Well-Rounded: Visualizing Floating Point Representations

Neil Ryan, Katie Lim, Gus Smith, Dan Petrisko

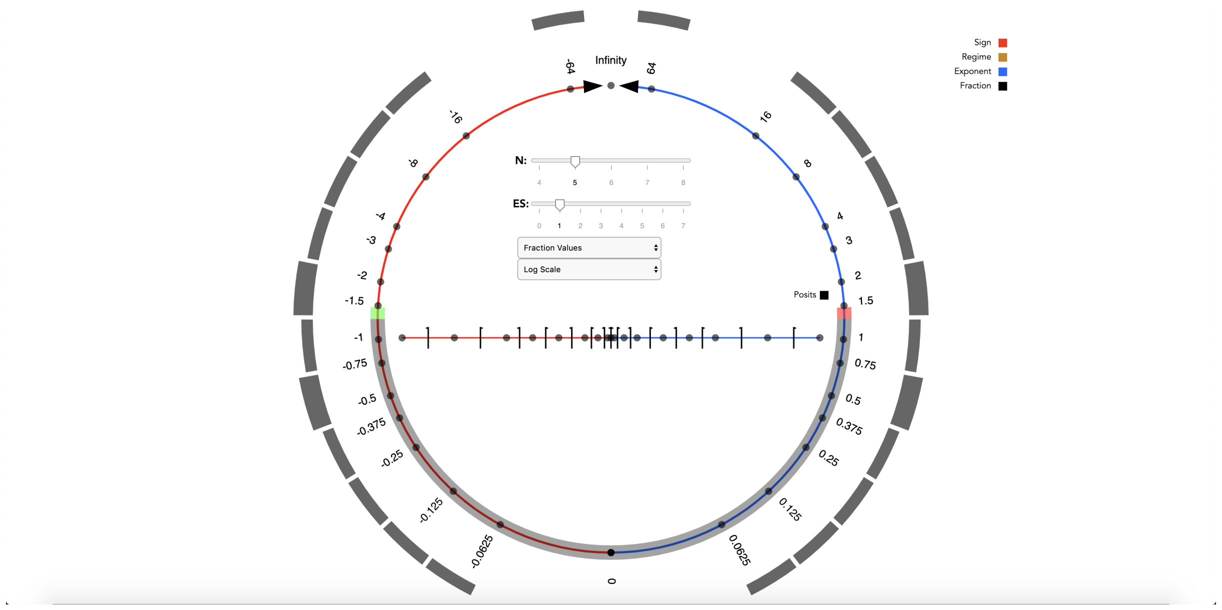


Fig. 1: A snapshot visualization of posit properties

Abstract— While common numerical datatypes such as IEEE-754 are familiar to programmers, alternative datatypes can provide dramatic benefits in certain application domains. However, lack of familiarity is a major obstacle to their adoption. This visualization allows users to explore posits, one such alternative datatype.

1 Introduction

Machine learning and AI have brought about revolutions in countless fields, revolutions that require incredibly high power and memory bandwidth. Computer architects have responded by rethinking even the most basic design decisions, such as how decimal numbers are represented in hardware. As a result, the age-old IEEE 754 floating point types (floats) have several modern competitors—such as the posit [5]—which are smaller, more energy-efficient, and have better numerical behavior.

Numerical datatypes specify how a computer represents real numbers in hardware. Real numbers, which are continuous and infinite, cannot be fully represented in a finite number of bits. However, to build an efficient computer which can operate over real numbers, one generally needs to represent real numbers in a fixed number of bits, for example, 32 or 64. Thus, a big part of the decision of how to represent real numbers is in deciding *which* real numbers you will represent. Additionally, once you know roughly which numbers you would like to represent, you must find a concise way to express these numbers, in as few bits as possible.

For decades, the computing world has relied on the IEEE 754 floating point standard [1]. As shown in figure 2, this standard represents real numbers as a sign bit, a number of of exponent bits, and a number of fraction bits. Roughly, the value of the number represented is:

$$\text{sign} * 2^{\text{exponent}} * 1.\text{fraction}.$$

This encoding is fairly general, and, with enough bits, can represent a very useful range of numbers. As a result, the standard became widely adopted, and has been the de facto floating point representation since its inception. Today, it is present in almost all processors, down to small microprocessors.

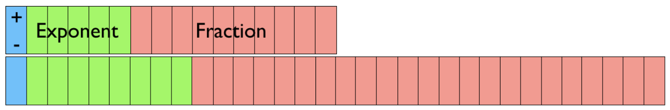


Fig. 2: IEEE 754 16-bit half-precision and 32-bit single-precision floating point types. Sign bit is in blue, exponent is in green, fraction is in red

Despite dominating the numerical datatypes space, IEEE 754 is not without its shortcomings. As the specification indicates, the type is incredibly complex, and with caveats such as *denormal numbers* requiring equally complex hardware (and sometimes just being left to implement in software) [6]. In addition, standard IEEE 754 floating point types use far more bits than they need to for many applications. That is, with a well-designed type, we can often use half as many bits with little drop in performance. In fact, Google's TensorFlow framework advocates for the use of the `bfloat16` type, which is simply the IEEE 754 32-bit single-precision type with 16 bits of the fraction cut off. They claim this type is often able to achieve the exact same training accuracy when training machine learning models [4], while saving half of the space!

