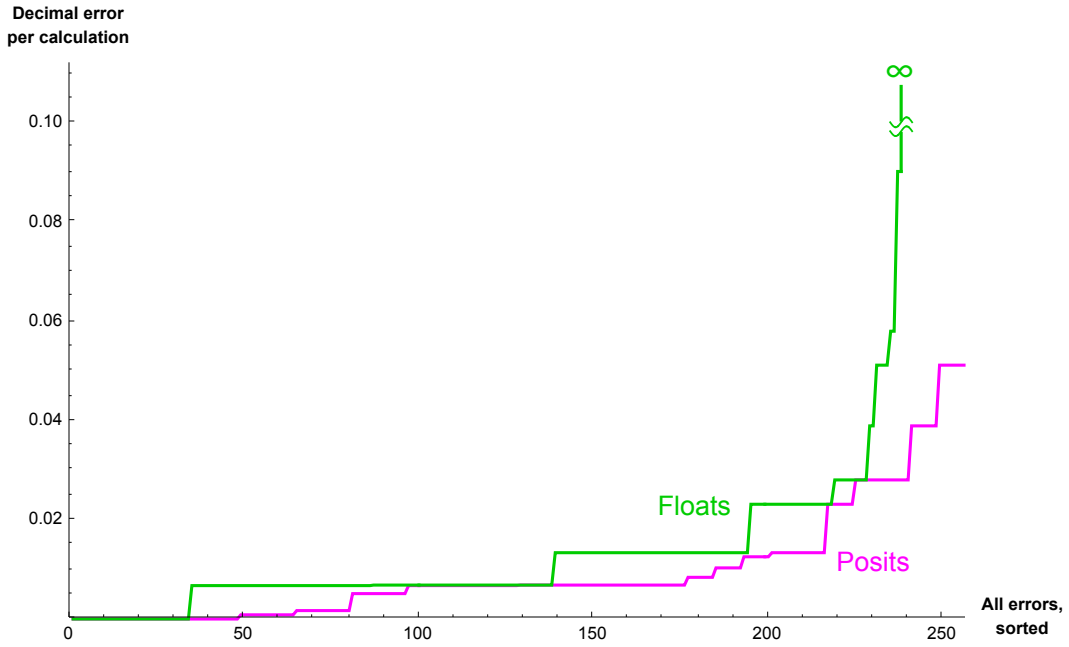
7.6.1 Reciprocation

We can compare the *reciprocal closure*, that is, the percentage of cases where $1/x$ is exactly representable as a member of the set. Compare the percentage of the entire set of floats or posits for which a reciprocal is exact, finite but inexact, produces a NaN, overflows, or underflows:

Only 34 of the float values have exact reciprocals. In contrast, 48 of the 256 unum values have exact reciprocals, and **never** experience catastrophic loss of accuracy through overflow. The IEEE float definition is a "kludge" in that it has subnormal numbers at the low end (the reciprocals of which incorrectly overflow to infinity), but replaces the high end numbers with NaN values.

The following graph makes it much easier to visualize the relative performance of floats and posits. The entire set of decimal losses in computing $1/x$ is sorted from smallest to largest, and

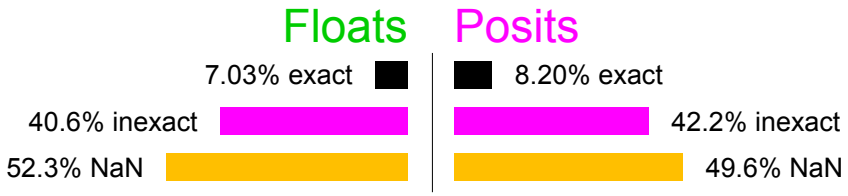
plotted.



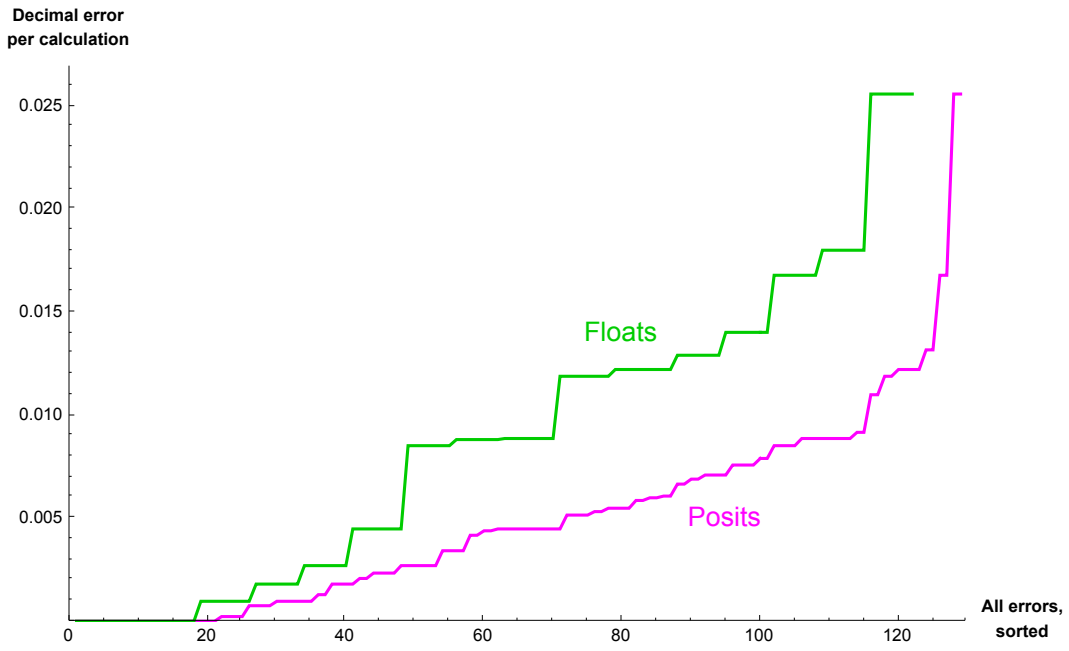
The graph of posit decimal losses grows more slowly than that for floats, and never goes to infinity.

7.6.2 Square Roots

We can also compare *square root closure*, that is, the percentage of cases where $\sqrt{x}$ is exactly representable as a member of the set. The **`sqrtbarchart`** routine also finds out the fraction of the time a square root is exact, finite but inexact, or produces a NaN. The square root operation cannot overflow or underflow. Negative inputs produce NaN results, but since the posit "±∞" really means *unsigned infinity*, the square root of ±∞ is ±∞ and thus is closed for that input.



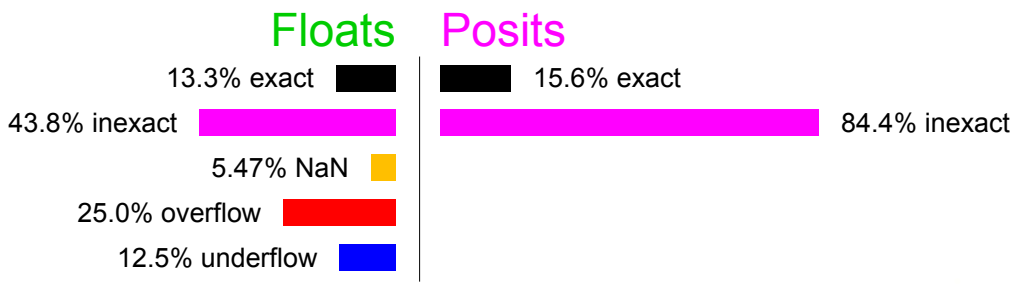
Posits do better, but at first glance it looks like the advantage is slight. The bar chart does not reveal just how much *more* inexact the floats are. The difference in the sorted losses is more dramatic than it was for computing $1/x$. Here are the sorted losses for every $\sqrt{x}$ value that is not a NaN:



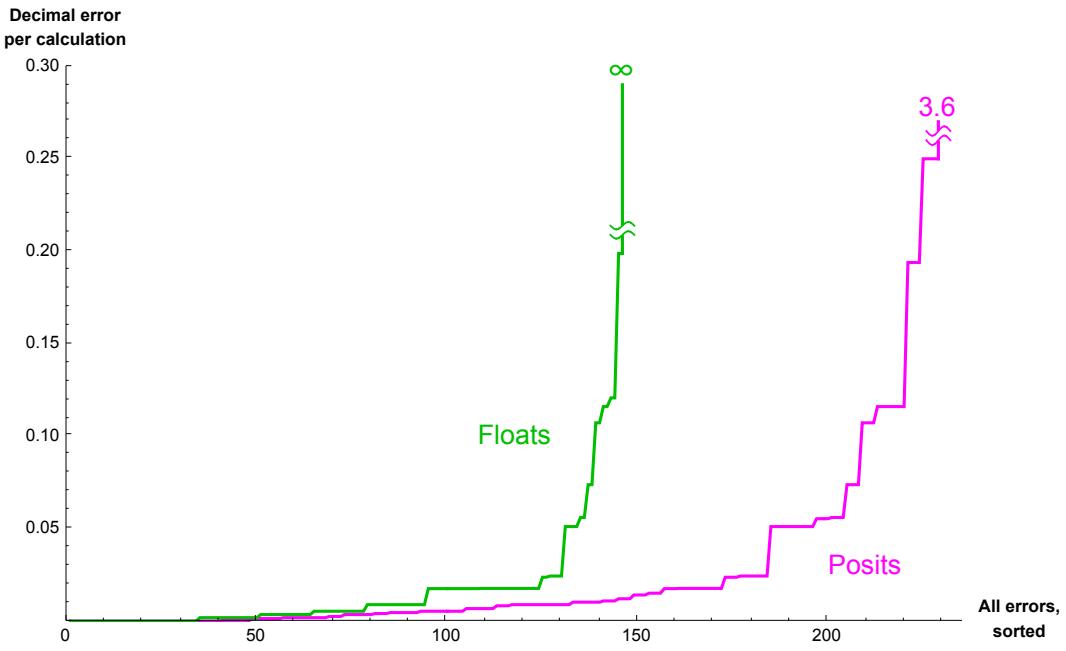
The posit errors are about half those of the floats (for the results that are not indeterminate).

7.6.3 Square

Another common unary operation is x^2. Overflow and underflow are a common disaster when squaring floats. Posits experience their largest decimal loss for squares that would overflow or underflow by IEEE rounding rules, but at least the loss is a few decimals and not *infinite*.



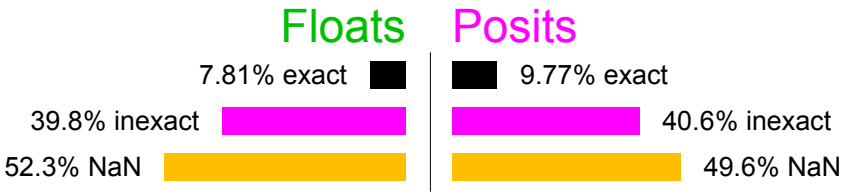
Posits do much better, mainly by not having any exception cases at all.



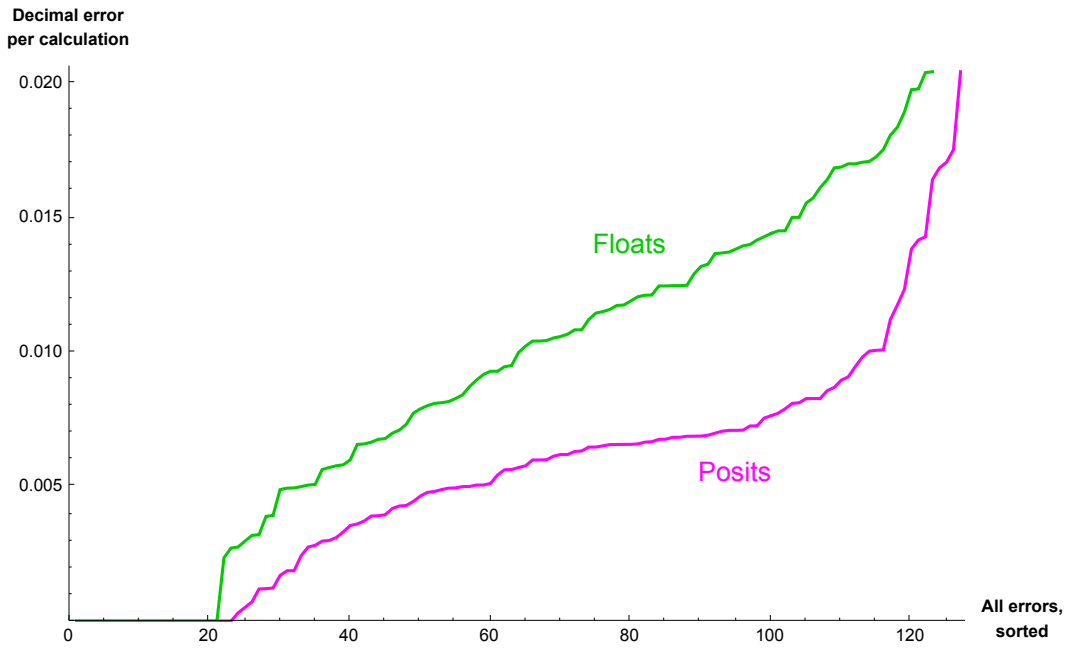
Every posit can be squared. (The square of unsigned infinity is again unsigned infinity.) In contrast, almost half the squarings of floats result in *complete loss of information* about the result.

7.6.4 Logarithm (base 2)

We can also compare the *logarithm base 2 closure*, that is, the percentage of cases where $\log_2(x)$ is exactly representable as a member of the set. As with square roots, about half the values produce a NaN since the logarithm of a negative value is a complex number. Note: we allow posits to return ±∞ as the logarithm of zero and the logarithm of ±∞. Remember, posit infinity is *unsigned*, like zero.



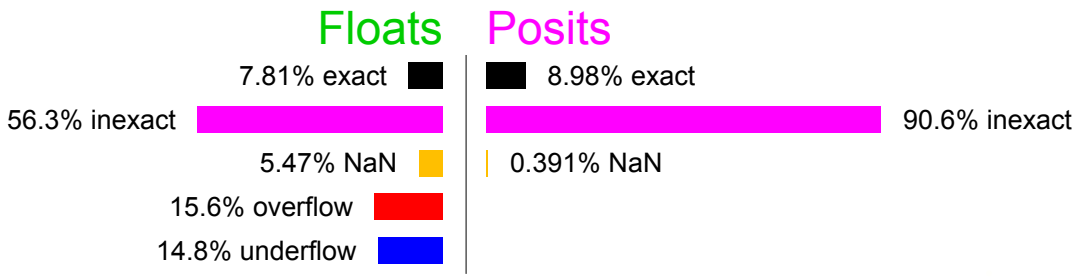
Posits do better, and again at first glance it looks like the advantage is slight. There are more integer powers of 2 in the posit environment, for which the logarithm base 2 is expressible exactly. Here are the sorted losses for every value that is not a NaN or infinity:



The posit errors are again about half those of the floats.

7.6.5 Exponential (base 2)

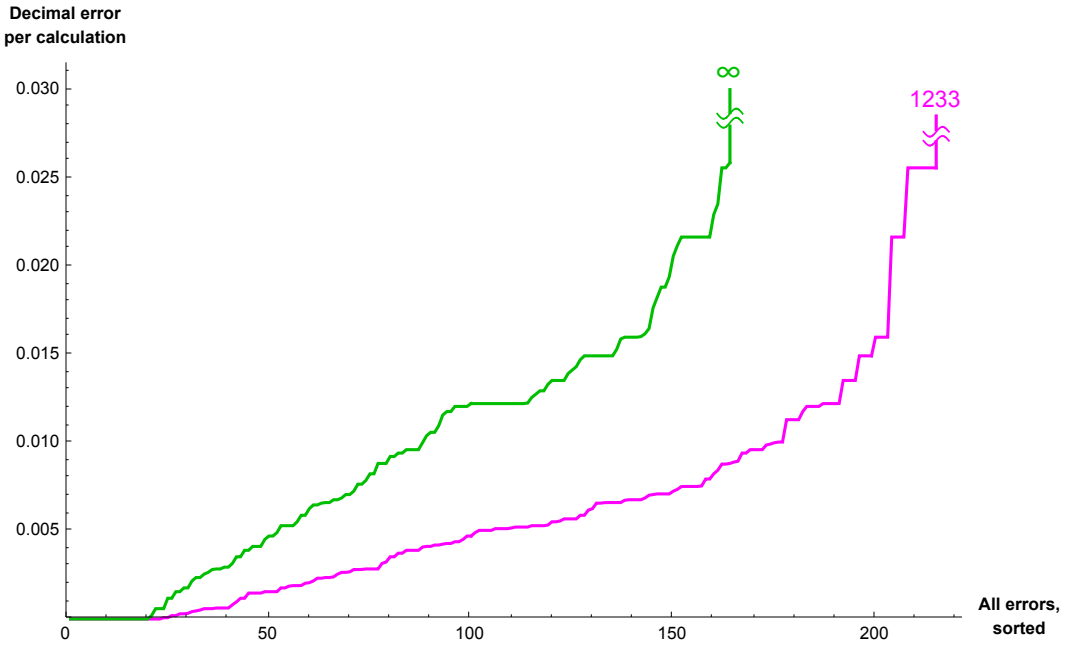
Maybe one more: 2^x. If you can do $\log_2(x)$, it just takes a scale factor to get to $\ln(x)$ or $\log_{10}(x)$ or any other logarithm base. Similarly, once you can do 2^x, it is easy to derive a scale factor that also gets you e^x or 10^x and so on.



Posits have one exception case: the result is NaN when the argument is ±∞. We can only use ±∞ as a legitimate answer if the result consists entirely of infinite values, but $2^{-\infty}$ is zero, so $2^{\pm\infty}$ is indeterminate.

The maximum decimal loss for posits is very large, because 2^{maxpos} will be rounded back to *maxpos*. For this example set, just a few errors are as high as $\log_{10}(2^{4096}) \approx 1233$ decimals. So, which is worse: the loss of over a thousand decimals, or the loss of an *infinite* number of decimals? Well, if you can stay away from those (rare) very largest values, it's still a win, because the error for smaller values is much better behaved for posits. Think of it this way: the only time you get a large decimal loss with the posits is when working with numbers far outside of what floats

can even *express* as input arguments.



For common unary operations $1/x$, $\sqrt{x}$, x^2, $\log_2(x)$, and 2^x, posits are consistently and uniformly more accurate than floats with the same number of bits, and produce meaningful results over a larger dynamic range. The advantage of posits becomes greater with larger precision. If we were to show the graphs for the unary operations comparing 32-bit floats with 32-bit posits, they would be hard to read because the posit errors would hug the *x*-axis of the plot when graphed at a scale large enough to show the float errors!

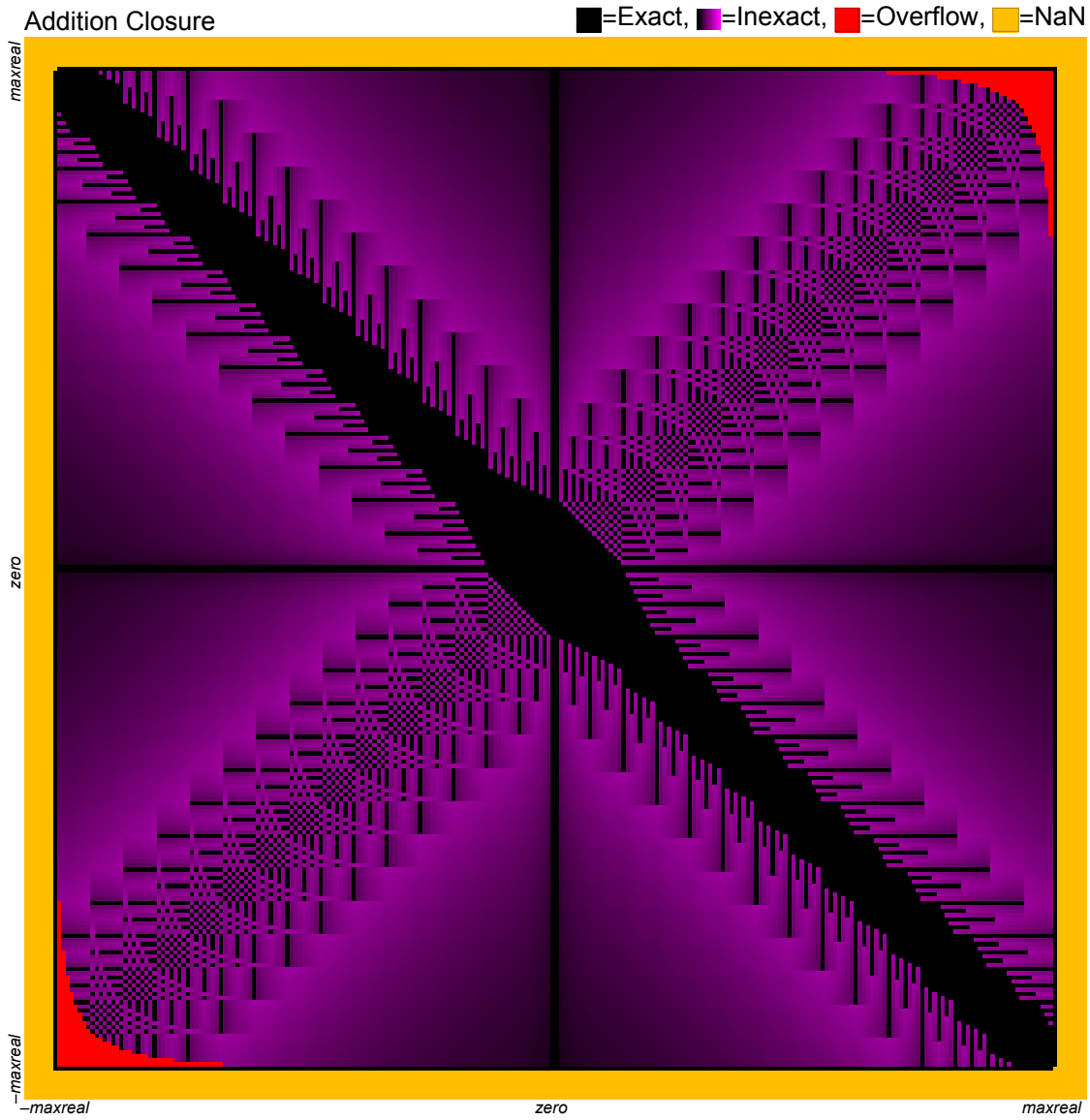
We now turn our attention to the four elementary arithmetic operations that take two arguments: Addition, subtraction, multiplication, and division.

7.7 Two-argument operations

Now we start to examine the four conventional arithmetic operations + – × ÷ that take two operands. To help visualize 65 536 results, we can make 256 by 256 "closure plots" that show at a glance what fraction of the results are exact, inexact, overflow, underflow, or NaN.

7.7.1 Comparing addition (or subtraction)

Because $x - y = x + (-y)$ works perfectly in both floats and posits, there is no need to study subtraction separately. For the addition operation, we compute $z = x + y$ exactly, and compare it with the sum that is returned by the rules of each number system. It can happen that the result is exact, that it must be rounded to a nearby finite nonzero number, that it can overflow or underflow, or can be an indeterminate form like $\infty - \infty$ that produces a NaN. Each of these is color-coded so we can look at the entire addition table at a glance. In the case of rounded results, the color-coding is a gradient from black (exact) to magenta (maximum error of either posits or floats). Here's what the closure plots look like for the floats and the unums. First, the floats:

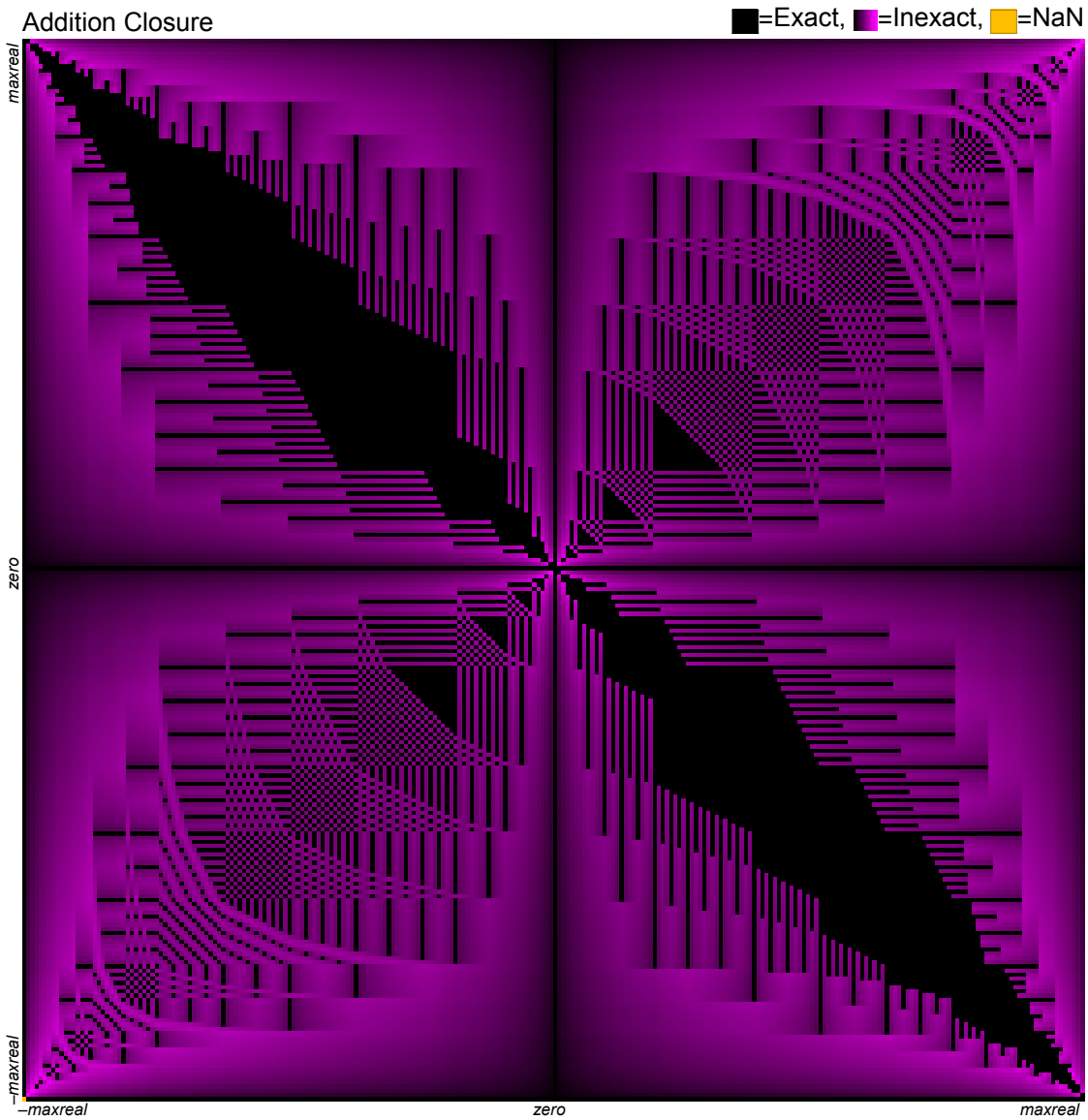


Addition Closure	
18.533%	exact
70.068%	inexact
10.641%	NaN
0.757%	overflow

Because of all the NaN representations in the inputs, the float table will always be framed with the NaN results, shown in yellow. If you look carefully in the upper left and lower right corner, you see additional NaN results for adding ∞ to –∞, and –∞ to ∞.

Sums of large positive or large negative values overflow to ∞ or –∞, but underflow does not occur for the addition operator.The error fades from magenta to black where the inputs are very different magnitude, because the loss of decimal accuracy is very small in such cases.

Here is the closure plot for posits, and its summary table:



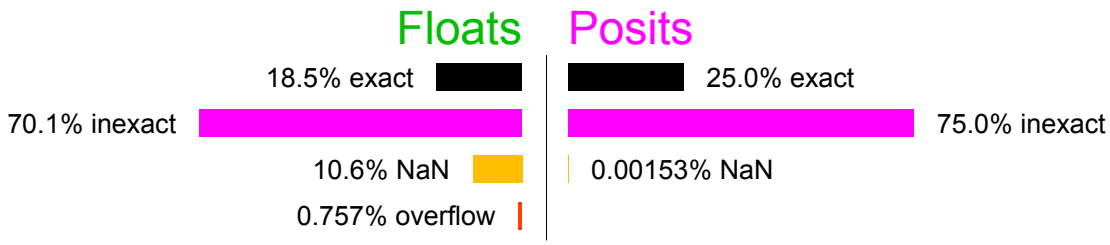
Addition Closure	
25.005%	exact
74.994%	inexact
0.002%	NaN
0.000%	overflow

The first thing you notice is that the black regions where sums are exact are much broader than they were for floats. Those are the regions where posits of similar magnitude but opposite sign are being added, resulting in cancellation of digits. When the magnitudes are near the center of the dynamic range, scraping off digits results in a smaller magnitude number that requires fewer fraction bits to represent exactly. Tapered precision is just what you want for those situations, and those situations happen more often with posits than floats because posits have more numbers

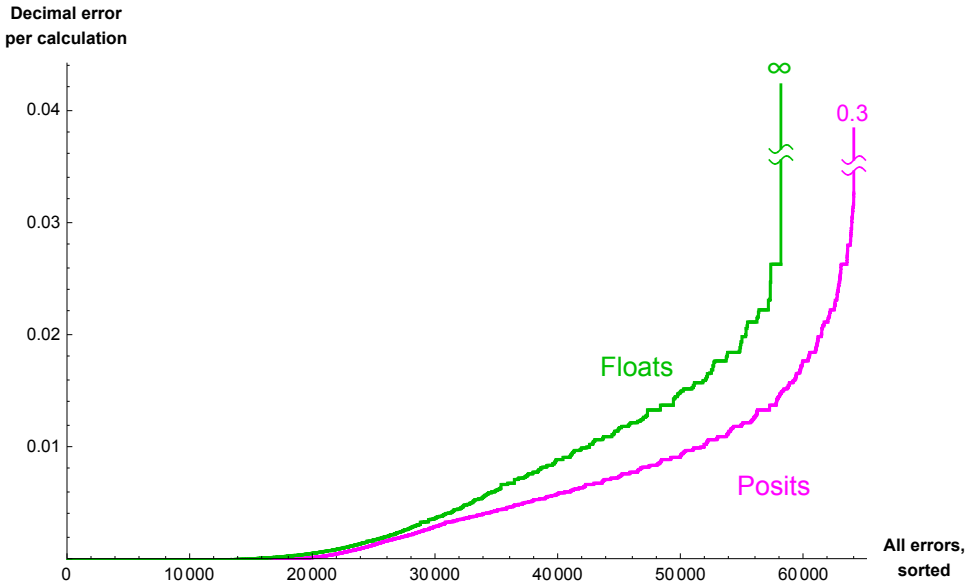
near the center of the dynamic range.

It may look like there are no NaN cases, but there is exactly *one* amber square, in the extreme bottom left, representing what happens when you add ±∞ to ±∞. Overflow cases have been eliminated.

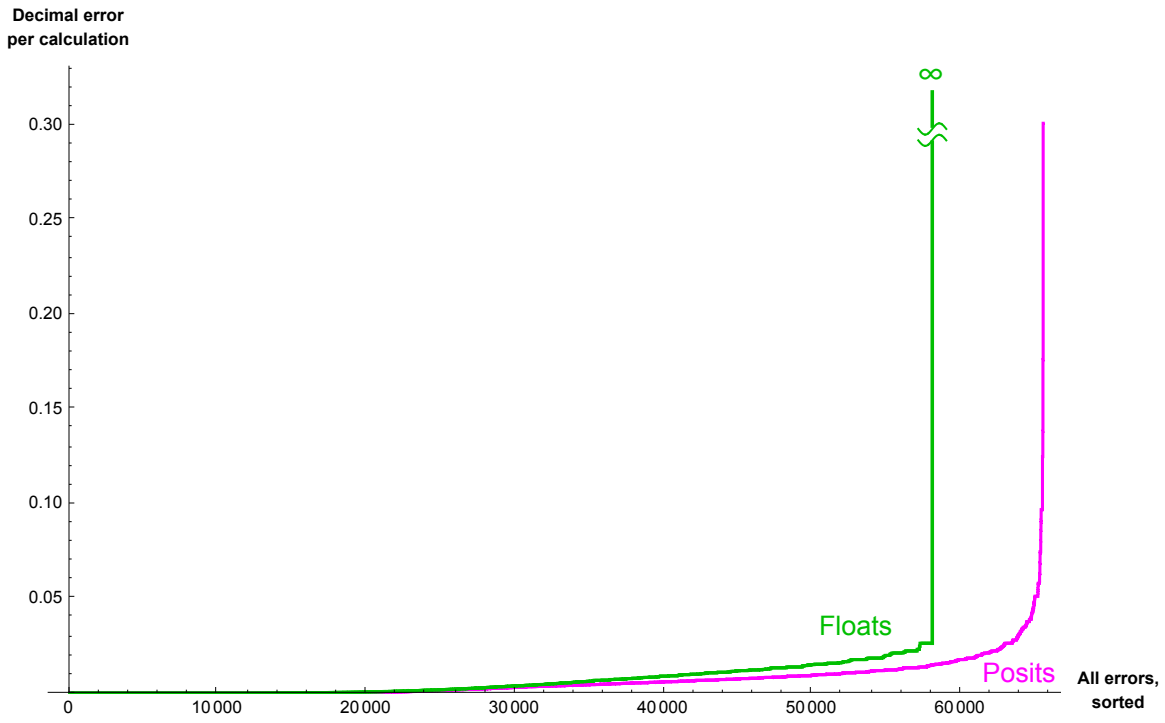
Here is a summary of addition performance for floats versus posits:



The greater incidence of *exact* sum operations is immediately evident. As with the single-operand functions, we can sort the decimal errors and plot to compare accuracy loss for float versus posit addition:



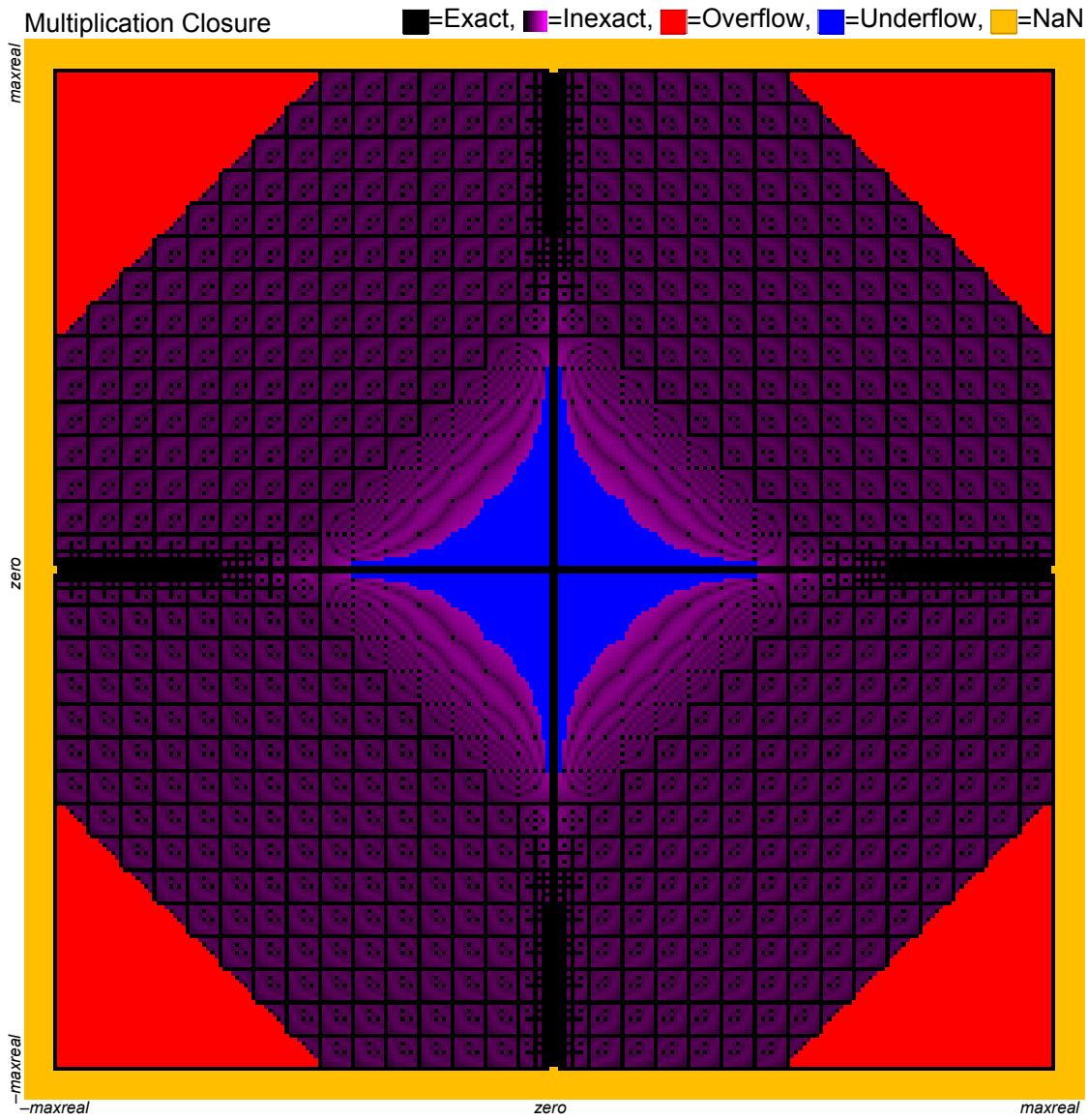
For clarity, that graph only plots the first 64 000 of the 65 536 errors. Plotting the entire set shows that the last few posit addition cases have a decimal error as high as 0.3, but that only happens when expressing numbers well beyond the dynamic range of floats anyway.



The worst case is if *maxpos* + *maxpos* is rounded back to *maxpos*, a decimal error of $\log_{10}(2) \approx 0.3$. IEEE floats instead "round" the result to **infinity**, thereby creating an infinite decimal error.

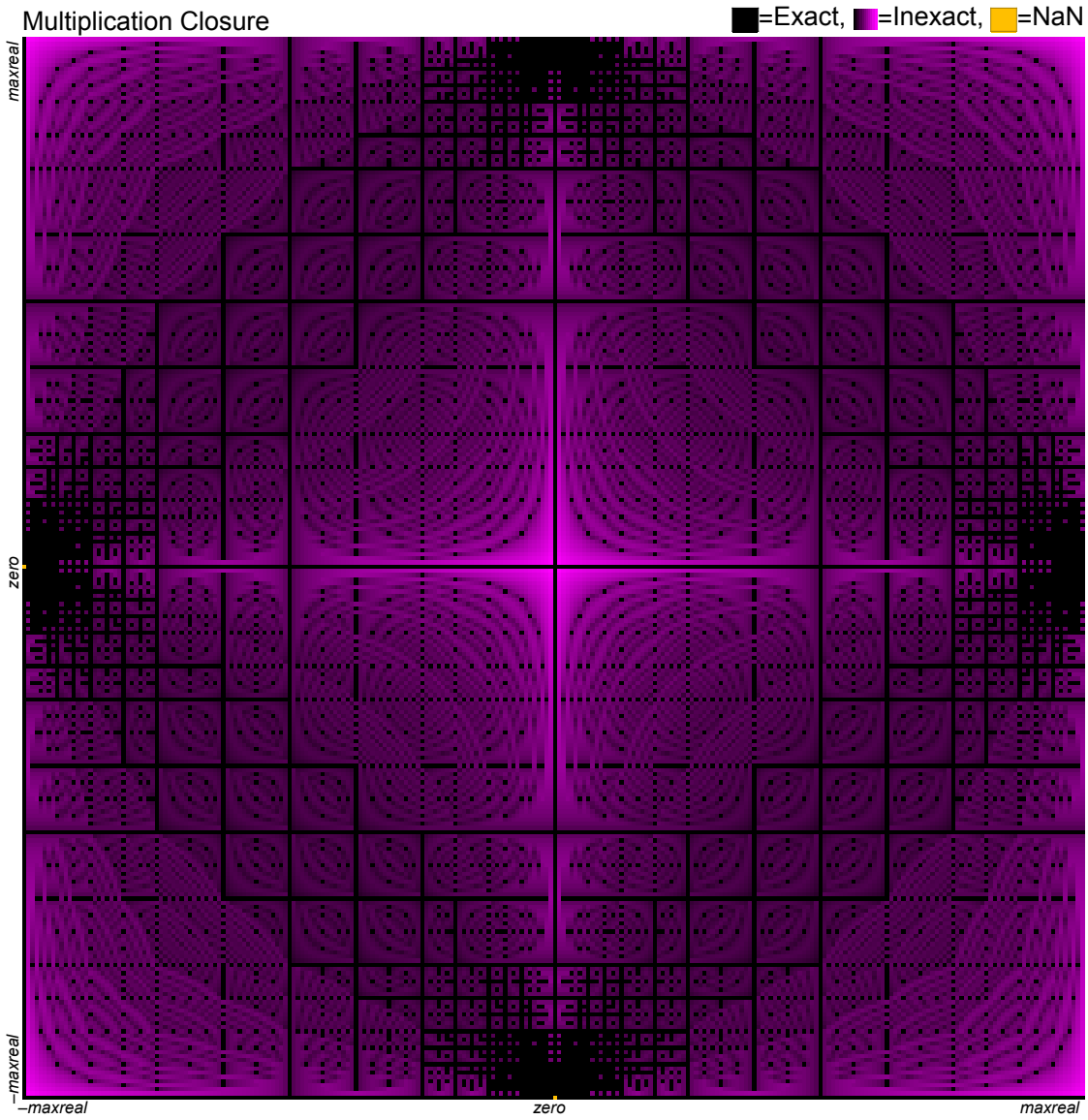
7.6.2 Comparing multiplication closure

We use a similar approach for comparing how well floats and posits multiply. Unlike addition, multiplication can cause floats to underflow. The "gradual underflow" region provides some protection, as you can see in the center of the closure graph. Without denormalized floats, the blue underflow region would be a full diamond shape.



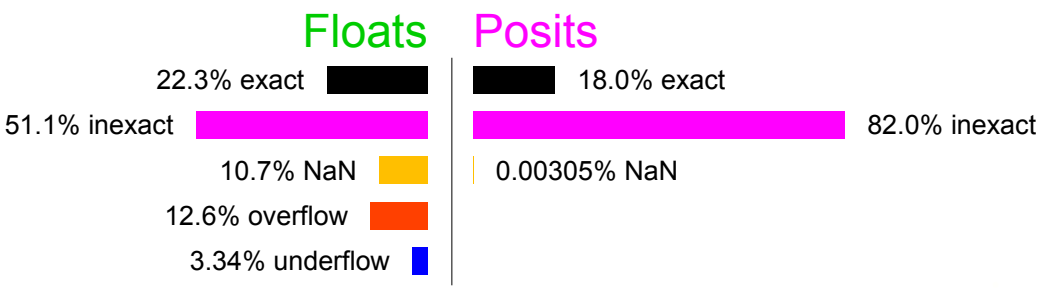
Multiplication Closure	
22.272%	exact
49.481%	inexact
4.950%	underflow
12.646%	overflow
10.651%	NaN

The plot for posit multiplication is much less colorful, and that's a good thing. Only two pixels light up as NaN, right next where the axes have their "zero" label. That is where ±∞ is multiplied by 0.

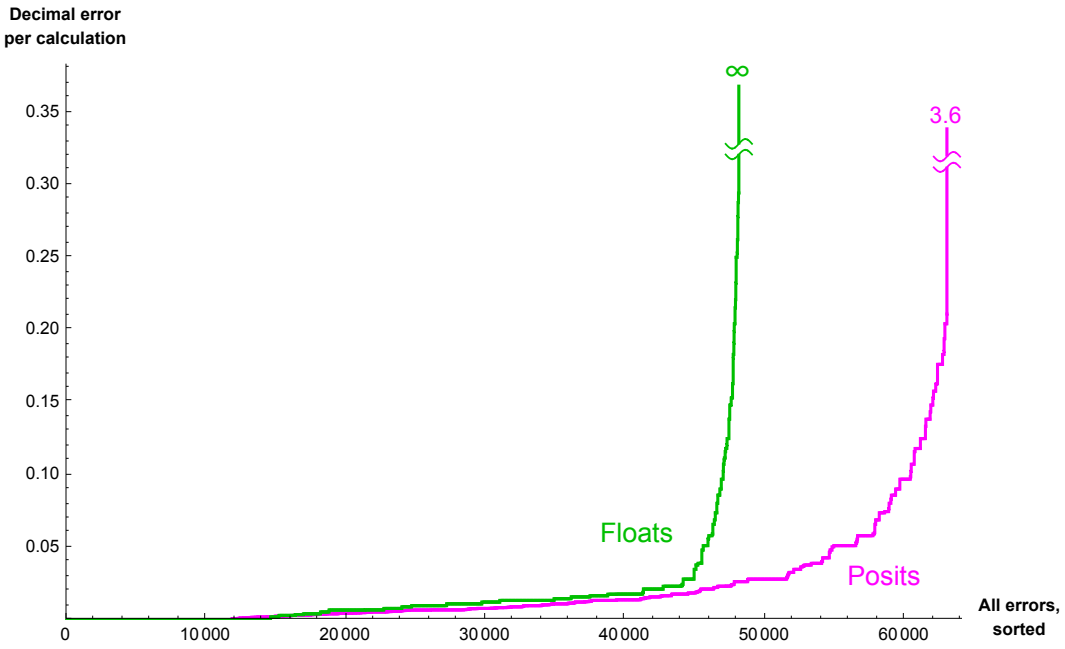


Multiplication Closure	
18.002%	exact
81.995%	inexact
0.000%	underflow
0.000%	overflow
0.003%	NaN

The floats have more cases where the product is exact than do posits, but at a *terrible cost*. As the diagrams show, almost one-quarter of all float products will overflow or underflow, and that fraction does not decrease for higher precision floats.

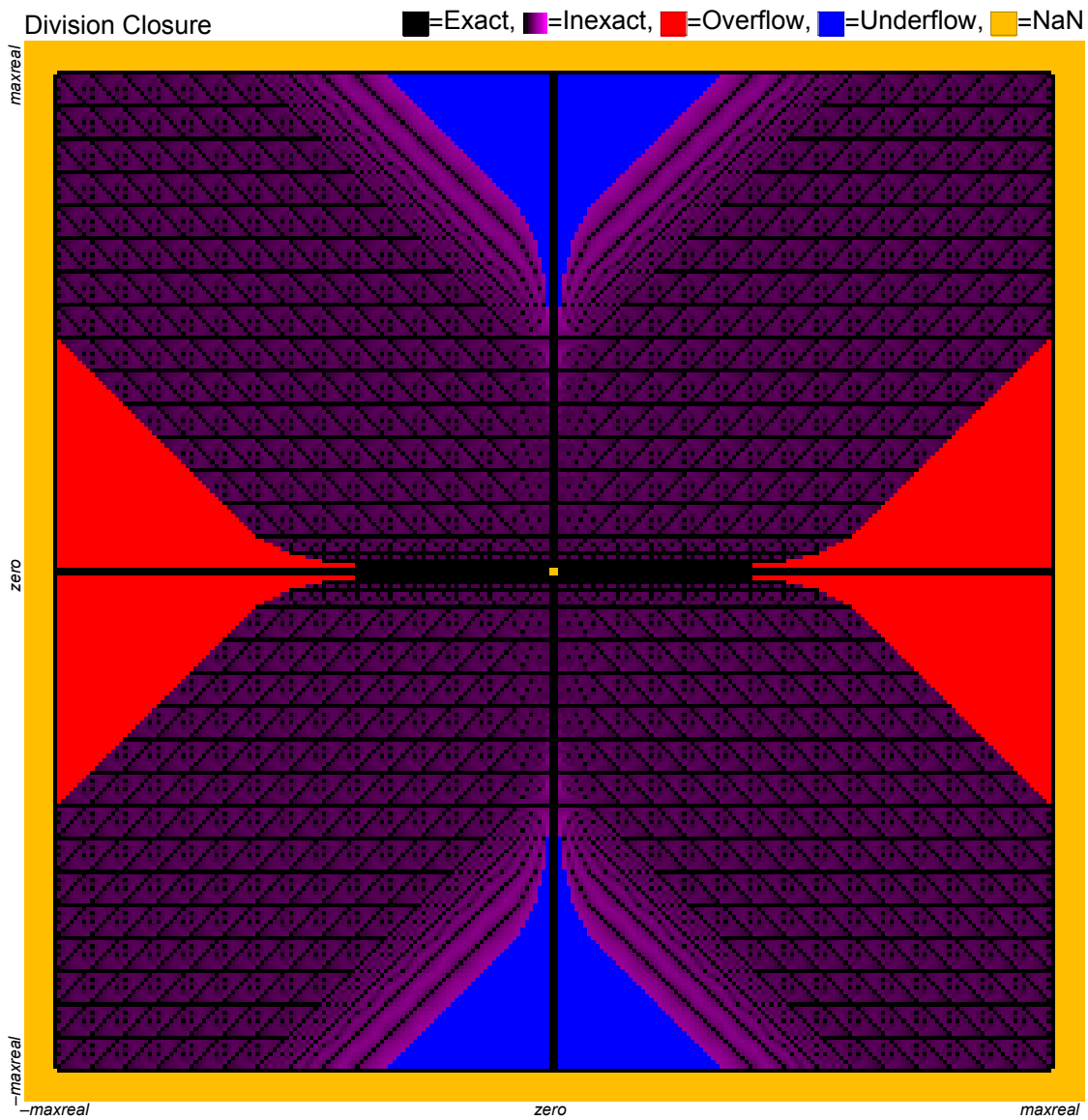


The worst-case rounding for posits occurs for *maxpos* × *maxpos*, which is rounded back to *maxpos*. For this set of posits, that represents a (very rare) loss of about 3.6 decimals. As the following graph shows, posits are dramatically better at minimizing multiplication error than floats:



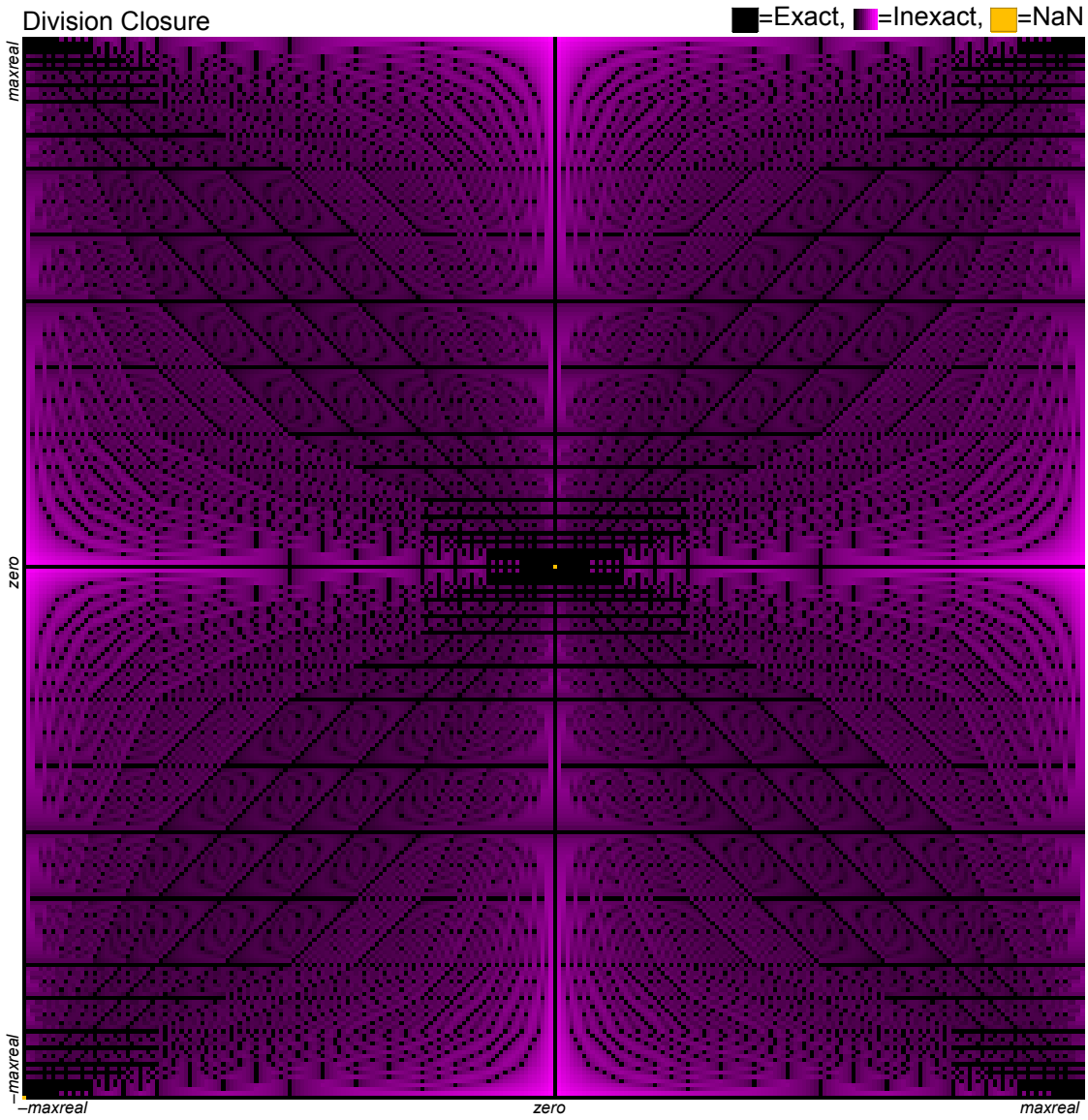
7.7.3 Comparing division closure

Similarly, we can compare the closure for division operations. It is almost a permutation of the quadrants of the multiplication plot.



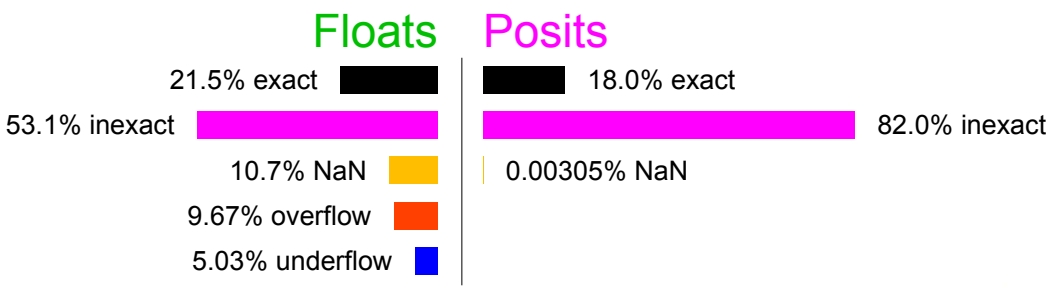
Division Closure	
21.539%	exact
51.276%	inexact
6.866%	underflow
9.668%	overflow
10.651%	NaN

The situation is similar to that for multiplication; somewhat more cases of exact results than for posits, but almost a quarter of the cases overflow or underflow catastrophically. Notice that the posit closure plot comes much closer to looking symmetrical about the diagonal line from top left to bottom right:



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

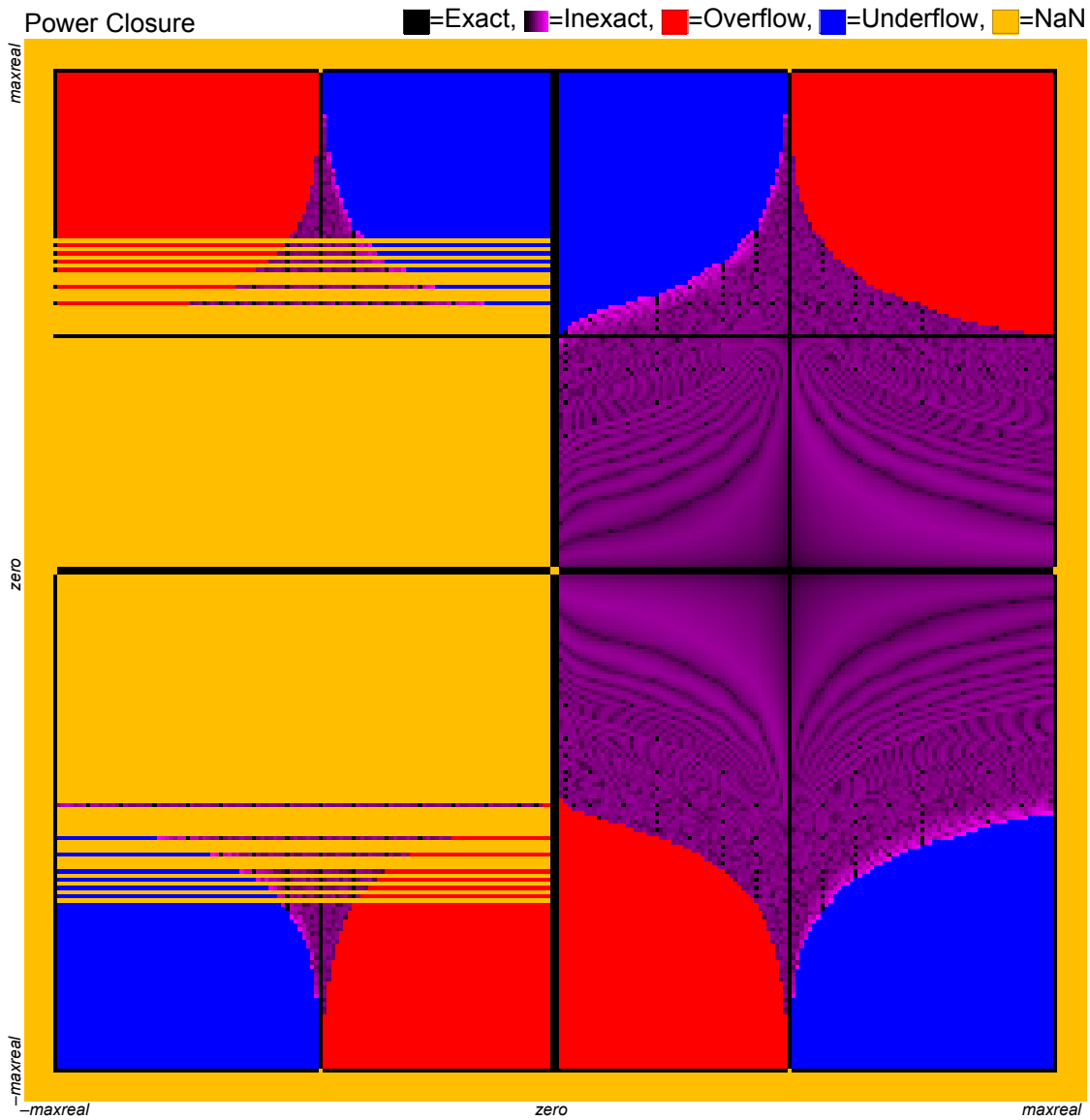
The two NaN values are visible in the bottom left corner ($\pm\infty/\pm\infty$) and the center ($0/0$). Here is a summary of the comparison for division:



The plot of sorted errors is almost identical to the one for multiplication, so there is no need to repeat it here.

7.7.4 Comparing power closure

Let's try to compare the closure for x^y operations. This is far more difficult to compute than the standard two-argument arithmetic operations, and many of the results are complex-valued and thus turn into NaN. Also, 0^0 and 1^∞ are NaN. The pattern is bizarre and spectacular:

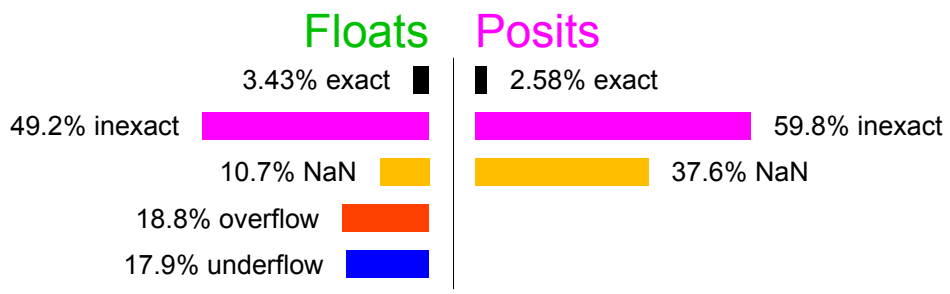


Power Closure	
3.433%	exact
25.734%	inexact
16.835%	underflow
17.194%	overflow
36.804%	NaN

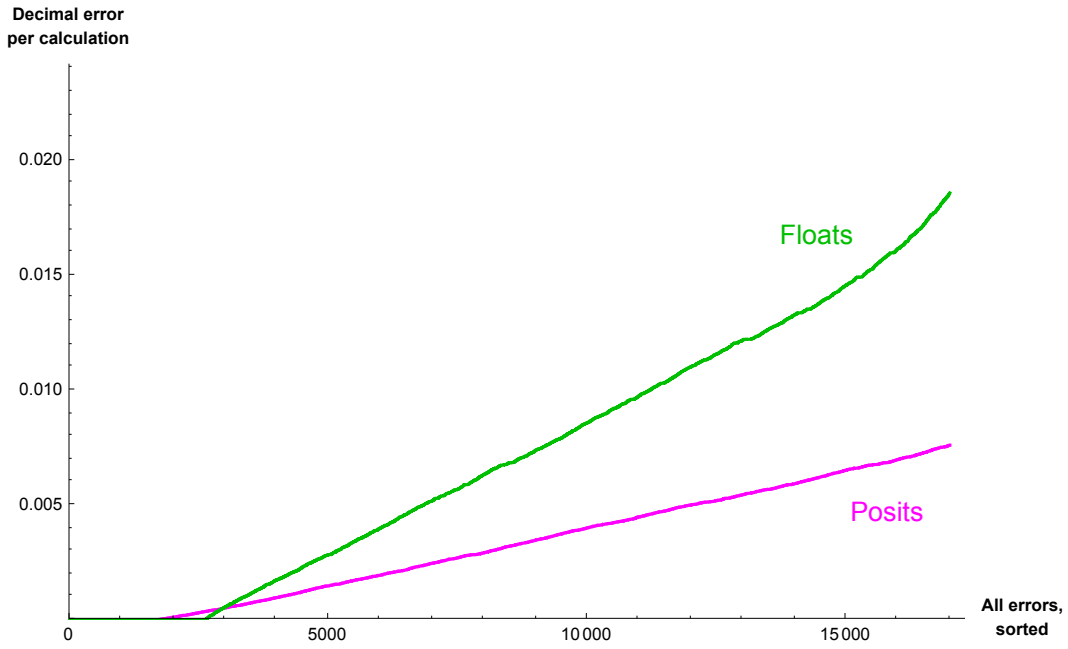
The posit closure plot is similarly thought-provoking. In general, a negative real to the power of a negative real has an imaginary component and thus trips the NaN indicator for both floats and posits, creating a large gold band on the left. But there are exceptions, like when the exponent is a negative *even integer*. In the extreme bottom left and top left corner, all the posits are negative even integers.

Power Closure	
2.579%	exact
59.821%	inexact
0.000%	underflow
0.000%	overflow
37.601%	NaN

Notice that 0^0 is a NaN, with amber squares in the center of each plot. The float square is larger because it has two kinds of zero, hence four ways to land on 0^0. Here is a summary of the comparison for the power function:



The plot of sorted errors eventually goes wildly high, but for the region of values that are not NaN and are not very large or very small magnitude, posits do their usual accuracy improvement, though there is a small region on the left where floats get a “head start” by having slightly more exact values.

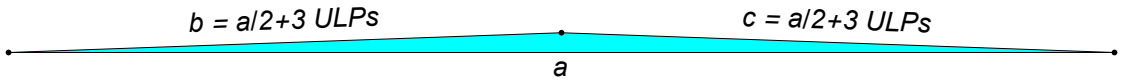


8 A Quad-Precision Test Example

8.1 The Area of a Very Thin Triangle

Here’s a classic “thin triangle” problem, taken from *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, by David Goldberg, published in the March, 1991 issue of *Computing Surveys*:

> Find the area *A* of a triangle with sides *a, b, c*, when two of the sides *b* and *c* are together *barely* longer than half the longest side, *a*. Specifically, two of the sides are 3 ULPs larger than half the longest side.



Think of it like pavement buckling in the hot sun. The classic formula for the area A uses a temporary value s:

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

The problem with the formula for a thin triangle is that s is very close to the value of a, and the calculation of s accumulates a rounding error of two ULPs… so the relative error is magnified about as badly as one could ever imagine. We want to test the ability of posits to surpass floats at high precision, so we will make the sides

$$a = 7$$
$$b = c = 7 + 3\times 2^{-111}$$

which are values that an IEEE quad precision (128-bit) float can express exactly. Suppose all values are in *light years*. Sides b and c are larger than half of a by only about 1/200 the diameter of a proton, yet that is enough to pop the triangle up to a height of about 85 centimeters, the width of a standard door opening! Let's use the area formula with *Mathematica* extended precision rational values to see what the correct value for the area is (in square light years), to forty decimals of accuracy:

```
area = Module[{a = 7, b = 7 / 2 + 3 × 2^-111, c, s}, c = b;
  s = (a + b + c) / 2;
  N[√(s (s - a) (s - b) (s - c)), 39]]
```

$$3.14784204874900425235885265494550774498 \times 10^{-16}$$

That's in square light years, a hard quantity to visualize. The area is about 55 times the surface area of the earth.

8.2 Thin triangle area with IEEE quad precision floats

IEEE quad precision uses 15 bits of exponent, so we can set the float environment and check the dynamic range as follows:

```
setfloatenv[{128, 15}]
{N[smallsubnormal], N[maxfloat]}
```

$$\{6.475175119438025 \times 10^{-4966},\ 1.189731495357232 \times 10^{4932}\}$$

Here is the calculation with floats, using the underbar to mean the float version of whatever is above it. That is, an underbar rounds whatever it underlines to the nearest float.

```
a = 7; b = c = 7 / 2 + 3 × 2^-111;
s = ((a + b) + c) / 2;
t = ((s × (s - a)) × (s - b)) × (s - c);
N[√t, 36]
```

$3.63481490842332134725920516158057683 \times 10^{-16}$

That's incorrect by more than 15 percent, a massive error considering that IEEE quad precision has about 33 decimals of precision. Coloring incorrect digits in orange, floats give us

$$3.63\,481\,490\,842\,332\,134\,725\,920\,516\,158\,057\,683 \times 10^{-16}$$

In other words, it got *one* decimal digit correct. Since the measurement unit is square light years, the error is about as big as the surface area of the planet Neptune.

8.3 Thin triangle area with posits: an unfair fight

Now try it with posits, simply by using the overbar instead of the underbar. Choosing *es* = 7 gives a posit environment a very similar dynamic range to that of IEEE 128-bit floats (quad precision):

```
setpositenv[{128, 7}]
{N[minpos], N[maxpos]}
```

$\{9.73262367930742 \times 10^{-4856}, 1.027472172920962 \times 10^{4855}\}$

```
a = 7; b = c = 7 / 2 + 3 × 2^-111;
s = ((a + b) + c) / 2;
t = ((s × (s - a)) × (s - b)) × (s - c);
N[√t, 39]
```

$3.14784204874900425235885265494550774439 \times 10^{-16}$

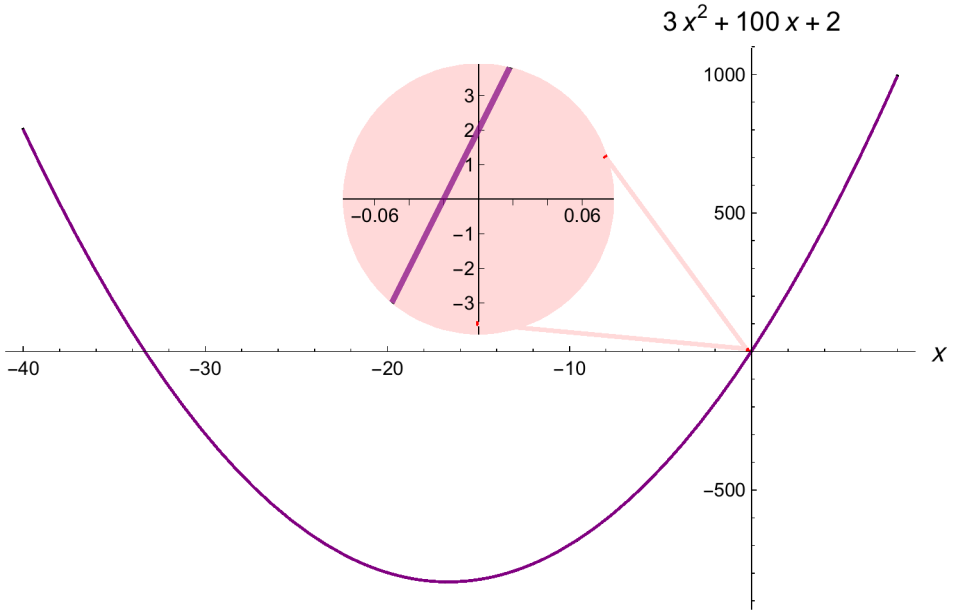
Or with the orange wrong-digit notation, posits give us an area of

$$3.14784204874900425235885265494550774439 \times 10^{-16}$$

This ultra-precise answer (correct to 37 decimals) could be converted to a 16-bit posit and it would still have far more accuracy than the 128-bit float result.

9 The Quadratic Formula

We can trot out the example used in *The End of Error: Unum Computing*, the use of the quadratic formula $\frac{-b \pm \sqrt{b^2 - 4\,a\,c}}{2\,a}$ when b is quite a bit larger than a and c, causing loss of significant digits in one of the two roots. The coefficients used were $a = 3$, $b = 100$, $c = 2$.



The roots, correct to nine decimals, are

$$r_1 = \frac{1}{6}\left(-100 + \sqrt{9976}\right) = -0.0200120144\cdots$$
$$r_2 = \frac{1}{6}\left(-100 - \sqrt{9976}\right) = -33.3133213\cdots$$

The problem is that when b^2 is much larger than $a\,c$ and the square root is inexact, one root will have its relative inaccuracy magnified by the subtraction of similar quantities. Traditional numerical analysis texts teach that programmers should use a trick that rearranges the algebra.

Ideally, a number system should be robust enough that the unwary can use it **without memorizing a trick for every situation**.

First, find the roots using single-precision floats:

```
setfloatenv[{32, 8}]
quadf = {(-100 + Sqrt[100^2 - 4 × 2 × 3])/6, (-100 - Sqrt[100^2 - 4 × 2 × 3])/6};
Print["Computed roots for floats:  ", N[quadf, 9]]
Print["Absolute errors for floats: ",
  N[{Abs[quadf[[1]] - 1/6 (-100 + Sqrt[9976])], Abs[quadf[[2]] - 1/6 (-100 - Sqrt[9976])]}]]
```

```
Computed roots for floats:  {-0.0200119019, -33.3133202}
Absolute errors for floats: {1.12566 × 10^-7, 1.159 × 10^-6}
```

Here is the same calculation using posits:

```
setpositenv[{32, 2}]
quadp = {(-100 + Sqrt[100^2 - 4 × 2 × 3])/6, (-100 - Sqrt[100^2 - 4 × 2 × 3])/6};
Print["Computed roots for posits:  ", N[quadp, 9]]
Print["Absolute errors for posits: ",
  N[{Abs[quadp[[1]] - 1/6 (-100 + Sqrt[9976])], Abs[quadp[[2]] - 1/6 (-100 - Sqrt[9976])]}]]
```

```
Computed roots for posits:  {-0.0200120609, -33.3133216}
Absolute errors for posits: {4.64572 × 10^-8, 2.71512 × 10^-7}
```

The comparison is easier to see using orange wrong-digit notation:

	Floats	Posits	
r_1	−0.02001190 ⋯	−0.02001206 ⋯	posits get six digits right instead of four
r_2	−33.3133202 ⋯	−33.3133216 ⋯	posits get eight digits right instead of seven

Incidentally, if we *did* use the algebraic rearrangement trick, posits still beat floats. The trick is to write $\frac{-b+\sqrt{b^2-4ac}}{2a}$ as $\frac{2c}{-b-\sqrt{b^2-4ac}}$ to avoid scraping off significant digits by subtracting similar numbers. Like most numerical tricks, it is far from obvious, has to be worked out with tedious algebra, and is prone to errors in coding it. Here is the smaller magnitude root, computed using the trick with floats:

```
quadf = 2 × 2 / (-100 - √(100^2 - 4 × 2 × 3)); N[quadf, 9]
N[Abs[quadf - 1/6 (-100 + √9976)]]
```

```
-0.0200120136

8.07458 × 10^-10
```

When the trick is used with posits, the error is about 8 times smaller:

```
quadp = 4 / (-100 - √(100^2 - 4 × 2 × 3)); N[quadp, 9]
N[Abs[quadp - 1/6 (-100 + √9976)]]
```

```
-0.0200120143

1.08966 × 10^-10
```

That is,

	Floats	Posits	
r_1	−0.020012013**6** ⋯	−0.020012014**3** ⋯	posits get eight digits right instead of seven.

10 Fused Operations and the Quire Data Type

10.1 An overview of "deferred rounding"

So far, we have shown how posits could be a drop-in replacement for floats. However, they can be far more powerful than that. After porting a code from floats to posits, if the improved accuracy is still not enough (like when trying to replace 64-bit floats with 32-bit posits), the *quire* is easy to use and can produce massive increases in answer quality. There is nothing like it in floating-point environments because the IEEE 754 committee has, for over thirty years, firmly and repeatedly *rejected* requests to put something like it into the Standard.

The most recent version (2008) of the IEEE 754 standard includes the *fused multiply-add* in its repertoire. "Fusing" means deferring the rounding until the last operation in a computation involving more than one operation, by performing all operations using exact integer arithmetic in a scratch area with a set size. It is *not* the same as extended-precision arithmetic, which can increase the size of integers until the computer runs out of memory. The posit environment supports the following fused operations, using a fixed-point scratch value called a *quire*.

Fused multiply-add	$(a \times b) + c$
Fused add-multiply	$(a + b) \times c$
Fused multiply-multiply-subtract	$(a \times b) - (c \times d)$
Fused sum	$\sum a_i$
Fused dot product (scalar product)	$\sum a_i\, b_i$

(By the way, the dictionary tells that “quire” was a medieval term for a booklet made from a fixed number of pieces of paper or parchment.) We need to find out how many bits of quire are needed as a function of *nbits* and *es*. Notice that all of the operations in the above list are *subsets* of the fused dot product in terms of their processor hardware requirements.

As with all unum environments, fusing of operations must be **explicit**, never **covert**. If a programmer writes

```
x := (a * b) + c;
```

where all variables are of the same type, a compiler designer might be tempted to “help out” by using the fused multiply-add. A likely result is that the programmer gets a different answer than that produced by another compiler that rounds after the multiply and again rounds after the add. Tracking that down can produce many wasted hours trying to figure out why. In a posit environment, you could request the fused multiply-add with a routine call:

```
x := FMA (a, b, c);
```

But what might be easier to read, and more flexible to use, is to declare a quire variable, like this:

```
posit a, b, c, x;
quire q;
q = (a * b) + c;
x = q;
```

Because the multiply-accumulation into the quire has no rounding error, this is as repeatable and reproducible as an integer multiply-add. It is more *flexible*, because you could also do things this way:

```
posit a, b, c, x;
quire q;
q = a * b;
(*  Some other code can go here, like to compute c.  *)
q += c;
x = q;
```

We go far beyond fusing just one multiply-add, however, and permit fusing of a bunch of them. At first, you might think this would require an extended-precision math library and an unpredictable

amount of storage, but that is not the case. We simply have to put a limit on how many accumulates we can do without risk of overflow. It turns out that it can be quite a high limit, especially for 32-bit and 64-bit posits.

The quire approach was introduced by Ulrich Kulisch in the 1970s, at a time when transistors were too precious to think about putting a fixed-size register with hundreds of bits into a processor. Kulisch and his colleagues have developed a vast and rich set of techniques for doing large-scale calculations where the results are accurate to a *single rounding error*. He called the storage an "exact accumulator," but that's a mouthful and we prefer the simpler coined term, "quire." All of the techniques Kulisch created are once again available in posit environments.

Perhaps the biggest surprise is this: Using a quire can make a code *faster*, not slower. And not by just a little bit: the most recent hardware tests by Koenig and Biancolin at the ASPIRE lab at Berkeley show that quire operations are about 3 to 6 times faster than rounding after every operation. If an approach makes code that much faster and also makes it much more accurate, why would anyone not want to adopt it?

10.2 Quire size

The smallest magnitude nonzero value that can arise in doing a dot product is $minpos^2$. Every other product is an integer multiple of $minpos^2$. If we have to perform the dot product of vectors {*maxpos*, *minpos*} and {*maxpos*, *minpos*} as an exact operation in a scratch area, we need an integer big enough to hold $maxpos^2 \,/\, minpos^2$. Recall that $maxpos = useed^{nbits-2}$ and $useed = 2^{2^{es}}$. Also, $minpos = 1/maxpos$. So

$$\frac{maxpos^2}{minpos^2} = useed^{2\times 2\times (nbits-2)} = 2^{(4\,nbits-8)\,2^{es}}$$

As a fixed-point unsigned integer, $(4\,nbits - 8)\,2^{es} + 2$ bits can hold the sum of $minpos^2$ and $maxpos^2$. For example, for 8-bit posits with $es = 0$, *minpos* is $2^{-6} = 1/64$ and *maxpos* is $2^6 = 64$. So you need 12 bits to the right of the binary point, and 13 bits to the left of the binary point, to hold $maxpos^2 + minpos^2 = 4096 + 1/4096$ as a fixed-point integer:

$$4096 + \frac{1}{4096} = \mathbf{1\,000\,000\,000\,000.000000000001}$$

We need one more bit for the sign. But we also need to allow for some carry bits in the worst case where the numbers added together were all $maxpos^2$ (or all $-maxpos^2$). If we accommodate at least a billion terms in the sum, and sometimes far more, there shouldn't be much complaining. So add 30 bits to the integer size. (2^{30} is a little over a billion: 1 073 741 824.), And for good measure, round that up to the nearest power of two so the hardware design is clean. Here is our definition for the size of the quire, **quiresize**, for posit sizes from 4 to 256. (The reason for including sizes 4 and 8 will become more clear when we define rules for the *valid* type.)

```
quiresize[n_, e_] := 2^⌈Log[2, (4 n-8) 2^e+1+30]⌉
quireexcess[n_, e_] := quiresize[n, e] - ((4 n - 8) 2^e + 2)
```

With these we can generate the table for the standard posit sizes:

Posit size, nbits	Posit *es* value	Quire size (bits)	Excess bits for carries
8	0	64	38
16	1	256	142
32	2	512	30
64	3	2048	62
128	4	8192	126
256	5	32 768	254

The quire sizes are large if you think of them as registers, but far smaller than what it takes to do extended-precision arithmetic like *Mathematica* uses. A modern processor core might have 64 general-purpose 64-bit registers; the quire for 64-bit posits would fit in 32 of those.

Earlier, we mentioned that posits as small as 8 bits suffice for Machine Learning. That is true even if the multiply-accumulation is stored with only 8 bits. It seems likely that the quire for 8-bit posits, being only 64 bits in size and capable of doing a quarter-trillion accumulates without the possibility of overflow, could be quite useful in such applications, perhaps improving the rate of convergence.

10.3 What quires can and cannot do

A programmer might think, “These quire variables are so safe and accurate, I think I’ll make all my variables of quire type.” Not so fast. The list of things you can reasonably ask hardware to do is a short one, and it does not include *multiplying* the quire by anything. It’s for *accumulation*. Assume there is only one quire register in a core. The assembler instructions should include

- Clear the quire
- Load the quire from memory
- Store the quire to memory
- Add the product of two posits to the quire
- Subtract the product of two posits from the quire
- Add a quire stored in memory to the quire
- Subtract a quire stored in memory from the quire
- Convert a quire into a posit

From those, it is easy to build up expressions like $a \times d - b * c$ in the quire. Simple summation is a special case of multiplying by 1, as is done with fused multiply-add hardware. If you want to multiply the number in the quire by something, you have to convert it back to a posit. I have to

admit to being tempted to add one more assembler instruction to the list because it would be so incredibly useful:

- Convert the square root of a quire into a posit

That would let you evaluate things like $r = \sqrt{x^2 + y^2 + z^2}$ with only one rounding error, and the result r would be about the same magnitude as the largest of x, y, and z instead of having to temporarily store r^2 in a posit, which might land where the magnitude is so large or small that accuracy is lost. Physics simulations are full of such calculations of distance, and the square root algorithm is a straightforward one to build into hardware; the concern is that for a large quire, the operation could take many clock cycles to determine the correct rounding, making context switching coarse-grained and difficult.

We won't be seeing the quire explicitly in the sections that follow, because we have the luxury of operating in Mathematica, for which quire exact operations are small subset of what it can do with its built-in extended-precision arithmetic. If we don't put an overbar on a result, it is assumed exact and not rounded. We simply have to restrict the operations where we defer rounding to the above list.

11 Fast Fourier Transforms (*incomplete section*)

Here is a garden-variety complex Fast Fourier Transform, in both float form and posit form. The float form uses fused multiply-add, and the posit form goes slightly farther in fusing complex operations of the form $a*b - c*d$.

```
(* Conventional power-of-two Fast Fourier Transform with floats. *)
cfftf[rr_List, n_Integer, iflg_Integer] :=
 Module[{gg = rr, k = n / 2, th = If[iflg ≥ 0, -π, π], tw, ww, tr, ti, i, j},
  While[k ≥ 1,
   ww = {-2 (Sin[th / (2 k)])^2, Sin[th / k]};
   tw = {1, 0};
   For[j = 0, j < k, j++,
    For[i = 1, i ≤ n, i += 2 k,
     {tr, ti} = {gg⟦i+j,1⟧ - gg⟦i+j+k,1⟧, gg⟦i+j,2⟧ - gg⟦i+j+k,2⟧};
     {gg⟦i+j,1⟧, gg⟦i+j,2⟧} = {gg⟦i+j,1⟧ + gg⟦i+j+k,1⟧, gg⟦i+j,2⟧ + gg⟦i+j+k,2⟧};
     {gg⟦i+j+k,1⟧, gg⟦i+j+k,2⟧} = {tw⟦1⟧ * tr - tw⟦2⟧ * ti, tw⟦1⟧ * ti + tw⟦2⟧ * tr}];
    tr = tw⟦1⟧ * ww⟦1⟧ - tw⟦2⟧ * ww⟦2⟧ + tw⟦1⟧;
    tw⟦2⟧ = tw⟦1⟧ * ww⟦2⟧ + tw⟦2⟧ * ww⟦1⟧ + tw⟦2⟧;
    tw⟦1⟧ = tr];
   k = k / 2];
  For[i = j = 0, i < n - 1, i++,
   If[i < j,
    {tr, ti} = {gg⟦j+1,1⟧, gg⟦j+1,2⟧};
    {gg⟦j+1,1⟧, gg⟦j+1,2⟧} = {gg⟦i+1,1⟧, gg⟦i+1,2⟧};
    {gg⟦i+1,1⟧, gg⟦i+1,2⟧} = {tr, ti}];
   k = n / 2; While[k ≤ j, {j = j - k; k = k / 2}]; j = j + k];
  gg / Sqrt[n]]
```

```
(* Conventional power-of-two Fast Fourier Transform with posits. *)
cfftp[rr_List, n_Integer, iflg_Integer] :=
 Module[{flg = 0, gg = rr, k = n / 2, th = If[iflg ≥ 0, -π, π], tw, ww, tr, ti, i, j},
  While[k ≥ 1,
   ww = {Conjugate[-2 (Sin[th / (2 k)])^2], Conjugate[Sin[th / k]]};
   tw = {1, 0};
   For[j = 0, j < k, j++,
    For[i = 1, i ≤ n, i += 2 k,
     {tr, ti} = {Conjugate[gg[[i+j,1]] - gg[[i+j+k,1]]], Conjugate[gg[[i+j,2]] - gg[[i+j+k,2]]]};
     {gg[[i+j,1]], gg[[i+j,2]]} = {Conjugate[gg[[i+j,1]] + gg[[i+j+k,1]]], Conjugate[gg[[i+j,2]] + gg[[i+j+k,2]]]};
     {gg[[i+j+k,1]], gg[[i+j+k,2]]} = {Conjugate[tw[[1]] * tr - tw[[2]] * ti], Conjugate[tw[[1]] * ti + tw[[2]] * tr]}];
    tr = Conjugate[tw[[1]] * ww[[1]] - tw[[2]] * ww[[2]] + tw[[1]] * 1]; (* Short fused dot product*)
    tw[[2]] = Conjugate[tw[[1]] * ww[[2]] + tw[[2]] * ww[[1]] + tw[[2]] * 1];
    tw[[1]] = tr];
   If[flg == 0, gg /= 2; flg = 1, flg = 0];
   k = k / 2];
  For[i = j = 0, i < n - 1, i++,
   If[i < j,
    {tr, ti} = {gg[[j+1,1]], gg[[j+1,2]]};
    {gg[[j+1,1]], gg[[j+1,2]]} = {gg[[i+1,1]], gg[[i+1,2]]};
    {gg[[i+1,1]], gg[[i+1,2]]} = {tr, ti}];
   k = n / 2;
   While[k ≤ j, {j = j - k; k = k / 2}]; j = j + k];
  gg]
```

First, try using half-precision floats.

```
setfloatenv[{16, 5}]
```

For example, just put a 1 in the second entry, and see if we can do a forward FFT and an inverse FFT and get something similar to what we started with:

```
n = 16; rr = Table[{0, 0}, n]; rr[[2]] = {1, 0}; rr
```

```
{{0, 0}, {1, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0},
 {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}}
```

```
rrf = cfftf[cfftf[rr, n, 1], n, -1]
```

$$\left\{\{0,0\},\left\{1,-\frac{985}{8\,388\,608}\right\},\{0,0\},\left\{-\frac{1333}{16\,777\,216},\frac{819}{16\,777\,216}\right\},\right.$$
$$\{0,0\},\left\{-\frac{1}{32\,768},-\frac{1}{32\,768}\right\},\{0,0\},\left\{-\frac{437}{16\,777\,216},\frac{821}{67\,108\,864}\right\},$$
$$\{0,0\},\left\{0,\frac{473}{8\,388\,608}\right\},\{0,0\},\left\{\frac{437}{16\,777\,216},\frac{821}{67\,108\,864}\right\},$$
$$\left.\{0,0\},\left\{\frac{1}{32\,768},-\frac{1}{32\,768}\right\},\{0,0\},\left\{\frac{1333}{16\,777\,216},\frac{819}{16\,777\,216}\right\}\right\}$$

The rounding errors hit the points where the sines and cosines are not special angles. Here is the total error:

```
N[Total[Total[Abs[rr - rrf]]]]
```

```
0.000628978
```

Now do the same thing with posits:

```
setpositenv[{16, 2}];
rrp = cfftp[cfftp[rr, n, 1], n, -1]
N[Total[Total[Abs[rr - rrp]]]]
```

$$\left\{\{0,0\},\left\{1,-\frac{261}{8\,388\,608}\right\},\{0,0\},\left\{0,\frac{49}{16\,777\,216}\right\},\{0,0\},\right.$$
$$\left\{\frac{69}{2\,097\,152},\frac{61}{1\,048\,576}\right\},\{0,0\},\left\{0,\frac{49}{16\,777\,216}\right\},\{0,0\},\left\{0,\frac{53}{2\,097\,152}\right\},\{0,0\},$$
$$\left.\left\{0,\frac{49}{16\,777\,216}\right\},\{0,0\},\left\{\frac{187}{2\,097\,152},-\frac{67}{1\,048\,576}\right\},\{0,0\},\left\{0,\frac{49}{16\,777\,216}\right\}\right\}$$

```
0.000312209
```

The usual benchmark is for 1024 points. Fill all the values with –1, 0, or 1 chosen randomly (seismic explorers and radio astronomers often use such low-precision inputs) and measure the total error:

```
SeedRandom[141 213 562];
n = 1024;
rr = Table[{Random[], Random[]}, n];
rrf = cfftf[cfftf[rr, n, 1], n, -1];
N[Total[Total[Abs[rr - rrf]]]]
```

```
1.74042
```

Here is the 1024-point case done with posits:

```
setpositenv[{16, 1}]
```

```
SeedRandom[141 213 562];
n = 1024;
rr = Table[{Random[], Random[]}, n];
rrp = cfftp[cfftp[rr, n, 1], n, -1];
N[Total[Total[Abs[rr - rrp]]]]
0.617407
```

12 Linear Algebra with Posits: A New Paradigm

12.1 Gaussian elimination is an *iterative method*

One thing floats do well is to guess answers x to a system of linear equations, $A\,x = b$ where A is an n by n matrix and b is a given vector of n values. (At least, they do well *under certain conditions*, and people have learned to live with those restrictions.) The LINPACK benchmark fills A with random numbers between −1 and 1, and assigns the sum of each row to the values of b, so that the answer *should* be $x = \{1, 1, \ldots, 1\}$. We use both the underbar and overbar operator to make sure each number in A is in the common vocabulary of the two number systems. That benchmark specifies 64-bit representation of numbers in a way that looks like some kind of floating-point, not necessarily IEEE 754 Standard; therefore, 64-bit posits can certainly be used on the LINPACK benchmark!

There's a good reason for requiring 64-bit precision, which we will now explore. What would happen, say, if we tried using *16-bit* floats and posits? Also, what happens if we recognize that so-called "direct solvers" are simply iterative solvers that usually stop after one iteration? (Unless Gaussian elimination is performed with infinite precision, it will only approximate the underlying mathematics and in general must be applied iteratively to correct the residual error.)

We also do something new, which is to **correct an error in the design of the benchmark rules**. The driver assumes the answer is $x = \{1, 1, \ldots, 1\}$, but that incorrectly assumes that there was no rounding in computing b! The values in A have a range of magnitudes, so the exact sum of a row will in general contain more significant bits than fit exactly into the fraction part of a float or a posit. Therefore, we tweak the smallest value in each row of A to make sure that there is *no* rounding in the right-hand side b. Here is an example using 16-bit IEEE floats. The **powQ** helper function returns `True` if a number ***a*** can be exactly represented in ***p*** bits, and `False` if it cannot.

```
powQ[a_, p_] := Module[{abits = RealDigits[a, 2], len, nd},
  nd = abits[[2]];
  len = Length[abits[[1]]];
  Which[
   p > len - nd, False,
   abits[[1, p+nd]] == 0, False,
   True, True]]
```

Set up the environments. Create the A matrix one row at a time, tweaking it in the least-significant bits until it sums to a number that can be represented exactly; if the tweaking fails, which happens

in certain uncommon cases, it simply generates a fresh row and tries again. This may seem reminiscent of *reverse error analysis*, where the answer produced by arithmetic that rounds is considered to be the exact solution to a problem that is *close* to the one specified; however, that is not really what this is. This is completing the design goal of LINPACK, which is to create an idealized problem where both the input values and the correct answer are expressible in the numerical vocabulary of the computing environment.

```
setfloatenv[{16, 5}]
setpositenv[{16, 2}]
SeedRandom[141 213 562]; n = 100;
b = Table[0, n]; a = Table[b, n];
Module[{abits, at, bbits, bt, (*den,*)fs, i, j, pow, row, temp},
 For[i = 1, i ≤ n, i++,
  Label[L1];
  at = Table[RandomReal[{-1, 1}], {j, 1, n}];
  bt = ∑_{j=1}^{n} at⟦j⟧;
  fs = Min[nfbits - esize, 1 + Length[fractionbits[x2p[bt]]]];
  bbits = RealDigits[bt, 2];
  For[j = Length[bbits⟦1⟧], j ≥ fs + 1, j--,
   pow = j - bbits⟦2⟧;
   If[bbits⟦1,j⟧ == 1,
    temp = 1; While[temp ≤ n ⋀ ¬ powQ[at⟦temp⟧, pow], temp++];
    If[temp > n, Goto[L1]]; (* Failed; go get another row. *)
    at⟦temp⟧ -= 2^-pow
   ]
  ];
  a⟦i⟧ = at;
  b⟦i⟧ = ∑_{j=1}^{n} at⟦j⟧;
 ]]
```

Check that the row sums equal the right-hand side, using exact arithmetic (easily done using a quire):

```
Table[Sum[a⟦i,k⟧, {k, 1, n}] - b⟦i⟧, {i, 1, n}]
```

```
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

We solve the system LINPACK style. Gaussian elimination with partial pivoting, rounding after every operation. Here's the algorithm, about as terse as it is possible to make it without making calls to BLAS-like routines.

```
linsolver[a_, b_] := Module[
  {aa = Transpose[Join[Transpose[a], {b}]], piv, i, j, k, n = Length[b], temp},
  For[k = 1, k ≤ n, k++,
   piv = k - 1 + (Position[aa[[k;;n,k]], Max[aa[[k;;n,k]]]])[[1,1]];
   For[j = 1, j ≤ n + 1, j++, temp = aa[[piv,j]];
    aa[[piv,j]] = aa[[k,j]];
    aa[[k,j]] = temp];
   temp = 1 / aa[[k,k]]; For[j = k, j ≤ n + 1, j++, aa[[k,j]] *= temp];
   For[j = k + 1, j ≤ n + 1, j++,
    temp = aa[[k,j]];
    For[i = k + 1, i ≤ n, i++,
     aa[[i,j]] -= aa[[i,k]] * temp]]];
  For[j = n, j > 1, j--,
   temp = aa[[j,n+1]];
   For[i = j - 1, i ≥ 1, i--,
    aa[[i,n+1]] -= aa[[i,j]] * temp]];
  aa[[1;;n,n+1]]]
```

That solver will use exact rational-number arithmetic in *Mathematica*, so test it on our sample matrix:

```
linsolver[a, b]
```

```
{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
```

Had we not “purified” the input matrix, we would not have gotten that answer but instead seen a mess of rational numbers close to 1 with huge integers for their numerator and denominator.

For the IEEE float version of the solver, we help its accuracy as much as we can by using the *fused multiply-add* that is now part of the IEEE 754 Standard (2008). There are only four places in the following code where we need to round to the nearest float, as shown by the underscoring.

```
linsolverf[a_, b_] := Module[{aa = Transpose[Join[Transpose[a], {b}]],
   piv, i, j, k, nn = Length[b], temp},
  For[k = 1, k ≤ nn, k++,
   piv = k - 1 + (Position[Abs[aa⟦k;;nn,k⟧], Max[Abs[aa⟦k;;nn,k⟧]]])⟦1,1⟧;
   For[j = 1, j ≤ nn + 1, j++, temp = aa⟦piv,j⟧;
    aa⟦piv,j⟧ = aa⟦k,j⟧;
    aa⟦k,j⟧ = temp];
   (*Print[aa⟦k,k⟧];
   *)temp = 1 / aa⟦k,k⟧;
   For[j = k, j ≤ nn + 1, j++, aa⟦k,j⟧ = temp * aa⟦k,j⟧];
   For[j = k + 1, j ≤ nn + 1, j++,
    temp = aa⟦k,j⟧;
    For[i = k + 1, i ≤ nn, i++,
     aa⟦i,j⟧ = aa⟦i,j⟧ - aa⟦i,k⟧ * temp]]];
  (* Backsolve *)
  For[j = nn, j > 1, j--,
   temp = aa⟦j,nn+1⟧;
   For[i = j - 1, i ≥ 1, i--,
    aa⟦i,nn+1⟧ = aa⟦i,nn+1⟧ - aa⟦i,j⟧ * temp]];
  aa⟦1;;nn,nn+1⟧]
```

We should get a list of rational numbers (floats) that are pretty close to 1. We also compute the average deviation of the answer from all 1s:

```
xf = linsolverf[a, b]
Print["Average float error: ", N[Sum[Abs[xf - 1][[i]], {i, 1, n}] / n]]
```

$$\left\{\frac{2029}{2048}, \frac{1919}{2048}, \frac{245}{256}, \frac{571}{512}, \frac{2041}{2048}, \frac{271}{256}, \frac{551}{512}, \frac{1047}{1024}, \frac{1999}{2048}, \frac{1829}{2048}, \frac{241}{256}, \frac{267}{256}, \frac{1943}{2048},\right.$$
$$\frac{1979}{2048}, \frac{1969}{2048}, \frac{271}{256}, \frac{1017}{1024}, \frac{847}{1024}, \frac{541}{512}, \frac{1975}{2048}, \frac{573}{512}, \frac{2017}{2048}, \frac{15}{16}, \frac{2035}{2048}, \frac{121}{128}, \frac{1033}{1024},$$
$$\frac{65}{64}, \frac{143}{128}, \frac{515}{512}, \frac{957}{1024}, \frac{1061}{1024}, \frac{1941}{2048}, \frac{1061}{1024}, \frac{921}{1024}, \frac{1961}{2048}, \frac{1123}{1024}, \frac{539}{512}, \frac{1087}{1024}, \frac{229}{256},$$
$$\frac{1013}{1024}, \frac{1167}{1024}, \frac{1995}{2048}, \frac{111}{128}, \frac{961}{1024}, \frac{259}{256}, \frac{2019}{2048}, \frac{1845}{2048}, \frac{549}{512}, \frac{1937}{2048}, \frac{491}{512}, \frac{1009}{1024},$$
$$\frac{251}{256}, \frac{1859}{2048}, \frac{2025}{2048}, \frac{239}{256}, \frac{1123}{1024}, \frac{283}{256}, \frac{1635}{2048}, \frac{1083}{1024}, \frac{983}{1024}, \frac{1983}{2048}, \frac{129}{128}, \frac{1075}{1024},$$
$$\frac{517}{512}, \frac{527}{512}, \frac{1051}{1024}, \frac{475}{512}, \frac{529}{512}, \frac{991}{1024}, \frac{1837}{2048}, \frac{517}{512}, \frac{1945}{2048}, \frac{1857}{2048}, \frac{539}{512}, \frac{1045}{1024}, \frac{937}{1024},$$
$$\frac{273}{256}, \frac{2021}{2048}, \frac{1959}{2048}, \frac{1045}{1024}, \frac{1089}{1024}, \frac{545}{512}, \frac{2025}{2048}, \frac{1885}{2048}, \frac{2015}{2048}, \frac{1033}{1024}, \frac{1119}{1024}, \frac{1005}{1024},$$
$$\left.\frac{1011}{1024}, \frac{535}{512}, \frac{35}{32}, \frac{1877}{2048}, \frac{1055}{1024}, \frac{939}{1024}, \frac{479}{512}, \frac{133}{128}, \frac{123}{128}, \frac{953}{1024}, \frac{1029}{1024}, \frac{1973}{2048}\right\}$$

```
Average float error: 0.0530811
```

Half-precision IEEE floats have decimal accuracy that wobbles between about 3 and 3.3 decimals, but we wound up with only about 1 decimal of accuracy (worst case). The posit version is identical to the float version, for comparison purposes, and also includes the fused multiply-add. (It is possible to rearrange the loops so that the innermost loop is a dot product, and thus get another order **n** reduction in the number of roundings undergone by each result value.)

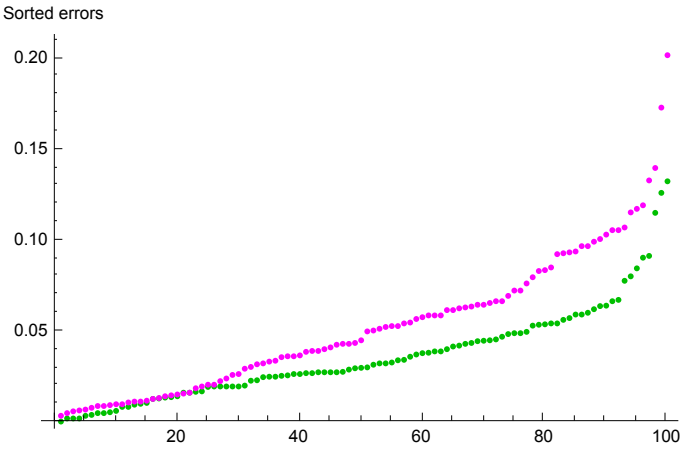
```
linsolverp[a_, b_] := Module[{aa = Transpose[Join[Transpose[a], {b}]],
   piv, i, j, k, nn = Length[b], temp},
  For[k = 1, k ≤ nn, k++,
   piv = k - 1 + (Position[Abs[aa⟦k;;nn,k⟧], Max[Abs[aa⟦k;;nn,k⟧]]])⟦1,1⟧;

   For[j = 1, j ≤ nn + 1, j++, temp = aa⟦piv,j⟧;
    aa⟦piv,j⟧ = aa⟦k,j⟧;
    aa⟦k,j⟧ = temp];
   temp = N[1 / aa⟦k,k⟧];
   For[j = k, j ≤ nn + 1, j++, aa⟦k,j⟧ = N[temp * aa⟦k,j⟧]];
   For[j = k + 1, j ≤ nn + 1, j++,
    temp = aa⟦k,j⟧;
    For[i = k + 1, i ≤ nn, i++,
     aa⟦i,j⟧ = N[aa⟦i,j⟧ - aa⟦i,k⟧ * temp]]];
  (* Backsolve *)
  For[j = nn, j > 1, j--,
   temp = aa⟦j,nn+1⟧;
   For[i = j - 1, i ≥ 1, i--,
    aa⟦i,nn+1⟧ = N[aa⟦i,nn+1⟧ - aa⟦i,j⟧ * temp]];
  aa⟦1;;nn,nn+1⟧]
```

Again we can produce the list of 100 output values, rational numbers close to 1, and find their average deviation from 1. The average float error is about 50 percent greater than the average posit error.

```
xp = linsolverp[a, b]
Print["Average posit error: ", N[Sum[Abs[xp - 1]⟦i⟧, {i, 1, n}] / n]]
ListPlot[{Sort[Abs[xp - 1]], Sort[Abs[xf - 1]]}, PlotRange → {All, All},
 PlotStyle → {■, ■}, AxesLabel → {"", "Sorted errors"}]
```

$$\left\{\frac{2015}{2048}, \frac{2137}{2048}, \frac{1057}{1024}, \frac{1875}{2048}, \frac{4027}{4096}, \frac{3973}{4096}, \frac{2031}{2048}, \frac{1029}{1024}, \frac{4017}{4096}, \frac{273}{256}, \frac{1981}{2048},\right.$$
$$\frac{4057}{4096}, \frac{263}{256}, \frac{553}{512}, \frac{2141}{2048}, \frac{989}{1024}, \frac{1989}{2048}, \frac{1153}{1024}, \frac{3835}{4096}, \frac{511}{512}, \frac{3937}{4096}, \frac{1037}{1024},$$
$$\frac{2147}{2048}, \frac{1019}{1024}, \frac{985}{1024}, \frac{2095}{2048}, \frac{1993}{2048}, \frac{245}{256}, \frac{537}{512}, \frac{1067}{1024}, \frac{519}{512}, \frac{3913}{4096}, \frac{1933}{2048},$$
$$\frac{2089}{2048}, \frac{4055}{4096}, \frac{969}{1024}, \frac{3991}{4096}, \frac{3937}{4096}, \frac{2109}{2048}, \frac{1041}{1024}, \frac{3625}{4096}, \frac{2139}{2048}, \frac{2235}{2048}, \frac{1051}{1024},$$
$$\frac{4095}{4096}, \frac{261}{256}, \frac{2179}{2048}, \frac{499}{512}, \frac{3941}{4096}, \frac{4071}{4096}, \frac{4041}{4096}, \frac{263}{256}, \frac{2065}{2048}, \frac{1931}{2048}, \frac{2207}{2048},$$
$$\frac{3821}{4096}, \frac{493}{512}, \frac{2319}{2048}, \frac{513}{512}, \frac{1065}{1024}, \frac{261}{256}, \frac{3957}{4096}, \frac{263}{256}, \frac{261}{256}, \frac{3983}{4096}, \frac{3877}{4096},$$
$$\frac{2099}{2048}, \frac{991}{1024}, \frac{2037}{2048}, \frac{2171}{2048}, \frac{4003}{4096}, \frac{1051}{1024}, \frac{2175}{2048}, \frac{3893}{4096}, \frac{2103}{2048}, \frac{261}{256}, \frac{1937}{2048},$$
$$\frac{3995}{4096}, \frac{2099}{2048}, \frac{3879}{4096}, \frac{2019}{2048}, \frac{4053}{4096}, \frac{3967}{4096}, \frac{535}{512}, \frac{4043}{4096}, \frac{537}{512}, \frac{1937}{2048}, \frac{1963}{2048},$$
$$\left.\frac{2015}{2048}, \frac{511}{512}, \frac{251}{256}, \frac{67}{64}, \frac{2055}{2048}, \frac{2233}{2048}, \frac{2169}{2048}, \frac{257}{256}, \frac{2169}{2048}, \frac{263}{256}, \frac{3949}{4096}, \frac{3975}{4096}\right\}$$

```
Average posit error: 0.0361914
```



Using 16-bit numbers for a matrix as large as 100-by-100 is clearly risky, since we only have about three decimals of accuracy but need to accumulate sums of up to 100 numbers. We can compute the *residual*, which is the vector of discrepancies $b - Ax$. In the case of posits, the residual can be calculated with the quire, rounding only once. For floats, it needs to be spelled out with a matrix-vector multiply loop using fused multiply-adds.

```
rp = b - a.xp;
rf = Table[1, n];
For[i = 1, i ≤ n, i++, s = 0;
  For[j = 1, j ≤ n, j++,
   s = s + a[[i,j]] × xf[[j]]];
  rf[[i]] = b[[i]] - s];
```

If we now solve for the residual on the right-hand side, we can use the solutions to correct the result. This is really a form of Newton-Raphson iteration. Incidentally, we're being lazy here by calling the solver again, since the solver factored the matrix into lower-upper form and all we have to do is use that to backsolve, an order n^2 amount of work instead of order n^3 to do a linear solve from scratch.

```
x1p = linsolverp[a, rp];
x1f = linsolverf[a, rf];
```

The corrected posit form is now quite close, with several cases of hitting 1 exactly in the answer vector.

```
xp = xp + x1p
```

$$\left\{\frac{1025}{1024}, \frac{1023}{1024}, \frac{2049}{2048}, \frac{257}{256}, \frac{2049}{2048}, \frac{1025}{1024}, \frac{2053}{2048}, \frac{2051}{2048}, \frac{2049}{2048}, \frac{4085}{4096}, \frac{4093}{4096}, 1, \frac{4093}{4096},\right.$$
$$\frac{4093}{4096}, \frac{4089}{4096}, \frac{4095}{4096}, \frac{2049}{2048}, \frac{2041}{2048}, \frac{1025}{1024}, \frac{4091}{4096}, \frac{513}{512}, \frac{4095}{4096}, \frac{511}{512}, 1, \frac{1023}{1024},$$
$$\frac{2047}{2048}, \frac{1025}{1024}, \frac{2053}{2048}, \frac{4095}{4096}, \frac{2047}{2048}, \frac{2049}{2048}, \frac{2047}{2048}, \frac{1025}{1024}, \frac{2043}{2048}, \frac{1023}{1024}, \frac{2053}{2048}, \frac{2051}{2048},$$
$$\frac{1027}{1024}, \frac{4091}{4096}, \frac{4095}{4096}, \frac{2053}{2048}, \frac{4089}{4096}, \frac{2041}{2048}, \frac{2045}{2048}, \frac{2047}{2048}, 1, \frac{2043}{2048}, \frac{1025}{1024}, \frac{2047}{2048},$$
$$\frac{2047}{2048}, \frac{2049}{2048}, \frac{4093}{4096}, \frac{4089}{4096}, \frac{4095}{4096}, \frac{4087}{4096}, \frac{513}{512}, \frac{2053}{2048}, \frac{2039}{2048}, \frac{2049}{2048}, \frac{511}{512}, \frac{2047}{2048},$$
$$1, \frac{1025}{1024}, 1, \frac{2049}{2048}, \frac{2049}{2048}, \frac{4089}{4096}, \frac{4095}{4096}, \frac{4093}{4096}, \frac{1023}{1024}, \frac{2049}{2048}, \frac{2047}{2048}, \frac{4087}{4096}, \frac{1025}{1024},$$
$$1, \frac{4095}{4096}, \frac{513}{512}, \frac{4095}{4096}, \frac{2047}{2048}, \frac{1025}{1024}, \frac{2049}{2048}, \frac{1025}{1024}, 1, \frac{2045}{2048}, \frac{4095}{4096}, \frac{1023}{1024}, \frac{2051}{2048},$$
$$\left.1, \frac{2049}{2048}, 1, \frac{2053}{2048}, \frac{2045}{2048}, \frac{4095}{4096}, \frac{4085}{4096}, \frac{2045}{2048}, \frac{1025}{1024}, \frac{4093}{4096}, \frac{4089}{4096}, \frac{2047}{2048}, \frac{1025}{1024}\right\}$$

The float residual correction improves only the result only by about a factor of 2, and still never lands exactly on the correct answer, 1:

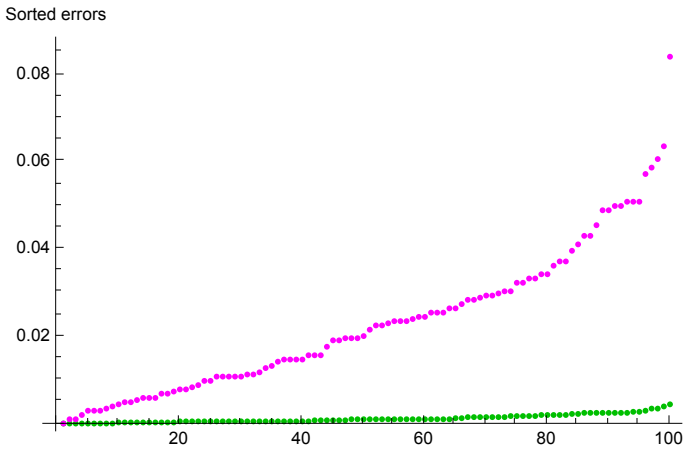
```
xf = xf + x1f
```

$$\left\{\frac{1023}{1024}, \frac{261}{256}, \frac{2037}{2048}, \frac{241}{256}, \frac{1035}{1024}, \frac{1999}{2048}, \frac{1989}{2048}, \frac{989}{1024}, \frac{2031}{2048}, \frac{1075}{1024}, \frac{1039}{1024}, \frac{2039}{2048}, \frac{525}{512},\right.$$
$$\frac{1039}{1024}, \frac{263}{256}, \frac{1013}{1024}, \frac{1011}{1024}, \frac{1089}{1024}, \frac{993}{1024}, \frac{529}{512}, \frac{1955}{2048}, \frac{1023}{1024}, \frac{533}{512}, \frac{259}{256}, \frac{1049}{1024},$$
$$1, \frac{2025}{2048}, \frac{1931}{2048}, \frac{509}{512}, \frac{131}{128}, \frac{2033}{2048}, \frac{525}{512}, \frac{63}{64}, \frac{267}{256}, \frac{517}{512}, \frac{1967}{2048}, \frac{995}{1024}, \frac{1987}{2048},$$
$$\frac{1053}{1024}, \frac{1027}{1024}, \frac{243}{256}, \frac{521}{512}, \frac{543}{512}, \frac{1057}{1024}, \frac{509}{512}, \frac{1027}{1024}, \frac{527}{512}, \frac{2001}{2048}, \frac{1035}{1024}, \frac{65}{64},$$
$$\frac{511}{512}, \frac{1051}{1024}, \frac{529}{512}, \frac{129}{128}, \frac{527}{512}, \frac{243}{256}, \frac{487}{512}, \frac{555}{512}, \frac{2019}{2048}, \frac{1051}{1024}, \frac{261}{256}, \frac{507}{512}, \frac{251}{256},$$
$$\frac{1027}{1024}, \frac{1015}{1024}, \frac{2009}{2048}, \frac{531}{512}, \frac{1031}{1024}, \frac{1039}{1024}, \frac{269}{256}, \frac{501}{512}, \frac{1039}{1024}, \frac{1059}{1024}, \frac{1001}{1024}, \frac{1017}{1024},$$
$$\frac{1061}{1024}, \frac{499}{512}, \frac{1029}{1024}, \frac{65}{64}, \frac{2041}{2048}, \frac{125}{128}, \frac{999}{1024}, \frac{257}{256}, \frac{267}{256}, \frac{1019}{1024}, \frac{129}{128}, \frac{493}{512}, \frac{1035}{1024},$$
$$\left.\frac{515}{512}, \frac{2009}{2048}, \frac{973}{1024}, \frac{1055}{1024}, \frac{1013}{1024}, \frac{537}{512}, \frac{1057}{1024}, \frac{2007}{2048}, \frac{1047}{1024}, \frac{131}{128}, \frac{2025}{2048}, \frac{2021}{2048}\right\}$$

Here are the average deviations from an all-1s correct answer, and plots of the errors, sorted:

```
Print["Average float error: ", N[Sum[Abs[xf - 1][[i]], {i, 1, n}] / n]];
Print["Average posit error: ", N[Sum[Abs[xp - 1][[i]], {i, 1, n}] / n]];
ListPlot[{Sort[Abs[xp - 1]], Sort[Abs[xf - 1]]}, PlotRange → {All, All},
 PlotStyle → {■, ■}, AxesLabel → {"", "Sorted errors"}]
```

```
Average float error: 0.0231592
Average posit error: 0.00112549
```



Another iteration does not improve the float result. But look what it does to the posit result; it nails it!

```
rp = b - a.xp;
x1p = linsolverp[a, rp];
xp = xp + x1p
```

```
{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
```

The posit answer is *infinitely accurate*. The correction technique was of limited value for floats because you really need a quire to do residual correction properly. Since it only improves the answer slightly, most people never bother with more than one iteration of Gaussian elimination, and refer to it as a “direct method.” But it is really nothing of the kind; it is a guessing method that usually progresses toward the solution.

While we’re at it, we can reproduce the original LINPACK benchmark, using 64-bit IEEE floats and the same 100-by-100 matrix. Let’s see how well that works, where we display the answer in 6-point type to get a compact display

```
setfloatenv[{64, 11}]
xf = linsolverf[a, b]; Style[xf, 6]
```

```
{9 007 199 254 740 915/9 007 199 254 740 992, 1 125 899 906 842 643/1 125 899 906 842 624, 1, 4 503 599 627 370 305/4 503 599 627 370 496, 9 007 199 254 740 933/9 007 199 254 740 992, 2 251 799 813 685 211/2 251 799 813 685 248, 9 007 199 254 740 829/9 007 199 254 740 992, 9 007 199 254 740 901/9 007 199 254 740 992, 2 251 799 813 685 219/2 251 799 813 685 248,
 2 251 799 813 685 315/2 251 799 813 685 248, 4 503 599 627 370 423/4 503 599 627 370 496, 9 007 199 254 740 973/9 007 199 254 740 992, 2 251 799 813 685 265/2 251 799 813 685 248, 4 503 599 627 370 557/4 503 599 627 370 496, 4 503 599 627 370 609/4 503 599 627 370 496, 4 503 599 627 370 487/4 503 599 627 370 496, 2 251 799 813 685 223/2 251 799 813 685 248,
 2 251 799 813 685 361/2 251 799 813 685 248, 4 503 599 627 370 363/4 503 599 627 370 496, 2 251 799 813 685 265/2 251 799 813 685 248, 9 007 199 254 740 805/9 007 199 254 740 992, 4 503 599 627 370 531/4 503 599 627 370 496, 4 503 599 627 370 601/4 503 599 627 370 496, 1 125 899 906 842 629/1 125 899 906 842 624, 4 503 599 627 370 517/4 503 599 627 370 496,
 4 503 599 627 370 517/4 503 599 627 370 496, 2 251 799 813 685 221/2 251 799 813 685 248, 4 503 599 627 370 351/4 503 599 627 370 496, 4 503 599 627 370 543/4 503 599 627 370 496, 9 007 199 254 740 991/9 007 199 254 740 992, 4 503 599 627 370 481/4 503 599 627 370 496, 1 125 899 906 842 625/1 125 899 906 842 624, 4 503 599 627 370 471/4 503 599 627 370 496,
 4 503 599 627 370 627/4 503 599 627 370 496, 2 251 799 813 685 261/2 251 799 813 685 248, 9 007 199 254 740 715/9 007 199 254 740 992, 9 007 199 254 740 779/9 007 199 254 740 992, 9 007 199 254 740 825/9 007 199 254 740 992, 4 503 599 627 370 549/4 503 599 627 370 496, 9 007 199 254 740 981/9 007 199 254 740 992, 2 251 799 813 685 181/2 251 799 813 685 248,
 1 125 899 906 842 641/1 125 899 906 842 624, 1 125 899 906 842 671/1 125 899 906 842 624, 281 474 976 710 659/281 474 976 710 656, 2 251 799 813 685 265/2 251 799 813 685 248, 562 949 953 421 317/562 949 953 421 312, 4 503 599 627 370 601/4 503 599 627 370 496, 9 007 199 254 740 923/9 007 199 254 740 992, 9 007 199 254 740 951/9 007 199 254 740 992,
 9 007 199 254 740 973/9 007 199 254 740 992, 9 007 199 254 740 921/9 007 199 254 740 992, 562 949 953 421 321/562 949 953 421 312, 2 251 799 813 685 273/2 251 799 813 685 248, 4 503 599 627 370 441/4 503 599 627 370 496, 2 251 799 813 685 299/2 251 799 813 685 248, 4 503 599 627 370 403/4 503 599 627 370 496, 4 503 599 627 370 401/4 503 599 627 370 496,
 4 503 599 627 370 737/4 503 599 627 370 496, 1 125 899 906 842 615/1 125 899 906 842 624, 4 503 599 627 370 601/4 503 599 627 370 496, 4 503 599 627 370 557/4 503 599 627 370 496, 9 007 199 254 740 931/9 007 199 254 740 992, 4 503 599 627 370 519/4 503 599 627 370 496, 9 007 199 254 740 981/9 007 199 254 740 992,
 2 251 799 813 685 223/2 251 799 813 685 248, 4 503 599 627 370 411/4 503 599 627 370 496, 4 503 599 627 370 593/4 503 599 627 370 496, 9 007 199 254 740 981/9 007 199 254 740 992, 4 503 599 627 370 535/4 503 599 627 370 496, 2 251 799 813 685 313/2 251 799 813 685 248, 9 007 199 254 740 949/9 007 199 254 740 992,
 4 503 599 627 370 531/4 503 599 627 370 496, 4 503 599 627 370 629/4 503 599 627 370 496, 9 007 199 254 740 881/9 007 199 254 740 992, 2 251 799 813 685 255/2 251 799 813 685 248, 4 503 599 627 370 525/4 503 599 627 370 496, 4 503 599 627 370 389/4 503 599 627 370 496, 9 007 199 254 740 935/9 007 199 254 740 992,
 4 503 599 627 370 499/4 503 599 627 370 496, 9 007 199 254 740 861/9 007 199 254 740 992, 9 007 199 254 740 901/9 007 199 254 740 992, 9 007 199 254 740 941/9 007 199 254 740 992, 1 125 899 906 842 601/1 125 899 906 842 624, 4 503 599 627 370 625/4 503 599 627 370 496, 4 503 599 627 370 479/4 503 599 627 370 496,
 2 251 799 813 685 291/2 251 799 813 685 248, 9 007 199 254 740 753/9 007 199 254 740 992, 9 007 199 254 740 939/9 007 199 254 740 992, 1 125 899 906 842 621/1 125 899 906 842 624, 4 503 599 627 370 491/4 503 599 627 370 496, 9 007 199 254 740 773/9 007 199 254 740 992, 2 251 799 813 685 289/2 251 799 813 685 248,
 9 007 199 254 740 983/9 007 199 254 740 992, 4 503 599 627 370 677/4 503 599 627 370 496, 2 251 799 813 685 307/2 251 799 813 685 248, 4 503 599 627 370 485/4 503 599 627 370 496, 1 125 899 906 842 643/1 125 899 906 842 624, 4 503 599 627 370 563/4 503 599 627 370 496, 4 503 599 627 370 497/4 503 599 627 370 496, 140 737 488 355 325/140 737 488 355 328}
```

The answer is close, but misses the correct value, 1, for *every single entry*. To make it easier to read, here are the first few entries of the answer vector expressed as decimals, to 54 digits:

```
Table[N[xf[[i]], 54], {i, 1, 10}] // TableForm
```

```
0.999999999999991451282710386294638738036155700683593750
1.000000000000016875389974302379414439201354980468750000
1.000000000000000000000000000000000000000000000000000000
0.999999999999957589480459319020155817270278930664062500
0.999999999999993449684154711576411500573152641601562 5
0.999999999999983568699235547683201730251312255859375000
0.999999999999981903364698609948391094803810119628906250
0.999999999999989896970475911075482144951820373535156250
0.999999999999987121412914348184131085872650146484375000
1.000000000000029753977059954195283353328704833984375 00
```

The choice seems pretty clear; use 64-bit floats and get the wrong answer for every single one of the entries in the answer vector, or use *16-bit posits* and get the correct answer for every single entry. Despite the use of the residual correction, the posits would almost certainly be *much* faster time to solution, and would certainly use only 1/4 the storage while getting there.

The limit of using 16-bit posits is that they do run out of dynamic range for larger n, when the matrices are random elements like this. If the matrix is well-posed, as often is true for the linear systems needed for actual physics simulations, then 16-bit posits should be quite capable of producing answers that are reliable to about three decimals of accuracy, and that's often sufficient for scientific and engineering purposes.

12.2 Bailey's Numerical Nightmare, Redux

Recall this example from *The End of Error*: A 2-by-2 linear system David Bailey came up with whose condition number is about as bad as it gets. The matrix is only one ULP away from being singular! He posed the problem as

$$\begin{pmatrix} 0.25510582 & 0.52746197 \\ 0.80143857 & 1.65707065 \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0.79981812 \\ 2.51270273 \end{pmatrix}$$

The answer is $x = -1$, $y = 2$. Even though the values are only 8- and 9-decimals, you get into trouble using double precision floats that are capable of about 16 decimals of accuracy. Part of Bailey's point was that there is rounding error in converting the decimals to binary representation, which is magnified by the ill-conditioning of the matrix. Here is a way to express the problem that removes the rounding error from converting decimals to binary, where we use variables $\begin{pmatrix} a & b \\ c & d \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} u \\ v \end{pmatrix}$ to represent the system:

```
{{a, b}, {c, d}} = {{25 510 582, 52 746 197}, {80 143 857, 165 707 065}} / 2^26;
{u, v} = {79 981 812, 251 270 273} / 2^26;
```

All these input values are now exactly expressible with double-precision IEEE floats. *Mathematica* can confirm the exact answer:

```
LinearSolve[{{a, b}, {c, d}}, {u, v}]
```

```
{-1, 2}
```

However, that is still not enough to save floats from terrible rounding errors. Since the system is only 2-by-2, we can write the solution in compact form using Cramer's rule:

```
setfloatenv[{64, 11}];
det = a d - b c;
{x, y} = {(u d - b v) / det, (a v - u c) / det}
```

```
{0, 2}
```

Whoops. While the determinate is evaluated correctly at this precision, $u\,d - b\,v$ is not, resulting in an infinite decimal error underflow in the value for x. If we use 64-bit posits ($es = 3$), we get the correct value (−1, 2). But posits have accuracy to spare for such a problem; we can turn the *nbits* value all the way down to 59 bits and still get the correct result:

```
setpositenv[{59, 3}];
det = a d - b c;
{x, y} = {(u d - b v) / det, (a v - u c) / det}
```

```
{-1, 2}
```

Check the condition number of this matrix to see why "Bailey's Numerical Nightmare" is a good nickname for the test case:

```
Eigenvalues[{{a, b}, {c, d}}]
Print["Condition number is ", N[%[[1]] / %[[2]]]]
```

```
{(191 217 647 + 21 √82 911 992 118 405) / 134 217 728,
 1 / (33 554 432 (191 217 647 + 21 √82 911 992 118 405))}
```

```
Condition number is 3.65642 × 10^16
```

Well-posed systems have a condition number of magnitude near 1. Even though the input values (and the correct answer) all have magnitude near 1, you can still have a terrible condition number that produces wildly wrong results. That can still happen with posits, but posits do provide a little more protection than floats do! Here's what happens if we ask Mathematica to solve the system expressed using decimal inputs as Bailey intended:

```
{{a, b}, {c, d}} = {{0.25510582, 0.52746197}, {0.80143857, 1.65707065}};
{u, v} = {0.79981812, 2.51270273};
LinearSolve[{{a, b}, {c, d}}, {u, v}]
```

LinearSolve: Result for LinearSolve of badly conditioned matrix {{0.255106, 0.527462}, {0.801439, 1.65707}} may contain significant numerical errors.

```
{0.272534, 1.38454}
```

12.4 The XSC approach

12.4.1 An interval arithmetic stand-in for a valid environment

In the 1980s, a group of mathematicians at University of Karlsruhe, including Kulisch and S. Rump and others, developed ways to use quire-type refinement of linear solvers to improve a wide range of operations, not just solutions to Ax = *b*. For example, they demonstrated evaluation of polynomials, rational functions, and numerical derivatives to within 0.5 ULP. The key idea is to write such calculations as a sequence of operations depending on + – × ÷ operations on previously-computed values, which can then be rewritten as a sparse system of equations that is lower-triangular. That in turn can be solved to very high precision using an exact dot product. While languages like PASCAL-XSC and ACRITH use the approach, the lack of a standard for a quire (and the availability of fast hardware supporting it) has inhibited their wide adoption. The posit environment can make use of the techniques developed by the Karlsruhe group.

Much of their approach depends on interval bounds, which makes us wish for a complete valid environment. It's not like a full environment for valids, but for now we can lean on the built-in interval arithmetic in *Mathematica*. We'll need this later for iterative refinement that uses the quire. This should be regarded as an *unfinished* part of the prototype environment, for now. The **x2pint** function takes a real value or an interval with real endpoints and returns an interval that has posit endpoints and encloses the input argument.

```
x2pint[x_] := Module[{e, f, hi, i, lo, p,
    pinf = BitShiftLeft[1, nbits - 1], xlo = Min[x], xhi = Max[x], y},
   {e, y} = {2^(es-1), Abs[xlo]};
   lo = Which[(* First, take care of the exception values: *)
     0 ≤ xlo < minpos, 0,
     xlo < -maxpos ⋁ y == ∞, pinf, (* ±∞; 1 followed by all 0 bits *)
     -minpos < xlo < 0, Mod[-1, npat], (* -minpos; all 1 bits *)
     maxpos < xlo, pinf - 1, (* maxpos; 0 followed by all 1 bits *)
     True, If[y ≥ 1, (* Northeast quadrant: *)
      p = 1;
      i = 2; (* Shift in 1s from the right and scale down. *)
      While[y ≥ useed ⋀ i < nbits, {p, y, i} = {2 p + 1, y / useed, i + 1}];
      p = 2 p; i++, (* Else, southeast quadrant: *)
      p = 0;
      i = 1; (* Shift in 0s from the right and scale up. *)
      While[y < 1 ⋀ i ≤ nbits, {y, i} = {y * useed, i + 1}];
      If[i ≥ nbits, p = 2; i = nbits + 1,
       p = 1; i++]]; (* Extract exponent bits: *)
```

```
    While[e > 1 / 2 ⋀ i ≤ nbits, p = 2 p;
     If[y ≥ 2^e, y /= 2^e;
      p++];
     e /= 2; i++];
    y--; (* Fraction bits; subtract the hidden bit *)
    While[y > 0 ⋀ i ≤ nbits, y = 2 y; p = 2 p + ⌊y⌋; y -= ⌊y⌋; i++];
    p *= 2^(nbits+1-i);
    i++;
    (* For low bound, round up if x is negative, down if positive *)
    i = BitAnd[p, 1]; p = ⌊p / 2⌋;
    p = Which[
      i == 0 ⋀ (y == 1 ⋁ y == 0), p, (* If exact, leave it alone *)
      xlo > 0, p, (* inexact and x > 0, so truncate. *)
      True, p + 1 (* inexact and x < 0, so round up *)];
    Mod[If[xlo < 0, npat - p, p], npat (* Simulate 2's complement *)]
   ];

  {e, y} = {2^(es-1), Abs[xhi]};
  hi = Which[(* First, take care of the exception values: *)
    -minpos < xhi ≤ 0, 0,
    maxpos < xhi ⋁ y == ∞, pinf,
    0 < xhi ≤ minpos, 1, (* minpos; all 0 bits except last bit = 1 *)
    xhi ≤ -maxpos, pinf + 1, (*-maxpos; 1000...0001 *)
    True, If[y ≥ 1, (* Northeast quadrant: *)
     p = 1;
     i = 2; (* Shift in 1s from the right and scale down. *)
     While[y ≥ useed ⋀ i < nbits, {p, y, i} = {2 p + 1, y / useed, i + 1}];
     p = 2 p; i++, (* Else, southeast quadrant: *)
     p = 0;
     i = 1; (* Shift in 0s from the right and scale up. *)
     While[y < 1 ⋀ i ≤ nbits, {y, i} = {y * useed, i + 1}];
     If[i ≥ nbits, p = 2;
      i = nbits + 1, p = 1;
      i++]];
    (* Extract exponent bits: *)
    While[e > 1 / 2 ⋀ i ≤ nbits, p = 2 p;
     If[y ≥ 2^e, y /= 2^e;
      p++];
     e /= 2; i++];
    y--; (* Fraction bits; subtract the hidden bit *)
    While[y > 0 ⋀ i ≤ nbits, y = 2 y; p = 2 p + ⌊y⌋; y -= ⌊y⌋; i++];
    p *= 2^(nbits+1-i);
    i++;
    (* For high bound, round down if x is negative, up if positive *)
    i = BitAnd[p, 1]; p = ⌊p / 2⌋;
    p = Which[
      i == 0 ⋀ (y == 1 ⋁ y == 0), p, (* If exact, leave it alone *)
      xhi < 0, p, (* inexact and x < 0, so truncate. *)
      True, p + 1 (* inexact and x > 0, so round up *)];
    Mod[If[xhi < 0, npat - p, p], npat (* Simulate 2's complement *)]
   ];
  Interval[{p2x[lo], p2x[hi]}]
 ];
SetAttributes[x2pint, Listable]
```

Here's an example of how it finds a bound on the value of π, using a pair of posits as endpoints of a closed interval (a subset of what valids can represent):

```
setpositenv[{16, 1}]
x2pint[π]
N[%, 15]
```

$$\text{Interval}\left[\left\{\frac{6433}{2048}, \frac{3217}{1024}\right\}\right]$$

```
Interval[{3.14111328125000, 3.14160156250000}]
```

12.4.2 The XSC-style solver for a basic block

Kulisch & Miranker showed that a basic block with only plus-minus-times-divide operations can be converted to a lower triangular system of equations, and then solved to within 0.5 ULP using the exact dot product. One of their examples:

$$\text{Find } X = (a+b)\cdot c - \frac{d}{e}.$$

As a linear algebra problem, this expression is equivalent to solving six linear equations in six unknowns:

$$\begin{aligned} x_1 &= a \\ x_2 &= x_1 + b \\ x_3 &= c\, x_2 \\ x_4 &= d \\ e\, x_5 &= x_4 \\ x_6 &= x_3 - x_5 \end{aligned}$$

which becomes the system

$$\begin{pmatrix} 1 & & & & & \\ -1 & 1 & & & & \\ & c & -1 & & & \\ & & & 1 & & \\ & & & -1 & e & \\ & & -1 & 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{pmatrix} = \begin{pmatrix} a \\ b \\ 0 \\ d \\ 0 \\ 0 \end{pmatrix}.$$

and the X we seek is x_6. The matrix is lower triangular and sparse. Now let's see if we can pick posit values in {16, 1} that lead to rounding error, and use *Mathematica* to solve the system exactly as a basis for comparison:

```
setpositenv[{16, 1}];
A = {{1, 0, 0, 0, 0, 0}, {-1, 1, 0, 0, 0, 0}, {0, c, -1, 0, 0, 0},
    {0, 0, 0, 1, 0, 0}, {0, 0, 0, -1, e, 0}, {0, 0, -1, 0, 1, 1}};
B = {a, b, 0, d, 0, 0};
{a, b, c, d, e} = {32, 3 / 64, 1023 / 32 768, 3329 / 256, 13};
xexact = LinearSolve[A, B]
N[xexact, 9]
```

```
{0.255106, 0.782568, 0.62718, 1.65707, 0.127467, 0.499713}
```

```
{0.255106, 0.782568, 0.62718, 1.65707, 0.127467, 0.499713}
```

Here are the nearest posits to each of those values:

```
goal = xexact
N[goal, 8]
```

$$\left\{\frac{1045}{4096}, \frac{6411}{8192}, \frac{2569}{4096}, \frac{6787}{4096}, \frac{261}{2048}, \frac{8187}{16\,384}\right\}$$

```
{0.25512695, 0.78259277, 0.62719727, 1.6569824, 0.12744141, 0.49969482}
```

The input values were selected to cause trouble, starting with $32 + \frac{3}{64}$ which rounds up to $32\frac{1}{16}$ in this low-precision environment. The *c* value is selected to make the product approximately 1, and then adding a ratio (which also rounds) close to –1 magnifies the relative error. Now contrast the exact value with the computed value we get if we round after every operation:

```
(a + b) * c - d / e
N[%]
```

$$\frac{3}{4096}$$

```
0.000732422
```

That's a lot of rounding error for so few operations. Good! Let's see if we can fix it. Use the posit linear solver, and then compute the residual with quire-level precision (with a final rounding to the nearest posit):

```
x = linsolverp[A, B]
res1 = B - A.x
N[res1]
```

$$\left\{\frac{1045}{4096}, \frac{6411}{8192}, \frac{2569}{4096}, \frac{6787}{4096}, \frac{2089}{16\,384}, \frac{8187}{16\,384}\right\}$$

$$\left\{-\frac{11}{524\,288}, -\frac{1}{262\,144}, -\frac{3}{1\,048\,576}, \frac{93}{1\,048\,576}, -\frac{9}{16\,384}, 0\right\}$$

$$\{-0.0000209808, -3.8147 \times 10^{-6}, -2.86102 \times 10^{-6}, 0.0000886917, -0.000549316, 0.\}$$

Now solve using the residual as the right-hand side, add it as a correction to the first solution, and express that solution as posits:

```
adjust1 = linsolverp[A, res1];
x1 = x + adjust1
```

$$\left\{\frac{1045}{4096}, \frac{6411}{8192}, \frac{2569}{4096}, \frac{6787}{4096}, \frac{261}{2048}, \frac{8187}{16\,384}\right\}$$

Notice that this agrees with all six x_i values in **goal**:

```
goal - x1
```

$$\{0, 0, 0, 0, 0, 0\}$$

This is why quires and linear solvers are about a lot more than just A $x = b$ problems; they are the key to evaluating basic block sequences of + – × ÷ operations to within half an ULP, as if there were only one rounding. The setup of the sparse lower-triangular matrix and the iterative solution can be done automatically, as has been demonstrated since the 1980s with the XSC languages; S. Rump estimates that it increases the total work by a maximum of a factor of six, usually much less. Even if the number of operations goes up by a factor of six, they are using variables in registers and the closest level of cache, not main memory. If the technique allows low-precision arithmetic to get satisfactory results, the savings of memory bandwidth and storage (and the corresponding savings of power and energy costs) should be well worth the extra work, especially if the programmer can specify where to apply the technique and where not to.

12.4.3 Associative Law for Multiplication?

Experiment: See if the quire can get us the associative property of *multiplication*. Use a really low-precision posit environment, and a simple case where $(u \times v) \times w \neq u \times (v \times w)$ after rounding.

```
setpositenv[{8, 0}]
u = 35 / 32; v = 15 / 16; w = 15 / 16;
```

Here is the correct product, as a fraction and as an exact decimal:

```
pex = u v w
N[pex, 13]
```

$$\frac{7875}{8192}$$

```
0.9613037109375
```

Notice that this is not even expressible with a quire, which stores integer multiples of $\frac{1}{4096}$ for the {8, 0} posit environment. Here is the nearest posit to the correct product:

```
pex
N[%]
```

$$\frac{31}{32}$$

```
0.96875
```

This trio of numbers breaks the associative law if we round after each multiplication. Here's what we get if we group the last two terms:

```
u (v w)
N[%]
```

$$\frac{61}{64}$$

```
0.953125
```

If we instead happen to group the first two terms, we get the best-possible answer (the one within 0.5 ULP of the exact product, **pex**):

```
(u v) w
```

$$\frac{31}{32}$$

The incorrect rounding of 61/64 is only *very slightly* farther away from the true solution, so this is a sensitive test case. Half an ULP is 0.0078125 for these values. It's a "squeaker"; the difference between the distances to the exact answer **pex** is only 3/4096:

```
Abs[31 / 32. - pex]
Abs[61 / 64. - pex]
```

```
0.00744629
```

```
0.00817871
```

This shows that floating-point multiplication is not associative in general, when we round after every multiply. That is true whether we use floats or posits. (If we use *valids*, the two ways to compute the product will always produce overlapping valids, and the overlap can be considered a tighter bound on the correct value.)

Can quire-based linear solvers restore multiplicative associativity? Set up the product $u \cdot v \cdot w$ as a lower triangular system. When there are only multiplies, adds, and subtracts in an expression, it is always possible to do this with 1s or –1s on the diagonal.

$$\begin{aligned} x_1 &= u \\ x_2 &= x_1 \times v \\ x_3 &= x_2 \times w \end{aligned}$$

$$\begin{pmatrix} 1 & 0 & 0 \\ v & -1 & 0 \\ 0 & w & -1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} u \\ 0 \\ 0 \end{pmatrix}$$

Solving the system exactly produces the product in x_3. Here is the exact solution, using Kulisch's notations for all variables:

```
A = {{1, 0, 0}, {v, -1, 0}, {0, w, -1}};
b = {u, 0, 0};
n = Length[b];
x̂ = LinearSolve[A, b]
N[x̂, 13]
```

$$\left\{\frac{35}{32}, \frac{525}{512}, \frac{7875}{8192}\right\}$$

```
{1.093750000000, 1.025390625000, 0.9613037109375}
```

Construct an approximate inverse matrix **R** at posit precision:

```
R = Table[\overline{Inverse[A]_[[i,j]]}, {i, 1, n}, {j, 1, n}]; MatrixForm[R]
```

$$\begin{pmatrix} 1 & 0 & 0 \\ \frac{15}{16} & -1 & 0 \\ \frac{7}{8} & -\frac{15}{16} & -1 \end{pmatrix}$$

The refinement matrix is $I - R\,A$. Compute it at quire precision and bound with valids (using Mathe-

matica intervals for now, as stand-ins for an actual valid library):

```
ref = IdentityMatrix[n] - A.R; MatrixForm[ref]
ref = x2pint[ref];
MatrixForm[ref]
```

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ -\frac{1}{256} & 0 & 0 \end{pmatrix}$$

$$\begin{pmatrix} \text{Interval}[\{0, 0\}] & \text{Interval}[\{0, 0\}] & \text{Interval}[\{0, 0\}] \\ \text{Interval}[\{0, 0\}] & \text{Interval}[\{0, 0\}] & \text{Interval}[\{0, 0\}] \\ \text{Interval}\left[\left\{-\frac{1}{64}, 0\right\}\right] & \text{Interval}[\{0, 0\}] & \text{Interval}[\{0, 0\}] \end{pmatrix}$$

We compute $R \cdot b$ in quire precision, then round to posit as the approximate solution, $\tilde{x}$.

```
x̃ = R.b
```

$$\left\{\frac{35}{32}, \frac{33}{32}, \frac{61}{64}\right\}$$

We calculate the residual, or "defect" as the Germans like to call it, using quire precision… and again find the nearest posit form of that.

```
dq = b - A.x̃
d = dq
```

$$\left\{0, \frac{3}{512}, -\frac{7}{512}\right\}$$

$$\left\{0, \frac{1}{64}, -\frac{1}{64}\right\}$$

Now we need an interval enclosure around $\tilde{x} - \hat{x}$. Let's be conservative and give it the entire range of representable real numbers.

```
e = Table[Interval[{-maxpos, maxpos}], {i, 1, n}]
```

```
{Interval[{-64, 64}], Interval[{-64, 64}], Interval[{-64, 64}]}
```

This at last sets up the iteration scheme specified by Kulisch on page 31 of "Arithmetic of the Digital Computer". The provable bound shrinks with each iteration, tightening the noose until the uncertainty is only one ULP wide. More work needs to be done to show that valids can do even better than closed intervals. This is an example where interval-type algorithms are different from conventional iterative methods; they need to be crafted carefully so that they produce a *contrac-*

tive map. Some conventional iterative methods do not do this, and with interval arithmetic, the bounds get looser instead of tighter with each iteration!

Notice, however, that we do *not* require expertise from the user. A compiler can **automatically** generate the refinement process and produce an ULP-wide bound, then rounding it to the nearest posit. A compiler directive should say which code blocks should be insured for high accuracy in this way, so that the program does not run slower because of the refinement of answers that need not be highly accurate. The XSC environments do not have this level of programmer control, another reason they have not caught on.

```
e = x2pint[ref.e + R.d]
```

$$\left\{\text{Interval}[\{0, 0\}], \text{Interval}\left[\left\{-\frac{1}{64}, -\frac{1}{64}\right\}\right], \text{Interval}\left[\left\{-1, \frac{33}{32}\right\}\right]\right\}$$

```
e = x2pint[ref.e + R.d]
```

$$\left\{\text{Interval}[\{0, 0\}], \text{Interval}\left[\left\{-\frac{1}{64}, -\frac{1}{64}\right\}\right], \text{Interval}\left[\left\{0, \frac{1}{64}\right\}\right]\right\}$$

That last element is one ULP wide, so we're done. We could iterate one more time just to show that the enclosure is now stable:

```
e = x2pint[ref.e + R.d]
```

$$\left\{\text{Interval}[\{0, 0\}], \text{Interval}\left[\left\{-\frac{1}{64}, -\frac{1}{64}\right\}\right], \text{Interval}\left[\left\{0, \frac{1}{64}\right\}\right]\right\}$$

It's stable. The corrected answer:

```
x̃ + e
```

$$\left\{\text{Interval}\left[\left\{\frac{35}{32}, \frac{35}{32}\right\}\right], \text{Interval}\left[\left\{\frac{65}{64}, \frac{65}{64}\right\}\right], \text{Interval}\left[\left\{\frac{61}{64}, \frac{31}{32}\right\}\right]\right\}$$

Well, it worked. It found the best interval enclosure of the true product, one ULP wide. In replacing the bound with a posit, we would round to the "even" endpoint 31/32. This shows that the quire can restore the associative property of algebra to a posit computing environment, if needed. If we had used interval arithmetic (valids) to do the multiplication, the bound would be two ULPs wide after the two multiplications. The XSC approach brings that down to one ULP wide, which is then rounded uniquely and consistently.

12.4.4 When even the quire cannot rescue an ill-posed system

From Kulisch & Miranker, an example where having a low residual does *not* mean you are close to the answer. Notice that this is not a lower-triangular matrix; lower-triangular matrices are much

better behaved. This is a lot like Bailey's Numerical Nightmare, a deceptively simple-looking 2-by-2 system that nonetheless is terribly ill-posed.

```
setpositenv[{16, 1}]
A = {{a11, a12}, {a21, a22}} = {{0.780, 0.563}, {0.913, 0.659}};
b = {b1, b2} = {a11 - a12, a21 - a22};
n = Length[b];
Print[MatrixForm[A], MatrixForm[{X, Y}], " = ", MatrixForm[b]]
```

$$\begin{pmatrix} \frac{3195}{4096} & \frac{1153}{2048} \\ \frac{7479}{8192} & \frac{5399}{8192} \end{pmatrix} \begin{pmatrix} X \\ Y \end{pmatrix} = \begin{pmatrix} \frac{889}{4096} \\ \frac{65}{256} \end{pmatrix}$$

The exact answer *should* be {1, −1}, and exact arithmetic using rational numbers is capable of a true *direct* solution since there is no rounding:

```
LinearSolve[A, b]
```

```
{1, -1}
```

To set up the XSC tightening-enclosure iteration, we first find the posit approximation to the inverse matrix.

```
R = Inverse[A]; MatrixForm[R]
```

$$\begin{pmatrix} 6848 & -5824 \\ -9472 & 8128 \end{pmatrix}$$

Apply that approximate inverse to the right-hand side vector *b* to get a starting guess (quire precision):

```
x̃ = R.b
N[x̃]
```

$$\left\{\frac{483}{64}, \frac{127}{16}\right\}$$

```
{7.54688, 7.9375}
```

Those values didn't need rounding to the nearest posit, but happened to be already expressible as posits. Compute the defect using quire precision, and round to the nearest posit:

```
d̃ = b - A.x̃; N[d̃]
```

```
{-10.1367, -11.8672}
```

Notice that that's a pretty large defect! The problem lies in the refinement matrix operator, *ref* = I – A·R:

```
ref = IdentityMatrix[n] - A.R; MatrixForm[ref]
```

$$\begin{pmatrix} -\frac{513}{64} & -\frac{529}{16} \\ -\frac{1201}{128} & -\frac{1239}{32} \end{pmatrix}$$

Unless the *spectral radius* (the ratio of the largest eigenvalue to the smallest eigenvalue) is less than 1, we have no assurance of convergence.

```
N[Eigenvalues[ref]]
```

```
{-46.7315, -0.00290473}
```

The spectral radius is *not* less than 1, so we should not have high hopes for residual refinement. Applying the "refinement" operator will make things worse, that is, the interval bound will be come looser, not tighter.
Residual correction isn't working **even with the quire.** There exist problems so ill-posed that they cannot be solved this way. The XSC workaround is to detect these rare cases and report them to the user.

Furthermore, consider two candidate solutions. One of them is very close to the correct solution, $X = 0.999$, $Y = -1.001$ (expressed using the closest posit)

```
{X, Y} = {0.999, -1.001}; N[{X, Y}]
```

```
{0.999023, -1.00098}
```

Our human intuition says that the defect should be very small for such a close guess:

```
r = b - A.{X, Y}; N[r]
```

```
{0.00131154, 0.00153518}
```

For the second candidate solution, instead try a poorer guess, $X = 0.341$, $Y = -0.087$, and check the residual:

```
{X, Y} = {0.341, -0.087}; N[{X, Y}]
```

```
{0.341003, -0.0870056}
```

```
r = b - A.{X, Y}; N[r]
```

```
{0.0000315011, -0.0000758357}
```

The second choice for the solution vector is clearly farther from the correct solution, yet gives a *smaller residual*! At least it is possible to detect when the refinement method will fail. Having a lucky guess might not help you.

12.4.5 Kulisch polynomial challenge example

Kulisch gives the example of a polynomial that is difficult to evaluate accurately for certain input values. It has an exact root at $1/\sqrt{2}$. If we use a value close to that root, any errors will be magnified.

```
P[t_] := 8118 t^4 - 11 482 t^3 + t^2 + 5741 t - 2030
```

We should probably write out the definition of the polynomial using Horner's rule, the way a program would actually evaluate the polynomial.

```
P[t_] := (((8118 t - 11 482) t + 1) t + 5741) t - 2030
```

Kulisch's example uses $t = 0.707107$, and shows that the polynomial evaluation is correct to only two decimal places. To be fair, we need to find the exact answer using the closest *binary* representation of $\frac{707\,107}{1\,000\,000}$ as the input quantity. With the environment set for 32-bit posits:

```
setpositenv[{32, 2}]; t = 707 107 / 1 000 000
N[t, 27]
```

$$\frac{94\,906\,295}{134\,217\,728}$$

```
0.707107000052928924560546875
```

We can express the exact polynomial of that fraction using exact rational arithmetic and 97 decimals to express the output, though of course only the first few decimals are significant digits:

```
pex = P[t]; N[pex, 97]
```

$$-1.9157388506341908195757433328672682792140480557305329831555007480403673980\backslash$$
$$617895722389221191406\,25 \times 10^{-11}$$

Now look what happens when we rely on "machine precision," that is, internal 64-bit IEEE floats:

```
P[N[t]]
```

-1.93268×10^{-11}

Despite using more than 15 significant decimals, only the first two decimals are correct: $-1.9326\cdots \times 10^{-11}$. **Can we get a better answer with 32-bit posits, using XSC-style refinement techniques?**

Write Horner's rule for $P(t)$ using five temporary quantities, x_1 through x_5:

$$\begin{aligned} x_1 &= 8118 \\ x_2 &= x_1\, t - 11\,482 \\ x_3 &= x_2\, t + 1 \\ x_4 &= x_3\, t + 5741 \\ x_5 &= x_4\, t - 2030 \end{aligned} \quad \text{which becomes the system} \quad \begin{pmatrix} 1 & & & & \\ -t & 1 & & & \\ & -t & 1 & & \\ & & -t & 1 & \\ & & & -t & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} 8118 \\ -11\,482 \\ 1 \\ 5741 \\ -2030 \end{pmatrix}.$$

The best 32-bit posit representation of the exact value is the following, shown to 10 decimals:

```
pexp = pex; N[pexp]
colorcodep[x2p[pexp]]
```

-1.91574×10^{-11}

```
11111111110111010111011111010110→-00000000010001010000100000101010
```

The color coding shows that the fraction only has 20 bits for such a small-magnitude value, for which an ULP is about one part in about 10^6. That's appropriate, since the input value only has six significant digits. Here's the $Ax = b$ problem setup and the approximate solution $\hat{x}$, expressed as exact rational numbers and approximate decimals:

```
A = {{1, 0, 0, 0, 0}, {-t, 1, 0, 0, 0},
    {0, -t, 1, 0, 0}, {0, 0, -t, 1, 0}, {0, 0, 0, -t, 1}};
b = {8118, -11 482, 1, 5741, -2030};
n = Length[b];
x̂ = LinearSolve[A, b];
N[x̂, 12]
```

```
{8118.00000000, -5741.70537357,
 -4059.00006189, 2870.85264302, -1.91573885063 × 10^-11}
```

Here is what the approximate inverse looks like, at posit precision:

```
R = Inverse[A]; MatrixForm[R]
```

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ \frac{94\,906\,295}{134\,217\,728} & 1 & 0 & 0 & 0 \\ \frac{134\,217\,811}{268\,435\,456} & \frac{94\,906\,295}{134\,217\,728} & 1 & 0 & 0 \\ \frac{47\,453\,177}{134\,217\,728} & \frac{134\,217\,811}{268\,435\,456} & \frac{94\,906\,295}{134\,217\,728} & 1 & 0 \\ \frac{67\,108\,947}{268\,435\,456} & \frac{47\,453\,177}{134\,217\,728} & \frac{134\,217\,811}{268\,435\,456} & \frac{94\,906\,295}{134\,217\,728} & 1 \end{pmatrix}$$

The refinement matrix is $I - R\,A$, computed at quire precision but then rounded:

```
R = Inverse[A]; MatrixForm[R]
```

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ \frac{94\,906\,295}{134\,217\,728} & 1 & 0 & 0 & 0 \\ \frac{134\,217\,811}{268\,435\,456} & \frac{94\,906\,295}{134\,217\,728} & 1 & 0 & 0 \\ \frac{47\,453\,177}{134\,217\,728} & \frac{134\,217\,811}{268\,435\,456} & \frac{94\,906\,295}{134\,217\,728} & 1 & 0 \\ \frac{67\,108\,947}{268\,435\,456} & \frac{47\,453\,177}{134\,217\,728} & \frac{134\,217\,811}{268\,435\,456} & \frac{94\,906\,295}{134\,217\,728} & 1 \end{pmatrix}$$

```
ref = IdentityMatrix[5] - A.R;
ref = x2pint[ref];
Style[MatrixForm[ref], 5]
```

$$\begin{pmatrix} \text{Interval}[\{0,0\}] & \text{Interval}[\{0,0\}] & \text{Interval}[\{0,0\}] & \text{Interval}[\{0,0\}] & \text{Interval}[\{0,0\}] \\ \text{Interval}[\{0,0\}] & \text{Interval}[\{0,0\}] & \text{Interval}[\{0,0\}] & \text{Interval}[\{0,0\}] & \text{Interval}[\{0,0\}] \\ \text{Interval}\left[\left\{\frac{365\,645}{1\,125\,899\,906\,842\,624}, \frac{1\,462\,581}{4\,503\,599\,627\,370\,496}\right\}\right] & \text{Interval}[\{0,0\}] & \text{Interval}[\{0,0\}] & \text{Interval}[\{0,0\}] & \text{Interval}[\{0,0\}] \\ \text{Interval}\left[\left\{-\frac{650\,367}{562\,949\,953\,421\,312}, -\frac{1\,300\,733}{1\,125\,899\,906\,842\,624}\right\}\right] & \text{Interval}\left[\left\{\frac{365\,645}{1\,125\,899\,906\,842\,624}, \frac{1\,462\,581}{4\,503\,599\,627\,370\,496}\right\}\right] & \text{Interval}[\{0,0\}] & \text{Interval}[\{0,0\}] & \text{Interval}[\{0,0\}] \\ \text{Interval}\left[\left\{\frac{1\,102\,687}{1\,125\,899\,906\,842\,624}, \frac{34\,459}{35\,184\,372\,088\,832}\right\}\right] & \text{Interval}\left[\left\{-\frac{650\,367}{562\,949\,953\,421\,312}, -\frac{1\,300\,733}{1\,125\,899\,906\,842\,624}\right\}\right] & \text{Interval}\left[\left\{\frac{365\,645}{1\,125\,899\,906\,842\,624}, \frac{1\,462\,581}{4\,503\,599\,627\,370\,496}\right\}\right] & \text{Interval}[\{0,0\}] & \text{Interval}[\{0,0\}] \end{pmatrix}$$

We compute $R \cdot b$ in quire precision, then round to posit as the approximate solution, $\tilde{x}$.

```
x̃ = R.b
```

$$\left\{8118, -\frac{23\,518\,025}{4096}, -\frac{66\,502\,657}{16\,384}, \frac{23\,518\,025}{8192}, -\frac{3561}{268\,435\,456}\right\}$$

We always calculate the “defect” exactly, at quire precision, and round to posit precision:

```
d̃ = b - A.x̃
```

$$\left\{0, -\frac{3443}{67\,108\,864}, \frac{19\,472\,337}{549\,755\,813\,888}, -\frac{4\,812\,023}{274\,877\,906\,944}, \frac{28\,667\,951}{1\,099\,511\,627\,776}\right\}$$

Now we need an interval enclosure around $\tilde{x} - \hat{x}$ as a starting guess. As usual, be ultra-conservative and use the largest possible starting interval.

```
e = Table[Interval[{-maxpos, maxpos}], {i, 1, n}];
```

This at last sets up the iteration scheme specified by Kulisch on page 31 of “Arithmetic of the Digital Computer”. We can watch just what happens to the last element of e, since that’s the answer we seek.

```
e = x2pint[(ref.e + R.d̃)]; N[e⟦-1⟧]
```

```
Interval[{-3.32696 × 10^27, 3.32696 × 10^27}]
```

It has a long way to come down, but it’s shrinking fast.

```
e = x2pint[(ref.e + R.d̃)]; N[e⟦-1⟧]
```

```
Interval[{-1.41344 × 10^17, 1.41344 × 10^17}]
```

This is looking good.

```
e = x2pint[(ref.e + R.d̃)]; N[e⟦-1⟧]
```

```
Interval[{0.0000132657, 0.0000132657}]
```

That looks converged, but do it one more time just to be sure:

```
e = x2pint[(ref.e + R.d̃)]; N[e⟦-1⟧]
```

```
Interval[{0.0000132657, 0.0000132657}]
```

It has stopped changing. Now apply the correction:

```
x̃ = x̃ + e; N[x̃]
```

```
{Interval[{8118., 8118.}], Interval[{-5741.71, -5741.71}],
 Interval[{-4059., -4059.}], Interval[{2870.85, 2870.85}],
 Interval[{-2.00089 × 10^-11, -1.90994 × 10^-11}]}
```

That last interval is the bound, and it’s better than you can do with a 32-bit float by far, as we will see later. But before we stop, let’s try using a point in each interval as a new starting point, because we lost a lot of precision when correcting the last term. It was so far away from the right answer, there weren’t many significant digits available to really zoom in on the answer. The beauty of the method is that it produces a rigorous bound even with a poor starting point and a very

approximate inverse to the matrix A. As long as A R – I has a spectral radius less than 1, the enclosure interval will converge to an ULP-wide bound. This has to be one of the supreme achievements of the interval arithmetic community.

```
x̃ = Table[Min[x̃[[i]]], {i, 1, n}]; N[x̃]
```

```
{8118., -5741.71, -4059., 2870.85, -2.00089 × 10^-11}
```

Calculate an exact residual with this better starting guess for the answer:

```
d = b - A.x̃; N[d, 12]
```

```
{0, 0, 5.09592368303 × 10^-14, 5.45955751074 × 10^-13, 4.39966165171 × 10^-13}
```

That very small correction shows this is *much* closer to the true answer. But we aren't simply using a typical iterative method; we are *proving* that the answer lies inside an interval. Use the usual all-encompassing interval enclosure to start, and round **d** to the nearest posit. (If we do not round **d** to the nearest posit **d̃**, we cannot use the quire to compute *R·d* exactly).

```
e = Table[Interval[{-maxpos, maxpos}], {i, 1, n}];
d̃ = d̄; N[d̃]
```

```
{0., 0., 5.09592 × 10^-14, 5.45956 × 10^-13, 4.39966 × 10^-13}
```

With the same refinement operator as before, but this better starting point iterate until things stop changing. Display just the last value as a decimal, and when it seems to be stable, check by displaying it as a rational number.

```
e = x2pint[(ref.e + R.d̃)]; N[e[[-1]]]
```

```
Interval[{-3.32696 × 10^27, 3.32696 × 10^27}]
```

```
e = x2pint[(ref.e + R.d̃)]; N[e[[-1]]]
```

```
Interval[{-1.41344 × 10^17, 1.41344 × 10^17}]
```

```
e = x2pint[(ref.e + R.d̃)]; N[e[[-1]]]
```

```
Interval[{8.51492 × 10^-13, 8.51496 × 10^-13}]
```

```
e = x2pint[(ref.e + R.d̃)]; e⟦-1⟧
```

$$\text{Interval}\left[\left\{\frac{122\,713}{144\,115\,188\,075\,855\,872}, \frac{245\,427}{288\,230\,376\,151\,711\,744}\right\}\right]$$


```
e = x2pint[(ref.e + R.d̃)]; e⟦-1⟧
```

$$\text{Interval}\left[\left\{\frac{122\,713}{144\,115\,188\,075\,855\,872}, \frac{245\,427}{288\,230\,376\,151\,711\,744}\right\}\right]$$

That's stable. Add the correction, and look at just the last value, which is the bound on the answer:

```
x̃ = x̃ + e; bound = x2pint[x̃⟦-1⟧]; N[bound, 12]
```

$$\text{Interval}\left[\left\{-1.91573978903 \times 10^{-11}, -1.91573701347 \times 10^{-11}\right\}\right]$$

We can use color-coded output to show that these posit bounds are one ULP apart, so this is a maximum-accuracy result, with *six* decimals of accuracy.

```
colorcodep[x2p[bound⟦1,1⟧]]
colorcodep[x2p[bound⟦1,2⟧]]
```

```
11111111110111010111011111010110→-0000000001000101000100000101010

11111111110111010111011111010111→-0000000001000101000100000101001
```

That's pretty spectacular, because 32-bit posits produced an answer correct to six decimals with a provable enclosure, whereas 64-bit floats produced an answer correct to two decimals with no guarantee of correctness whatsoever. Score 1 for Dr. Kulisch.

Just for fun, let's see how awful the result would be if we had used 32-bit *floats*. We convert the value of *t* to the nearest float.

```
setfloatenv[{32, 8}]; t = 0.707107; N[t, 20]
```

```
0.70710700750350952148
```

The *exact* value of the polynomial using that value of *t* should be as follows:

```
fex = N[P[t], 49]
```

$$-1.98129023874666409641551753311301380427371463671 7 \times 10^{-11}$$

Here is the polynomial evaluated with Horner's rule and IEEE standard rounding after every

operation. It's kind of fun to see the stack of underlines, which tell you exactly where the roundings happen.

```
(((8118 t - 11 482) t + 1) t + 5741) t - 2030
```

0

Whoops. Infinite decimal error. Float arithmetic thinks the value is a root of the polynomial! But wait, we can at least use fused multiply-adds, which cuts the number of roundings from eight down to four:

```
(((8118 t - 11 482) t + 1) t + 5741) t - 2030
```

$$-\frac{895\,409}{17\,179\,869\,184}$$

That looks a lot bigger than the correct value we calculated, **fex**. Let's see just how much bigger it is:

```
N[% / fex]
```

2.63059×10^{6}

The answer is now off by a factor of *2.6 million*. Well, at least it got the sign right!

The combination of posit arithmetic, quire accumulation, and the shrinking interval enclosure technique allow 32-bit posits to dramatically outperform 64-bit floats. The 32-bit posits produced a tight, provable bound with six-decimal accuracy, whereas the 64-bit floats produced a guess with no indication of how many decimal in the guess are correct. And only the first two decimals were correct.

12.5 Solving a real PDE problem (Sandia's data set)

12.5.1 Insights regarding Ax = *b* and the quire data type

If A and *b* are expressed in posits (or floats) and A is nonsingular, there is exactly **one** answer for *x*: a vector of rational numbers.

There is exactly one $\tilde{x}$ expressible with posits that is closer to *x* than any other. (Rounding is unique).

Matrix A times *x* is a vector of quires, with **no** rounding error! (512 bits per quire, if we use 32-bit posits.)

Gaussian elimination with exact dot products as the innermost loop means $O(n^2)$ roundings per element instead of $O(n^3)$. On a petascale problem for which $n \approx 10^6$, that means about three more decimals of accuracy in the answer.

Let's see if we can import a small test file from Sandia based on a real PDE. Start with their 81-variable system, apparently generated by a 9-by-9 Poisson problem using a 5-point stencil. The first two numbers are the row-column coordinates within the matrix, and the long decimal is a double-precision component value. Apologies for this long table of numbers in the middle of this document…ugh!

```
sandia81A = {
   1, 1, 0.40000000000000013322676295501878 4851,
   2, 1, -0.09999999999999997779553950749686 9192,
   10, 1, -0.10000000000000006106226635438360 9723,
   1, 2, -0.09999999999999997779553950749686 9192,
   2, 2, 0.40000000000000013322676295501878 4851,
   3, 2, -0.09999999999999997779553950749686 9192,
   11, 2, -0.10000000000000008881784197001252 3234,
   2, 3, -0.09999999999999997779553950749686 9192,
   3, 3, 0.41666666666666679619268620626826 3049,
   4, 3, -0.10833333333333336478965236437943 5312,
   12, 3, -0.10833333333333339254522798000834 8823,
   3, 4, -0.10833333333333336478965236437943 5312,
   4, 4, 0.42500000000000015543122344752191 5659,
   5, 4, -0.10000000000000003330669073875469 6213,
   13, 4, -0.11666666666666671015040179781863 1167,
   4, 5, -0.10000000000000003330669073875469 6213,
   5, 5, 0.40000000000000018873791418627661 1872,
   6, 5, -0.10000000000000003330669073875469 6213,
   14, 5, -0.10000000000000003330669073875469 6213,
   5, 6, -0.10000000000000003330669073875469 6213,
   6, 6, 0.40000000000000013322676295501878 4851,
   7, 6, -0.10000000000000003330669073875469 6213,
   15, 6, -0.10000000000000003330669073875469 6213,
   6, 7, -0.10000000000000003330669073875469 6213,
   7, 7, 0.40000000000000013322676295501878 4851,
   8, 7, -0.09999999999999997779553950749686 9192,
   16, 7, -0.10000000000000006106226635438360 9723,
   7, 8, -0.09999999999999997779553950749686 9192,
   8, 8, 0.40000000000000013322676295501878 4851,
   9, 8, -0.09999999999999997779553950749686 9192,
   17, 8, -0.10000000000000008881784197001252 3234,
   8, 9, -0.09999999999999997779553950749686 9192,
   9, 9, 0.40000000000000013322676295501878 4851,
   18, 9, -0.10000000000000006106226635438360 9723,
   1, 10, -0.10000000000000006106226635438360 9723,
   10, 10, 0.40000000000000013322676295501878 4851,
   11, 10, -0.09999999999999997779553950749686 9192,
   19, 10, -0.10000000000000003330669073875469 6213,
   2, 11, -0.10000000000000008881784197001252 3234,
   10, 11, -0.09999999999999997779553950749686 9192,
```

```
11, 11, 0.41666666666666679619268620626826 3049,
12, 11, -0.10833333333333330927850113312160 8291,
20, 11, -0.10833333333333339254522798000834 8823,
3, 12, -0.1083333333333333925452279800083488 23,
11, 12, -0.10833333333333330927850113312160 8291,
12, 12, 0.51666666666666682949937694502295 9262,
13, 12, -0.15000000000000007771561172376095 7830,
21, 12, -0.15000000000000002220446049250313 0808,
4, 13, -0.11666666666666671015040179781863 1167,
12, 13, -0.15000000000000007771561172376095 7830,
13, 13, 0.58333333333333359238537241253652 6099,
14, 13, -0.12500000000000005551115123125782 7021,
22, 13, -0.19166666666666667961926862062682 6305,
5, 14, -0.10000000000000003330669073875469 6213,
13, 14, -0.12500000000000005551115123125782 7021,
14, 14, 0.45000000000000017763568394002504 6468,
15, 14, -0.10000000000000004718447854656915 2968,
23, 14, -0.12500000000000000000000000000000 0000,
6, 15, -0.10000000000000003330669073875469 6213,
14, 15, -0.10000000000000004718447854656915 2968,
15, 15, 0.40000000000000013322676295501878 4851,
16, 15, -0.10000000000000004718447854656915 2968,
24, 15, -0.10000000000000000555111512312578 2702,
7, 16, -0.10000000000000006106226635438360 9723,
15, 16, -0.10000000000000004718447854656915 2968,
16, 16, 0.40000000000000013322676295501878 4851,
17, 16, -0.09999999999999997777955395074968 69192,
25, 16, -0.10000000000000003330669073875469 6213,
8, 17, -0.10000000000000008881784197001252 3234,
16, 17, -0.09999999999999997777955395074968 69192,
17, 17, 0.40000000000000007771561172376095 7830,
18, 17, -0.09999999999999997777955395074968 69192,
26, 17, -0.10000000000000006106226635438360 9723,
9, 18, -0.10000000000000006106226635438360 9723,
17, 18, -0.09999999999999997777955395074968 69192,
18, 18, 0.40000000000000013322676295501878 4851,
27, 18, -0.10000000000000003330669073875469 6213,
10, 19, -0.10000000000000003330669073875469 6213,
19, 19, 0.41666666666666679619268620626826 3049,
20, 19, -0.10833333333333330927850113312160 8291,
28, 19, -0.10833333333333336478965236437943 5312,
11, 20, -0.10833333333333339254522798000834 8823,
19, 20, -0.10833333333333330927850113312160 8291,
20, 20, 0.51666666666666671847707448250730 5220,
21, 20, -0.14999999999999996669330926124530 3787,
29, 20, -0.15000000000000007771561172376095 7830,
12, 21, -0.15000000000000002220446049250313 0808,
20, 21, -0.14999999999999996669330926124530 3787,
21, 21, 0.73333333333333339254522798000834 8823,
22, 21, -0.21666666666666675733488034438778 4135,
30, 21, -0.21666666666666672957930472875887 0624,
13, 22, -0.19166666666666667961926862062682 6305,
21, 22, -0.21666666666666675733488034438778 4135,
22, 22, 0.90833333333333365899875389004591 8524,
23, 22, -0.22500000000000008881784197001252 3234,
```

```
    31, 22, -0.27500000000000002220446049250313080 8,
    14, 23, -0.12500000000000000000000000000000000 0,
    22, 23, -0.22500000000000000888178419700125232 34,
    23, 23, 0.70000000000000028865798640254070051 0,
    24, 23, -0.12500000000000000555111512312578270 21,
    32, 23, -0.22500000000000000055511151231257827 02,
    15, 24, -0.10000000000000000055511151231257827 02,
    23, 24, -0.12500000000000000555111512312578270 21,
    24, 24, 0.45000000000000001221245327087672194 47,
    25, 24, -0.10000000000000006106226635438360972 3,
    33, 24, -0.12500000000000000000000000000000000 0,
    16, 25, -0.10000000000000000333066907387546962 13,
    24, 25, -0.10000000000000006106226635438360972 3,
    25, 25, 0.40000000000000013322676295501878485 1,
    26, 25, -0.09999999999999999777955395074968691 92,
    34, 25, -0.10000000000000000333066907387546962 13,
    17, 26, -0.10000000000000006106226635438360972 3,
    25, 26, -0.09999999999999999777955395074968691 92,
    26, 26, 0.40000000000000007771561172376095783 0,
    27, 26, -0.09999999999999999777955395074968691 92,
    35, 26, -0.10000000000000006106226635438360972 3,
    18, 27, -0.10000000000000000333066907387546962 13,
    26, 27, -0.09999999999999999777955395074968691 92,
    27, 27, 0.40000000000000013322676295501878485 1,
    36, 27, -0.10000000000000000333066907387546962 13,
    19, 28, -0.10833333333333336478965236437943531 2,
    28, 28, 0.42500000000000009992007221626408863 8,
    29, 28, -0.11666666666666664076146275874634739 0,
    37, 28, -0.10000000000000000333066907387546962 13,
    20, 29, -0.15000000000000007771561172376095783 0,
    28, 29, -0.11666666666666664076146275874634739 0,
    29, 29, 0.58333333333333337034076748750521801 4,
    30, 29, -0.19166666666666659635254177374008577 3,
    38, 29, -0.12500000000000008326672684688674053 2,
    21, 30, -0.21666666666666672957930472875887062 4,
    29, 30, -0.19166666666666659635254177374008577 3,
    30, 30, 0.90833333333333343695414896501461044 0,
    31, 30, -0.27500000000000013322676295501878485 1,
    39, 30, -0.22500000000000000333066907387546962 13,
    22, 31, -0.27500000000000002220446049250313080 8,
    30, 31, -0.27500000000000013322676295501878485 1,
    31, 31, 1.10000000000000003108624468950438313 19,
    32, 31, -0.27500000000000013322676295501878485 1,
    40, 31, -0.27500000000000002220446049250313080 8,
    23, 32, -0.22500000000000000055511151231257827 02,
    31, 32, -0.27500000000000013322676295501878485 1,
    32, 32, 0.90833333333333335479764514275302644 82,
    33, 32, -0.19166666666666676288599546751356683 7,
    41, 32, -0.21666666666666667406815349750104360 3,
    24, 33, -0.12500000000000000000000000000000000 0,
    32, 33, -0.19166666666666676288599546751356683 7,
    33, 33, 0.58333333333333348136306995002087205 6,
    34, 33, -0.11666666666666672402818960563308792 2,
    42, 33, -0.15000000000000002220446049250313080 8,
    25, 34, -0.10000000000000000333066907387546962 13,
```

```
33, 34, -0.11666666666666672402818960563308792 2,
34, 34, 0.42500000000000009992007221626408863 8,
35, 34, -0.09999999999999997779553950749686919 2,
43, 34, -0.10833333333333336478965236437943531 2,
26, 35, -0.10000000000000006106226635438360972 3,
34, 35, -0.09999999999999997779553950749686919 2,
35, 35, 0.40000000000000007771561172376095783 0,
36, 35, -0.09999999999999997779553950749686919 2,
44, 35, -0.10000000000000006106226635438360972 3,
27, 36, -0.10000000000000003330669073875469621 3,
35, 36, -0.09999999999999997779553950749686919 2,
36, 36, 0.40000000000000013322676295501878485 1,
45, 36, -0.10000000000000003330669073875469621 3,
28, 37, -0.10000000000000003330669073875469621 3,
37, 37, 0.40000000000000013322676295501878485 1,
38, 37, -0.09999999999999997779553950749686919 2,
46, 37, -0.10000000000000003330669073875469621 3,
29, 38, -0.12500000000000008326672684688674053 2,
37, 38, -0.09999999999999997779553950749686919 2,
38, 38, 0.45000000000000012212453270876721944 7,
39, 38, -0.12499999999999998612221219218554324 5,
47, 38, -0.10000000000000006106226635438360972 3,
30, 39, -0.22500000000000003330669073875469621 3,
38, 39, -0.12499999999999998612221219218554324 5,
39, 39, 0.70000000000000017763568394002504646 8,
40, 39, -0.22500000000000008881784197001252323 4,
48, 39, -0.12500000000000005551115123125782702 1,
31, 40, -0.27500000000000002220446049250313080 8,
39, 40, -0.22500000000000008881784197001252323 4,
40, 40, 0.90833333333333365899875389004591852 4,
41, 40, -0.21666666666666678509045596001669764 5,
49, 40, -0.19166666666666665186369300499791279 4,
32, 41, -0.21666666666666667406815349750104360 3,
40, 41, -0.21666666666666678509045596001669764 5,
41, 41, 0.73333333333333350356753044252400286 5,
42, 41, -0.15000000000000007771561172376095783 0,
50, 41, -0.15000000000000002220446049250313080 8,
33, 42, -0.15000000000000002220446049250313080 8,
41, 42, -0.15000000000000007771561172376095783 0,
42, 42, 0.51666666666666682949937694502295926 2,
43, 42, -0.10833333333333339254522798000834882 3,
51, 42, -0.10833333333333337034076748750521801,
34, 43, -0.10833333333333336478965236437943531 2,
42, 43, -0.10833333333333339254522798000834882 3,
43, 43, 0.41666666666666679619268620626826304 9,
44, 43, -0.09999999999999997779553950749686919 2,
52, 43, -0.10000000000000003330669073875469621 3,
35, 44, -0.10000000000000006106226635438360972 3,
43, 44, -0.09999999999999997779553950749686919 2,
44, 44, 0.40000000000000007771561172376095783 0,
45, 44, -0.09999999999999997779553950749686919 2,
53, 44, -0.10000000000000006106226635438360972 3,
36, 45, -0.10000000000000003330669073875469621 3,
44, 45, -0.09999999999999997779553950749686919 2,
45, 45, 0.40000000000000013322676295501878485 1,
```

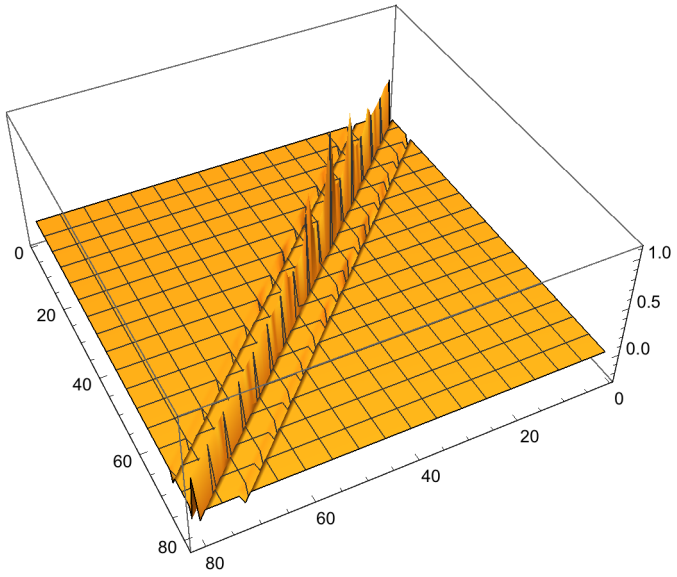
```
    54, 45, -0.10000000000000003330669073875469621 3,
    37, 46, -0.10000000000000003330669073875469621 3,
    46, 46, 0.40000000000000013322676295501878485 1,
    47, 46, -0.09999999999999997779553950749686919 2,
    55, 46, -0.10000000000000003330669073875469621 3,
    38, 47, -0.10000000000000006106226635438360972 3,
    46, 47, -0.09999999999999997779553950749686919 2,
    47, 47, 0.40000000000000013322676295501878485 1,
    48, 47, -0.09999999999999997779553950749686919 2,
    56, 47, -0.10000000000000006106226635438360972 3,
    39, 48, -0.12500000000000005551115123125782702 1,
    47, 48, -0.09999999999999997779553950749686919 2,
    48, 48, 0.45000000000000012212453270876721944 7,
    49, 48, -0.12500000000000005551115123125782702 1,
    57, 48, -0.10000000000000003330669073875469621 3,
    40, 49, -0.19166666666666665186369300499791279 4,
    48, 49, -0.12500000000000005551115123125782702 1,
    49, 49, 0.58333333333333337034076748750521801 4,
    50, 49, -0.15000000000000007771561172376095783 0,
    58, 49, -0.11666666666666666851703837437526090 1,
    41, 50, -0.15000000000000002220446049250313080 8,
    49, 50, -0.15000000000000007771561172376095783 0,
    50, 50, 0.51666666666666682949937694502295926 2,
    51, 50, -0.10833333333333339254522798000834882 3,
    59, 50, -0.10833333333333333703407674875052180 1,
    42, 51, -0.10833333333333333703407674875052180 1,
    50, 51, -0.10833333333333339254522798000834882 3,
    51, 51, 0.41666666666666679619268620626826304 9,
    52, 51, -0.10000000000000006106226635438360972 3,
    60, 51, -0.10000000000000000555111512312578270 2,
    43, 52, -0.10000000000000003330669073875469621 3,
    51, 52, -0.10000000000000006106226635438360972 3,
    52, 52, 0.40000000000000013322676295501878485 1,
    53, 52, -0.09999999999999997779553950749686919 2,
    61, 52, -0.10000000000000003330669073875469621 3,
    44, 53, -0.10000000000000006106226635438360972 3,
    52, 53, -0.09999999999999997779553950749686919 2,
    53, 53, 0.40000000000000007771561172376095783 0,
    54, 53, -0.09999999999999997779553950749686919 2,
    62, 53, -0.10000000000000006106226635438360972 3,
    45, 54, -0.10000000000000003330669073875469621 3,
    53, 54, -0.09999999999999997779553950749686919 2,
    54, 54, 0.40000000000000013322676295501878485 1,
    63, 54, -0.10000000000000003330669073875469621 3,
    46, 55, -0.10000000000000003330669073875469621 3,
    55, 55, 0.40000000000000013322676295501878485 1,
    56, 55, -0.09999999999999997779553950749686919 2,
    64, 55, -0.10000000000000003330669073875469621 3,
    47, 56, -0.10000000000000006106226635438360972 3,
    55, 56, -0.09999999999999997779553950749686919 2,
    56, 56, 0.40000000000000007771561172376095783 0,
    57, 56, -0.09999999999999997779553950749686919 2,
    65, 56, -0.10000000000000006106226635438360972 3,
    48, 57, -0.10000000000000003330669073875469621 3,
    56, 57, -0.09999999999999997779553950749686919 2,
```

```
57, 57, 0.40000000000000013322676295501878485 1,
58, 57, -0.10000000000000006106226635438360972 3,
66, 57, -0.10000000000000003330669073875469621 3,
49, 58, -0.11666666666666666851703837437526090 1,
57, 58, -0.10000000000000006106226635438360972 3,
58, 58, 0.42500000000000015543122344752191565 9,
59, 58, -0.10833333333333339254522798000834882 3,
67, 58, -0.10000000000000000555111512312578270 2,
50, 59, -0.10833333333333333703407674875052180 1,
58, 59, -0.10833333333333339254522798000834882 3,
59, 59, 0.41666666666666679619268620626826304 9,
60, 59, -0.10000000000000006106226635438360972 3,
68, 59, -0.10000000000000000555111512312578270 2,
51, 60, -0.10000000000000000555111512312578270 2,
59, 60, -0.10000000000000006106226635438360972 3,
60, 60, 0.40000000000000013322676295501878485 1,
61, 60, -0.10000000000000006106226635438360972 3,
69, 60, -0.10000000000000000555111512312578270 2,
52, 61, -0.10000000000000003330669073875469621 3,
60, 61, -0.10000000000000006106226635438360972 3,
61, 61, 0.40000000000000013322676295501878485 1,
62, 61, -0.09999999999999997777955395074968691 92,
70, 61, -0.10000000000000003330669073875469621 3,
53, 62, -0.10000000000000006106226635438360972 3,
61, 62, -0.09999999999999997777955395074968691 92,
62, 62, 0.40000000000000007771561172376095783 0,
63, 62, -0.09999999999999997777955395074968691 92,
71, 62, -0.10000000000000006106226635438360972 3,
54, 63, -0.10000000000000003330669073875469621 3,
62, 63, -0.09999999999999997777955395074968691 92,
63, 63, 0.40000000000000013322676295501878485 1,
72, 63, -0.10000000000000003330669073875469621 3,
55, 64, -0.10000000000000003330669073875469621 3,
64, 64, 0.40000000000000007771561172376095783 0,
65, 64, -0.10000000000000000555111512312578270 2,
73, 64, -0.10000000000000000555111512312578270 2,
56, 65, -0.10000000000000006106226635438360972 3,
64, 65, -0.10000000000000000555111512312578270 2,
65, 65, 0.40000000000000007771561172376095783 0,
66, 65, -0.10000000000000000555111512312578270 2,
74, 65, -0.10000000000000001942890293094023945 7,
57, 66, -0.10000000000000003330669073875469621 3,
65, 66, -0.10000000000000000555111512312578270 2,
66, 66, 0.40000000000000013322676295501878485 1,
67, 66, -0.10000000000000007494005416219806647 9,
75, 66, -0.10000000000000000555111512312578270 2,
58, 67, -0.10000000000000000555111512312578270 2,
66, 67, -0.10000000000000007494005416219806647 9,
67, 67, 0.40000000000000013322676295501878485 1,
68, 67, -0.10000000000000007494005416219806647 9,
76, 67, -0.09999999999999997777955395074968691 92,
59, 68, -0.10000000000000000555111512312578270 2,
67, 68, -0.10000000000000007494005416219806647 9,
68, 68, 0.40000000000000013322676295501878485 1,
69, 68, -0.10000000000000007494005416219806647 9,
```

```
    77, 68, -0.09999999999999997779553950749686919 2,
    60, 69, -0.10000000000000000555111512312578270 2,
    68, 69, -0.10000000000000000749400541621980664 79,
    69, 69, 0.40000000000000001332267629550187848 51,
    70, 69, -0.10000000000000000749400541621980664 79,
    78, 69, -0.09999999999999997779553950749686919 2,
    61, 70, -0.10000000000000000333066907387546962 13,
    69, 70, -0.10000000000000000749400541621980664 79,
    70, 70, 0.40000000000000000777156117237609578 30,
    71, 70, -0.10000000000000000555111512312578270 2,
    79, 70, -0.10000000000000000555111512312578270 2,
    62, 71, -0.10000000000000000610622663543836097 23,
    70, 71, -0.10000000000000000555111512312578270 2,
    71, 71, 0.40000000000000000777156117237609578 30,
    72, 71, -0.10000000000000000555111512312578270 2,
    80, 71, -0.10000000000000000194289029309402394 57,
    63, 72, -0.10000000000000000333066907387546962 13,
    71, 72, -0.10000000000000000555111512312578270 2,
    72, 72, 0.40000000000000001332267629550187848 51,
    81, 72, -0.10000000000000000555111512312578270 2,
    64, 73, -0.10000000000000000555111512312578270 2,
    73, 73, 0.40000000000000001332267629550187848 51,
    74, 73, -0.10000000000000000555111512312578270 2,
    65, 74, -0.10000000000000000194289029309402394 57,
    73, 74, -0.10000000000000000555111512312578270 2,
    74, 74, 0.40000000000000000777156117237609578 30,
    75, 74, -0.10000000000000000555111512312578270 2,
    66, 75, -0.10000000000000000555111512312578270 2,
    74, 75, -0.10000000000000000555111512312578270 2,
    75, 75, 0.40000000000000001332267629550187848 51,
    76, 75, -0.10000000000000000610622663543836097 23,
    67, 76, -0.09999999999999997779553950749686919 2,
    75, 76, -0.10000000000000000610622663543836097 23,
    76, 76, 0.40000000000000000777156117237609578 30,
    77, 76, -0.10000000000000000610622663543836097 23,
    68, 77, -0.09999999999999997779553950749686919 2,
    76, 77, -0.10000000000000000610622663543836097 23,
    77, 77, 0.40000000000000001332267629550187848 51,
    78, 77, -0.10000000000000000610622663543836097 23,
    69, 78, -0.09999999999999997779553950749686919 2,
    77, 78, -0.10000000000000000610622663543836097 23,
    78, 78, 0.40000000000000001332267629550187848 51,
    79, 78, -0.10000000000000000610622663543836097 23,
    70, 79, -0.10000000000000000555111512312578270 2,
    78, 79, -0.10000000000000000610622663543836097 23,
    79, 79, 0.40000000000000001332267629550187848 51,
    80, 79, -0.10000000000000000555111512312578270 2,
    71, 80, -0.10000000000000000194289029309402394 57,
    79, 80, -0.10000000000000000555111512312578270 2,
    80, 80, 0.40000000000000000777156117237609578 30,
    81, 80, -0.10000000000000000555111512312578270 2,
    72, 81, -0.10000000000000000555111512312578270 2,
    80, 81, -0.10000000000000000555111512312578270 2,
    81, 81, 0.40000000000000001332267629550187848 51};
```

While this clearly should be stored and solved as a sparse matrix, for purposes of rapid prototyping we simply populate a dense matrix so that we can use the dense matrix solver developed in a previous section. As is typical with a partial differential equation solved with a nearest-neighbor stencil (finite difference) approximation, the system has a main diagonal, adjacent diagonals, and diagonals that are 9 elements away, the width or height of the square domain. The plot shows the matrix with the values of coefficients in the *z* dimension.

```
A = Table[0, {i, 1, 81}, {j, 1, 81}];
For[i = 1, i < Length[sandia81A],
   i += 3, A[[sandia81A[[i]], sandia81A[[i+1]]]] = sandia81A[[i+2]]];
ListPlot3D[A, ViewPoint → {1, 2, 2}]
```



Sandia also supplied their right-hand side vector *b*, which does not have the obvious decimal approximation quality of the matrix input values.

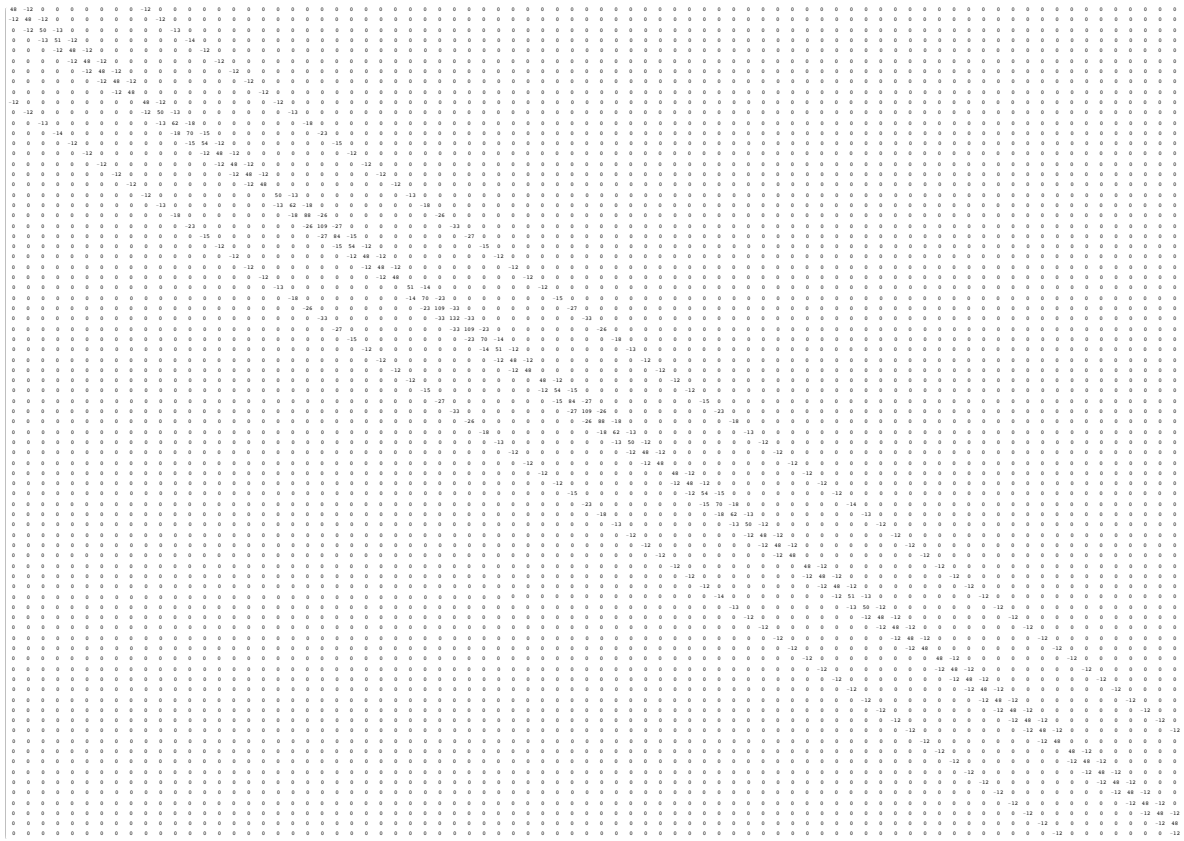
```
sandia81b = {
    1, 1, 0.56828262666770212874212203686110675,
    2, 1, -0.19436539490862536627702183977817852,
    3, 1, -0.68840704695802767965062685107113793 5,
    4, 1, -0.23109355820638430301094956575980177 1,
    5, 1, 0.54558337340532780945778768000309355 6,
    6, 1, 0.56828262666770112954139904104522429 4,
    7, 1, -0.19436539490862203560794796430855058 1,
    8, 1, -0.68840704695802656942760222591459751 1,
    9, 1, -0.23109355820638410872192025635740719 7,
    10, 1, -0.94219985842678599663457816859590821 0,
    11, 1, -0.91950060516441078917182494478765875 1,
```

```
    12, 1, 0.37391723175908198051331510214367881 4,
    13, 1, 1.15059416337079656322828213887987658 4,
    14, 1, 0.33718906846131752041983986600826028 7,
    15, 1, -0.94219985842678810605832495639333501 5,
    16, 1, -0.91950060516440956792649785711546428 5,
    17, 1, 0.37391723175908542220469144012895412 7,
    18, 1, 1.15059416337079634118367721384856849 9,
    19, 1, -1.15059416337079456482683781359810382 1,
    20, 1, -0.37391723175907620735358705132966861 1,
    21, 1, 0.91950060516441278757326927006943151 4,
    22, 1, 0.94219985842678410925543630582978949 0,
    23, 1, -0.33718906846132001842164527261047624 1,
    24, 1, -1.15059416337079478687144273862941190 6,
    25, 1, -0.37391723175907864984424122667405754 3,
    26, 1, 0.91950060516441367575168897019466385 2,
    27, 1, 0.94219985842678433130004123086109757 4,
    28, 1, 0.23109355820638616263451581289700698 1,
    29, 1, 0.68840704695802934498516378880594857 0,
    30, 1, 0.19436539490862125845183072669897228 5,
    31, 1, -0.56828262666770457123277537903049960 7,
    32, 1, -0.54558337340532736536857782994047738 6,
    33, 1, 0.23109355820638788348020398188964463 8,
    34, 1, 0.68840704695802679147220715094590559 6,
    35, 1, 0.19436539490861839962754231692088069 4,
    36, 1, -0.56828262666770434918817045399919152 3,
    37, 1, 1.29341783692349343581895482202526181 9,
    38, 1, 0.79937618487408423906259713476174511 0,
    39, 1, -0.79937618487408879097699809790356084 7,
    40, 1, -1.29341783692349432399737452215049415 8,
    41, 1, 0.00000000000000111022302462515654042 4,
    42, 1, 1.29341783692349521217579422227572649 7,
    43, 1, 0.79937618487408523826331929740263149 1,
    44, 1, -0.79937618487409134448995473576360382 1,
    45, 1, -1.29341783692349410195276959711918607 4,
    46, 1, 0.56828262666770223976442366620176471 8,
    47, 1, -0.19436539490862597689968538361426908 5,
    48, 1, -0.68840704695802856782904655119637027 4,
    49, 1, -0.23109355820638383116616410006827209 1,
    50, 1, 0.54558337340532891968081230515963397 9,
    51, 1, 0.56828262666770146260830642859218642 1,
    52, 1, -0.19436539490862286827521643317595589 9,
    53, 1, -0.68840704695802745760602192603982985 0,
    54, 1, -0.23109355820638394218846656258392613 4,
    55, 1, -0.94219985842678610765688063111156225 2,
    56, 1, -0.91950060516441167735024464491289109 0,
    57, 1, 0.37391723175908159193525648333888966 6,
    58, 1, 1.15059416337079722936209691397380083 8,
    59, 1, 0.33718906846131835308710833487566560 5,
    60, 1, -0.94219985842678810605832495639333501 5,
    61, 1, -0.91950060516441034508261509472504258 2,
    62, 1, 0.37391723175908497811548159006633795 8,
    63, 1, 1.15059416337079678527288706391118466 9,
    64, 1, -1.15059416337079589709446736378595233 0,
    65, 1, -0.37391723175907870535539245793188456 4,
    66, 1, 0.91950060516441278757326927006943151 4,
```

```
    67, 1, 0.9421998584267869958353003312367945 91,
    68, 1, -0.3371890684613184641094107973913196 47,
    69, 1, -1.1505941633707967852728870639111846 69,
    70, 1, -0.3739172317590812033571978645341005 18,
    71, 1, 0.9195006051644140088185963577416259 80,
    72, 1, 0.9421998584267869958353003312367945 91,
    73, 1, 0.2310935582063808058084219965166994 37,
    74, 1, 0.6884070469580265694276022259145975 11,
    75, 1, 0.1943653949086249221878119897155556 83,
    76, 1, -0.5682826266676993531845596407947596 16,
    77, 1, -0.5455833734053278094577876800030935 56,
    78, 1, 0.2310935582063824711429589342515100 72,
    79, 1, 0.6884070469580239048923431255389004 95,
    80, 1, 0.1943653949086220356079479643085505 81,
    81, 1, -0.5682826266676992421622571782791055 74};

b = Table[sandia81b⟦i+2⟧, {i, 1, Length[sandia81b], 3}];
```

Clearly, the matrix is constructed from exact rationals. Sandia gave the OK to reconstruct those original rationals, which will allow us to scale both A and *b* to improve the statement of A. If we multiply A by 120, all the values become very close to integers. With tiny 3-point type, the entire matrix fits in a page width once we reconstruct the original intention of the matrix description.

```
A = Round[120 A];
b = 120 b;
Style[MatrixForm[A], 3]
```



Here is the “correct answer” as supplied by Sandia using conventional double-precision floating point. We would expect that to have almost 15 decimals of accuracy, since this matrix appears to be diagonally dominant and well-conditioned.

```
sandia81x = {
    1, 1, -0.10445589578459187063419477681236458O,
    2, 1, -1.86118957003281138007366735109826549 9,
    3, 1, -1.72246357804553729131669115304248407 5,
    4, 1, 0.33592856668603693348629235515545588O,
    5, 1, 1.86296089880701387642147892620414495 5,
    6, 1, 0.54929877734319820792308064483222551 6,
    7, 1, -1.65687719865831462051630751375341787 9,
    8, 1, -1.71448160693834283918590699613559991 1,
    9, 1, 0.06831219699907999098886790534379542 8,
    10, 1, -4.23946027978257600210554301156662404 5,
    11, 1, -3.67418485721486165473947949067223817 1,
    12, 1, 1.11175924662782188434562158363405615 1,
    13, 1, 3.20714854390441539422340611054096370 9,
    14, 1, 1.11078251714554254725442206108709797 3,
    15, 1, -3.69171485745291549918078999326098710 3,
```

```
16, 1, -3.5186720159518967143696954735787585 38,
17, 1, 1.6147090434861250951570355027797631 92,
18, 1, 4.2986659769985013923587757744826376 44,
19, 1, -3.7572017818629928065377043822081759 57,
20, 1, -1.1241724807223756332774655675166286 53,
21, 1, 2.0270422971078216889395662292372435 33,
22, 1, 1.9588127973358753219912387066869996 49,
23, 1, -0.4428408629542803787160210049478337 17,
24, 1, -3.4862701240806259761484398040920495 99,
25, 1, -1.1457989995383877079859757941449061 04,
26, 1, 3.6541515022453845418226592300925403 83,
27, 1, 4.0057010338008396388431719969958066 94,
28, 1, 1.2076135475482656644885537389200180 77,
29, 1, 1.9607021647985096723942888274905271 83,
30, 1, 0.6666905772587675693685582700709346 68,
31, 1, -0.4261879101952409931719500946201151 24,
32, 1, -0.5182065072905017721893727866699919 10,
33, 1, 0.9670328088234223518071530634188093 25,
34, 1, 2.5067669572243760356400343880522996 19,
35, 1, 0.9469888795888243837595155127928592 26,
36, 1, -1.3520119283083724681659987254533916 71,
37, 1, 4.6042380664362498876585050311405211 69,
38, 1, 2.8423291740871698962678237876389175 65,
39, 1, -1.2736897387612948762125597568228840 83,
40, 1, -1.7455662292934521762077793027856387 20,
41, 1, 0.4909478941247824579363623342942446 47,
42, 1, 3.8377296533679277068529245298122987 15,
43, 1, 2.6218100999497573333485433977330103 52,
44, 1, -2.9646049618922738488890900043770670 89,
45, 1, -4.6779113599461110695187926467042416 33,
46, 1, 1.4328311748746287435096746776252985 00,
47, 1, -0.6662841643313396344794341530359815 81,
48, 1, -1.6380060584952782498646683961851522 33,
49, 1, 0.0275043553733280189688148453797111 87,
50, 1, 1.8323540040856016730685951188206672 67,
51, 1, 1.7232241635059648565686529764207080 01,
52, 1, -0.9781527985329186991592109734483528 88,
53, 1, -2.7555456184206517100676592235686257 48,
54, 1, -1.4608501803488669334996075122035108 51,
55, 1, -3.8894554692834160114500718918861821 29,
56, 1, -3.3586369987056210462128547078464180 23,
57, 1, 1.7370590999178299185246032720897346 74,
58, 1, 4.3851624241811961724124557804316282 27,
59, 1, 1.2616837560095648385072308883536607 03,
60, 1, -3.6671634157772321493951039883540943 26,
61, 1, -3.5584458900805175929349388752598315 48,
62, 1, 1.2654959366717231095122997430735267 70,
63, 1, 3.9009918390351310257813111093128100 04,
64, 1, -4.2100174690348142192419800267089158 30,
65, 1, -1.4208614094814435713232114721904508 77,
66, 1, 3.8205447151002043248979589407099410 89,
67, 1, 3.9950270855317095985981268313480541 11,
68, 1, -1.1833544160851252602384420242742635 31,
69, 1, -4.6731171082760667445654689799994230 27,
70, 1, -1.6589572310395417353845459729200229 05,
```

```
    71, 1, 3.735811098562083376606324236490763724,
    72, 1, 4.293379966109708512078668718459084630,
    73, 1, -0.023811363666440366754217805578264233,
    74, 1, 1.803836432305244708018676647043321282,
    75, 1, 1.775948032788597474862513081461656839,
    76, 1, -0.464242965337303747919150964662549086,
    77, 1, -1.945120712992527423068622738355770707,
    78, 1, -0.677051736494404976518524108541896567,
    79, 1, 1.599095293227147385195507922617252916,
    80, 1, 1.848319670862301089542256704589817673,
    81, 1, 0.114718342573754350510739641322288662};
xSandia = Table[sandia81x⟦i+2⟧, {i, 1, Length[sandia81x], 3}];
```

Start by using *Mathematica*'s built-in solver.

```
xMathematica = LinearSolve[A, b]
```

```
{-0.1044558957845920573040, -1.86118957003281196319, -1.72246357804553828948,
 0.335928566686036934373, 1.86296089880701384199, 0.549298777343197910959,
 -1.65687719865831647889, -1.71448160693834490957, 0.068312196999078767097,
 -4.23946027978257755345, -3.67418485721486384321, 1.111759246627820807735,
 3.207148543904416548120, 1.110782517145542428060, -3.691714857452917014680,
 -3.51867201595189856087, 1.614709043486123767160, 4.298665976998501067630,
 -3.757201781862994346930, -1.124172480722376442730, 2.027042297107821236260,
 1.958812797335875771590, -0.442840862954281433280, -3.486270124080628776270,
 -1.145798999538388837810, 3.654151502245383249400, 4.005701033800838323820,
 1.207613547548265756110, 1.960702164798509980680, 0.666690577258767909910,
 -0.426187910195241457073, -0.518206507290503285732, 0.967032808823420814715,
 2.506766957224375234960, 0.946988879588822986907, -1.352011928308374334730,
 4.604238066436250068840, 2.842329174087171180380, -1.273689738761294905590,
 -1.745566229293453237750, 0.490947894124781296819, 3.837729653367925829940,
 2.621810099949755447000, -2.964604961892276198270, -4.677911359946115157790,
 1.432831174874628780670, -0.666284164331340915970, -1.638006058495278933760,
 0.027504355373327512283 9, 1.832354004085601420270, 1.723224163505962859710,
 -0.978152798532921308813, -2.755545618420654624320, -1.460850180348869078620,
 -3.889455469283417252200, -3.358636998705621824690, 1.737059099917829809620,
 4.385162424181197750680, 1.261683756009564765760, -3.667163415777235922850,
 -3.558445890080520234900, 1.265495936671722664500, 3.900991839035132889520,
 -4.210017469034814888230, -1.420861409481442991070, 3.820544715100206326900,
 3.995027085531711076560, -1.183354416085126852540, -4.673117108276070021380,
 -1.658957231039542921630, 3.735811098562082846560, 4.293379966109710119450,
 -0.023811363666440033870 97, 1.803836432305245475310, 1.775948032788599536740,
 -0.464242965337302877141, -1.945120712992528590010, -0.677051736494406535791,
 1.599095293227147756800, 1.848319670862301435710, 0.114718342573754783384}
```

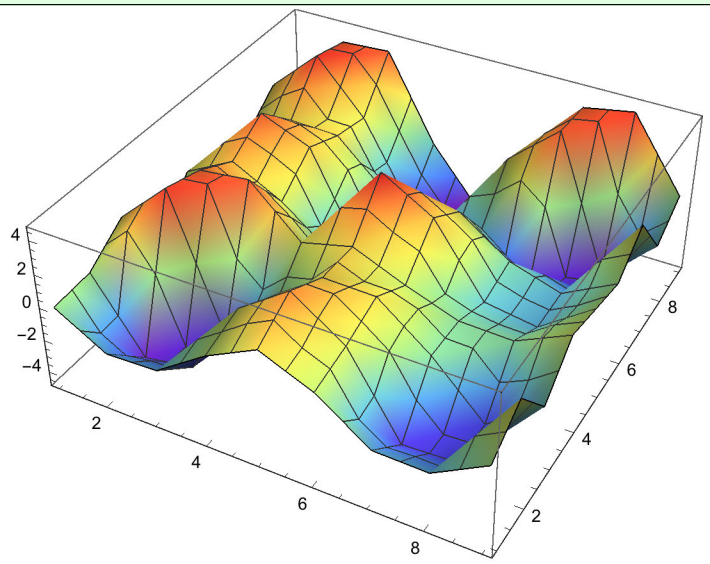
Compute the residual to check to see if this an accurate answer:

```
b - A.xMathematica
```

```
{0. × 10^-20, 0. × 10^-19, 0. × 10^-19, 0. × 10^-20, 0. × 10^-19, 0. × 10^-19, 0. × 10^-19, 0. × 10^-19,
 0. × 10^-20, 0. × 10^-19, 0. × 10^-19, 0. × 10^-19, 0. × 10^-19, 0. × 10^-19, 0. × 10^-19, 0. × 10^-19,
 0. × 10^-19, 0. × 10^-19, 0. × 10^-19, 0. × 10^-19, 0. × 10^-19, 0. × 10^-19, 0. × 10^-19, 0. × 10^-19,
 0. × 10^-19, 0. × 10^-19, 0. × 10^-19, 0. × 10^-19, 0. × 10^-19, 0. × 10^-19, 0. × 10^-19, 0. × 10^-19,
 0. × 10^-19, 0. × 10^-19, 0. × 10^-19, 0. × 10^-19, 0. × 10^-19, 0. × 10^-19, 0. × 10^-19,
 0. × 10^-19, 0. × 10^-19, 0. × 10^-19, 0. × 10^-19, 0. × 10^-19, 0. × 10^-19, 0. × 10^-19, 0. × 10^-19,
 0. × 10^-19, 0. × 10^-19, 0. × 10^-19, 0. × 10^-19, 0. × 10^-19, 0. × 10^-19, 0. × 10^-19,
 0. × 10^-19, 0. × 10^-19, 0. × 10^-19, 0. × 10^-19, 0. × 10^-19, 0. × 10^-19, 0. × 10^-19, 0. × 10^-19,
 0. × 10^-19, 0. × 10^-19, 0. × 10^-19, 0. × 10^-19, 0. × 10^-19, 0. × 10^-20, 0. × 10^-19,
 0. × 10^-19, 0. × 10^-19, 0. × 10^-19, 0. × 10^-19, 0. × 10^-19, 0. × 10^-19, 0. × 10^-20}
```

That looks like an accurate answer all right! Maybe we can make a 3D plot of it.

```
ans2D = Table[xMathematica[[8 i+j+1]], {i, 0, 8}, {j, 0, 8}];
ListPlot3D[ans2D, ColorFunction → "Rainbow"]
```



Let’s attempt the problem with mere 16-bit posits and see what happens.

```
xposit = linsolverp[A, b]; N[xposit, 3]
```

```
{-0.104, -1.86, -1.72, 0.336, 1.86, 0.549, -1.66, -1.71, 0.0683, -4.24,
 -3.67, 1.11, 3.21, 1.11, -3.69, -3.52, 1.61, 4.30, -3.76, -1.12, 2.03,
 1.96, -0.443, -3.49, -1.15, 3.65, 4.01, 1.21, 1.96, 0.667, -0.426,
 -0.518, 0.967, 2.51, 0.947, -1.35, 4.60, 2.84, -1.27, -1.75, 0.491,
 3.84, 2.62, -2.96, -4.68, 1.43, -0.666, -1.64, 0.0275, 1.83, 1.72,
 -0.978, -2.76, -1.46, -3.89, -3.36, 1.74, 4.39, 1.26, -3.67, -3.56,
 1.27, 3.90, -4.21, -1.42, 3.82, 4.00, -1.18, -4.67, -1.66, 3.74,
 4.29, -0.0238, 1.80, 1.78, -0.464, -1.95, -0.677, 1.60, 1.85, 0.115}
```

Here's the residual:

```
res1 = b - A.xposit; N[res1, 4]
```

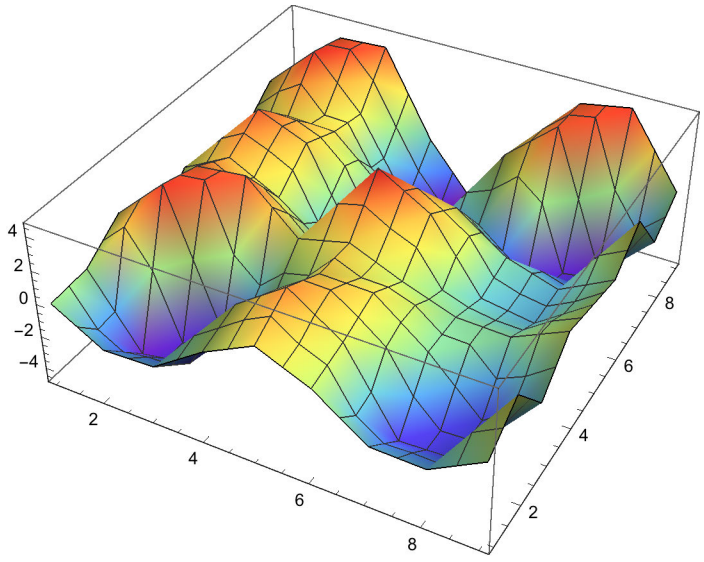
$$\{-7.760\times10^{-8},\ 1.954\times10^{-7},\ -6.711\times10^{-7},\ -5.065\times10^{-8},\ -2.789\times10^{-7},\ -9.717\times10^{-7},\ 8.363\times10^{-8},\ 3.854\times10^{-8},\ -4.809\times10^{-7},\ 4.502\times10^{-9},\ 1.537\times10^{-7},\ -5.900\times10^{-7},\ -6.337\times10^{-6},\ -1.665\times10^{-6},\ 6.080\times10^{-7},\ -2.879\times10^{-6},\ 1.615\times10^{-6},\ -1.357\times10^{-6},\ 1.580\times10^{-6},\ -1.645\times10^{-6},\ -2.195\times10^{-6},\ -7.384\times10^{-7},\ -5.032\times10^{-7},\ -2.913\times10^{-6},\ -9.544\times10^{-8},\ 2.298\times10^{-6},\ -7.868\times10^{-7},\ 1.256\times10^{-6},\ 2.819\times10^{-6},\ 1.695\times10^{-8},\ 8.823\times10^{-7},\ -6.859\times10^{-7},\ 2.481\times10^{-6},\ 1.288\times10^{-6},\ 9.893\times10^{-7},\ 8.376\times10^{-7},\ -1.551\times10^{-7},\ -3.911\times10^{-6},\ 7.724\times10^{-8},\ -1.542\times10^{-6},\ -1.319\times10^{-6},\ -3.277\times10^{-6},\ -2.309\times10^{-6},\ 5.355\times10^{-7},\ 1.317\times10^{-6},\ -1.016\times10^{-6},\ 7.095\times10^{-7},\ 1.173\times10^{-6},\ 4.541\times10^{-7},\ -5.658\times10^{-7},\ 1.442\times10^{-6},\ 6.201\times10^{-7},\ 6.867\times10^{-7},\ 1.128\times10^{-6},\ -1.538\times10^{-6},\ 3.872\times10^{-6},\ 1.849\times10^{-7},\ -2.187\times10^{-6},\ -1.572\times10^{-6},\ -3.755\times10^{-7},\ 1.368\times10^{-6},\ 9.545\times10^{-8},\ -5.743\times10^{-7},\ 3.368\times10^{-7},\ -5.425\times10^{-7},\ -6.082\times10^{-7},\ -8.762\times10^{-7},\ 1.115\times10^{-7},\ -3.538\times10^{-6},\ -4.208\times10^{-6},\ -4.453\times10^{-6},\ 1.370\times10^{-6},\ 8.976\times10^{-8},\ 9.086\times10^{-7},\ -1.693\times10^{-6},\ 7.760\times10^{-8},\ -1.263\times10^{-6},\ 2.127\times10^{-7},\ 3.414\times10^{-7},\ 4.640\times10^{-7},\ 3.289\times10^{-8}\}$$

If we use that to amend the answer, we get a result that matches the one supplied by Sandia, to the precision of 16-bit posits:

```
xposit = xposit + linsolverp[A, res1];
N[xposit, 3]
N[xSandia, 3]
ans2Dp = Table[xposit⟦8 i+j+1⟧, {i, 0, 8}, {j, 0, 8}];
ListPlot3D[ans2Dp, ColorFunction → "Rainbow"]
```

```
{-0.104, -1.86, -1.72, 0.336, 1.86, 0.549, -1.66, -1.71, 0.0683, -4.24,
 -3.67, 1.11, 3.21, 1.11, -3.69, -3.52, 1.61, 4.30, -3.76, -1.12, 2.03,
 1.96, -0.443, -3.49, -1.15, 3.65, 4.01, 1.21, 1.96, 0.667, -0.426,
 -0.518, 0.967, 2.51, 0.947, -1.35, 4.60, 2.84, -1.27, -1.75, 0.491,
 3.84, 2.62, -2.96, -4.68, 1.43, -0.666, -1.64, 0.0275, 1.83, 1.72,
 -0.978, -2.76, -1.46, -3.89, -3.36, 1.74, 4.39, 1.26, -3.67, -3.56,
 1.27, 3.90, -4.21, -1.42, 3.82, 4.00, -1.18, -4.67, -1.66, 3.74,
 4.29, -0.0238, 1.80, 1.78, -0.464, -1.95, -0.677, 1.60, 1.85, 0.115}
```

```
{-0.104, -1.86, -1.72, 0.336, 1.86, 0.549, -1.66, -1.71, 0.0683, -4.24,
 -3.67, 1.11, 3.21, 1.11, -3.69, -3.52, 1.61, 4.30, -3.76, -1.12, 2.03,
 1.96, -0.443, -3.49, -1.15, 3.65, 4.01, 1.21, 1.96, 0.667, -0.426,
 -0.518, 0.967, 2.51, 0.947, -1.35, 4.60, 2.84, -1.27, -1.75, 0.491,
 3.84, 2.62, -2.96, -4.68, 1.43, -0.666, -1.64, 0.0275, 1.83, 1.72,
 -0.978, -2.76, -1.46, -3.89, -3.36, 1.74, 4.39, 1.26, -3.67, -3.56,
 1.27, 3.90, -4.21, -1.42, 3.82, 4.00, -1.18, -4.67, -1.66, 3.74,
 4.29, -0.0238, 1.80, 1.78, -0.464, -1.95, -0.677, 1.60, 1.85, 0.115}
```

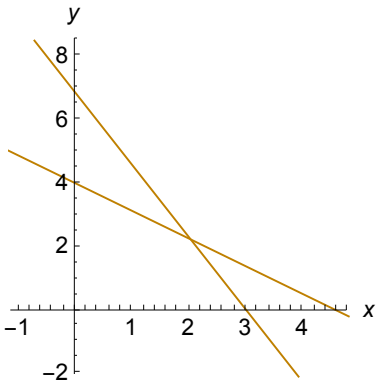


Any discrepancy in the answer is far too small to see. With the residual correction method, there appears to be no reason to use posit precision higher than 16-bit, which would obviously permit the solution of much larger problems than if we have to use 64-bit data for everything. The quire allows 16-bit posits to do the work of 64-bit floats.

12.6 The Time Complexity of Unum Linear Solvers

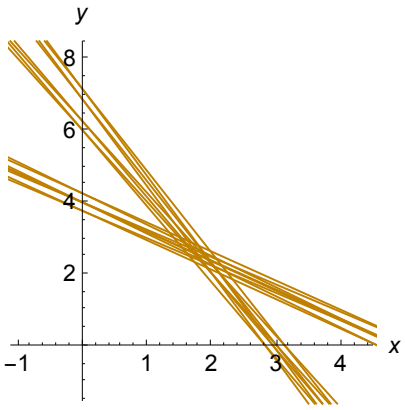
The A and *b* inputs to A *x* = *b* problems can be approximate in real applications. There is a theorem from interval arithmetic that says that finding the exact mathematical description of the entire

set of *x* values that solves the linear equations with interval bounds on the inputs is NP-hard. There is an easy way to understand why this is true: Every row of an exact matrix specifies a hyperplane in *n*-space, and the solution is the intersection of *n* hyperplanes. For example, a 2-by-2 system of linear equations simply represents the intersection of two lines.



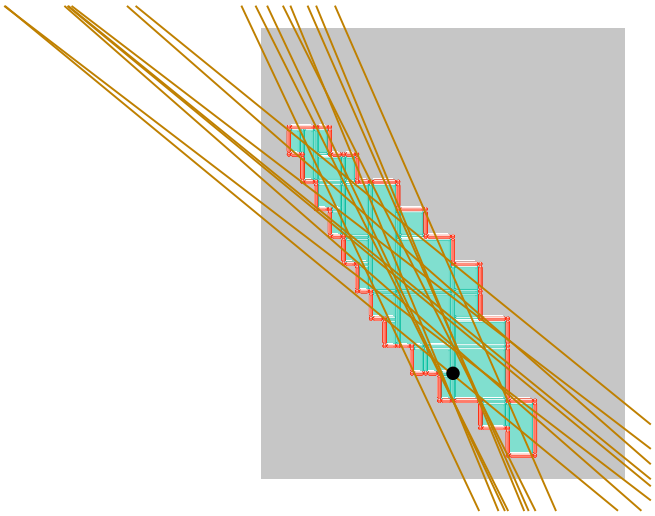
When the equations are specified in a number system, whether floats or posits, the exact intersection point has coordinates that in general are *not* floats or posits, though they are rational numbers. It would be coincidence if the denominators were powers of 2 such that they land in the numerical vocabulary. There is a unique *closest* point to the answer, found simply by rounding each element of the answer vector.

If the matrix coefficients are intervals instead of exact points, then a single row represents the intersection of *n* "hyperslabs," regions in *n*-space bounded by slabs. The only way to find that intersection is to examine every possible combination of minimum and maximum values for the bounds, or 2^n hyperplanes. A rigorous bound means finding the hyperplane with minimum and maximum of each of the n^2 matrix elements and *n* right-hand-side elements, a total of $2\left(n^2+n\right)$ cases. Even when *n* is as small as 2, the exact geometrical intersection representing the result can be laborious to compute, as the following figure indicates.



What about simply finding a particular solution *x* that seems to solve A *x* = *b*, and then perturbing *x* to find other solutions that work, iterating until we have found them all? This is one of the *ubox*-based methods, computing with multidimensional vectors that are either exact or an open interval one ULP wide for each component, or ubox. In the figure below, the black dot is a first guess produced by whatever solver you prefer (and checked to see that it really does solve the equa-

tions after rounding). That is an exact ubox in all dimensions, and the seed of an exploration. By testing the neighbor points to see if they also satisfy A $x = b$ (where "=" means interval intersection with the right-hand side vector), we can "paint" the interior of the complex shape to find all the uboxes that make up the solution set, shown in light green below. The red border represents a ubox that provably does *not* satisfy A $x = b$. That is, the exact matrix A times the ubox x results in intervals that do not intersect all components of b.



While this approach has the advantage that it is easily done on a massively parallel computer, and it is *much* simpler to program than to intersect hyperplanes with computational geometry, notice that in perturbing the x value by an ULP in each component means 3^n perturbations in each dimension, since each component can go down by an ULP, stay the same, or go up by an ULP. It works up to some value of n, but not a very large n. Like, for LINPACK with $n = 100$, having to test 3^{100} input vectors (uboxes) is clearly intractable. It doesn't matter whether the matrix is dense or sparse. At first, it looks like this is an impasse to getting reasonable time complexity for ubox-type solvers.

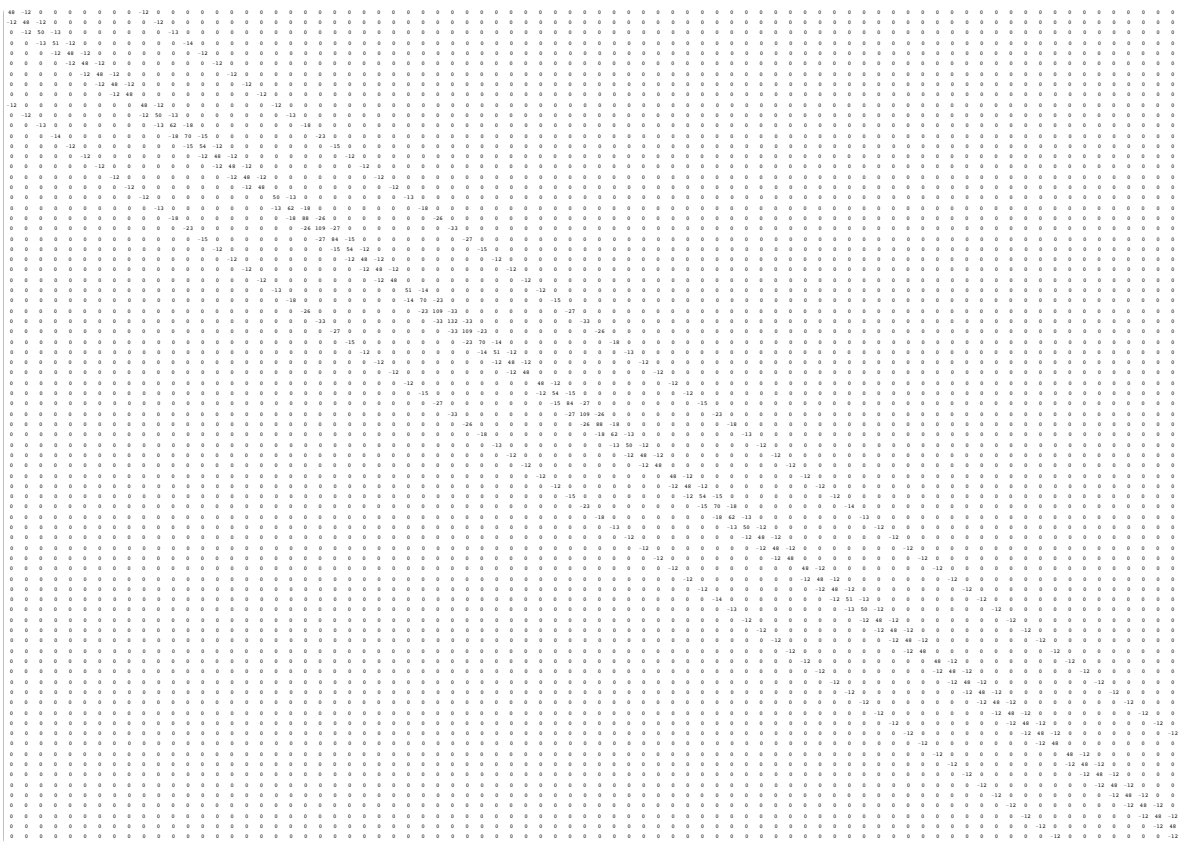
The LINPACK example leads to an insight, however: In Section 12.1, we perturbed the A matrix such that the exact answer matched a preconceived solution of all 1 values for the x_i. That's a form of *reverse error analysis*: The approximate solution to a problem can be viewed as the exact answer to a different problem, and if the different problem is only *slightly* different, then we might accept the answer. This argument dates back to classic 1950s papers by Wilkinson on numerical analysis, who created existence proofs for when some other problem is within a distance from the problem originally posed. But reverse error analysis does not actually construct the altered problem; it only states that it exists under certain conditions.

With unum arithmetic, we can **explicitly construct the altered problem for reverse error analysis.** Furthermore, this allows us to make the time complexity of perfect, bounded solutions as easy as it is for all current numerical methods for solving A $x = b$.

The original random matrix A for the LINPACK benchmark was altered so that the solution vector

was expressible as an exact point in *n*-space using the posit vocabulary. Every perturbation of that solution vector **fails** to solve the linear system! So we're done. There is only one "ubox" that solves the system, and it is exact in all dimensions and exactly 1 in every component. There are no ULP-wide ranges. This cannot be achieved with traditional interval arithmetic, incidentally; traditional interval arithmetic uses *closed* intervals of the form [*a*, *b*] where $a \le b$, and all $3^n - 1$ neighbors of the exact point will also yield a vector that touches *b* in every component, so they still create intractiable time complexity. With unum arithmetic, valids are *open* intervals, and do not include the endpoint that satisfies $A\,x = b$. That means any open-interval neighbor of the exact solution is part of that red boundary in the diagram above. The iterative method is $O(n^3)$ since the original Gaussian elimination (LU factorization) is $O(n^3)$ and the iterations are $O(n^2)$ since they depend only on evaluating the matrix-vector product using the quire and backsolving with U^{-1} and L^{-1} (the triangular matrices formed by Gaussian elimination), all of which require only $O(n^2)$ work. Hence, the time complexity of solving LINPACK is $O(n^3)$ for an *n*-by-*n* system, even if the iterative method is used to obtain a perfect, single exact solution. That result appears to be new.

However, for real problems, we do not generally have the option of tweaking matrix entries. The Sandia test data is a good example of that. Remember that after scaling by 120, the matrix was all integers:



That's because it results from expressing stencil operators like

$$x_{i,j} = \left(x_{i-1,j} + x_{i+1,j} + x_{i,j-1} + x_{i,j}\right)/4$$

We should regard the matrix A as exact an “untweakable” except in the case of LINPACK. That would seem to dash the argument of constructing the reverse-error-analysis problem.

Or does it?

Remember, Type III unums have the quire data type. When you try to solve $A\,x = b$ you find an approximate x, no matter whether you are using Gaussian elimination or Conjugate Gradient or Successive Over-Relaxation, or whatever your favorite method is. Call that $\tilde{x}$, a vector of posit elements that we hope result in $A\,\tilde{x}$ being close to b.

Then using exact quire dot products, we can compute $\tilde{b} = A\,\tilde{x}$. Exactly. The answer is not only $\tilde{x}$, but also the explicitly-constructed problem that reverse error analysis requires:

$A\,x = \tilde{b}$ is a problem close to the original problem, $A\,x = b$,
and the solution is **exactly** $x = \tilde{x}$.

There is no exponential complexity to bounding the solution. The complexity is whatever the traditional matrix solver method has for complexity, and then the perturbed version of the original problem *becomes part of the solution*. That is, both $\tilde{x}$ the altered $\tilde{b}$ are computed and treated as output for the problem. The user can then decide if $\tilde{b}$ is sufficiently close to the requested b. This is a new paradigm for the solution of $A\,x = b$ problems, and it is made possible only because of the quire data type. □

Sections on linear solvers were supported in part by the DARPA TRADES Program, Contract #HR0011-17-9-0007.