Despite its shortcomings, it is only recently that researchers have begun turning a critical eye towards the IEEE standard. A number of datatypes have been created in response; the `bfloat16` is one example of those types. In this work, however, we focus on the posit.

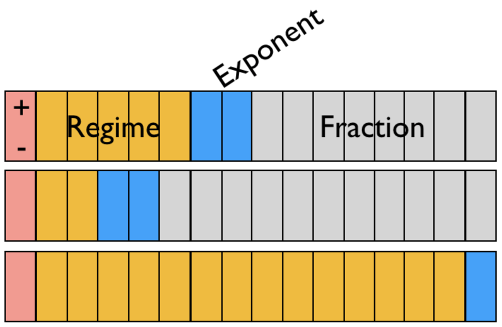


Fig. 3: Posits, with $n = 16$ and $es = 2$

The *posit* is a datatype originally presented by John Gustafson [5]. Posits can be of variable length n, with common settings of n being 8 and 16. Posits use the fields we are familiar with from IEEE 754: sign, exponent, and fraction. However, the exponent field can be of variable size—we define its maximum size in the parameter es. In addition, posits use a field called the *regime:* a field encoded as a run of 1s or 0s and terminated by a 0 or 1 respectively. The number of repeated 0s or 1s corresponds to k, which is used to calculater another multiplicative scaling factor, in addition to the 2^{exponent} factor. The resulting real number is calculated as

$$\text{useed}^k * 2^{\text{exponent}} * 1.\text{fraction}$$

where

$$\text{useed} = 2^{2^{es}}.$$

Note that the parameter es not only affects the maximum number of exponent bits, but the base number of the regime's scaling factor as well.

Posits are shown in figure 3. The regime is variable length; as a result, some posits may have partially or fully truncated fractions or exponents.

Posits have become a topic of interest for a number of reasons. First, from a numerical standpoint, the high *dynamic range* they are able to represent due to the additional useed^k scaling parameter makes them a great fit for applications which need high dynamic range, but not precision—for example, machine learning. A recent paper trained a neural network using 8-bit posits, showing competitive performance between an 8-bit posit and 32-bit float [2]. From the hardware perspective, hardware implementations of posit arithmetic units have shown to be competitive with existing floating-point units (FPUs) [3].

While posits show substantial promise in the datatypes space, it is often shocking to realize how little average programmers understand about any numerical representation of real numbers, let alone a new one such as the posit. To address this knowledge gap, we have created a visualization that allows the user to explore the posit, with the goal of getting an understanding for exactly how real numbers are represented in hardware. Specifically, we singled out three main goals:

1. Visualizing how a datatype's parameters affect the datatype,

2. Visualizing the rounding behavior of the datatype, and

3. Visualizing the distribution of the datatype's values in the infinite space of real numbers.

We seek to create a visualization tool that is useful to datatype researchers and general programmers alike. By spreading knowledge about new datatypes, we hope to make programmers think about their datatype choice, and perhaps to pave the way for datatypes like the posit to enter common usage.

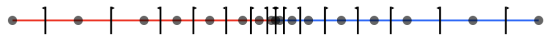


Fig. 4: Number Line

2 Related Work

The main inspiration for this work is John Gustafson's original posit paper [5]. In this work, Gustafson provides several visualizations to demonstrate the advantages of the data type. The basis of our visualization is the paper's *projective real number line* figure, which places posits on a number line bent into a circle, using the same point for $\pm\infty$. This makes sense for posits, as posits (unlike IEEE floats) use a single representation for $\pm\infty$. Using this as a base, we extend the visualization in a number of useful ways, as we will describe throughout the rest of the paper.

3 Methods

Our visualization is a bit unique in that we do not have a concrete dataset. Instead, we generate it dynamically using the posit value equation and the parameters n and es. This means, in addition to the above goals related to posits, some of our design decisions focused around controlling and summarizing details to the user for larger values of n while still providing a helpful visualization for users at lower values of n.

3.1 Encodings

As in Gustafson's work, we display values on a circle to demonstrate the wraparound property that exists in binary formats. We also bisect the range so that all positive values are on the right side and negative values are on the left side. We also provide a secondary visualization which is a number line populated by a slice of the value space, selected by brushing.

3.1.1 Positional

We use positional encoding to represent values of datatypes. In the original encoding, points are equally spaced along the circle—what we have labeled the "ordinal" scale. As one of our goals was to capture the distribute of the posits values, we added the ability to scale the spacing between points based on the actual values of the posits—thus, in addition to ordinal scaling, we support linear and log scaling as well.

Ordinal view shows the user the binary representation, evenly dividing the range. It also allows us to show the encodings for both infinity and negative infinity. (The number line does not support infinities, so they are disallowed as brush selections). This is most useful for user selection of values to explore the encoding space of the data type.

In linear view, the negative minimum and positive maximum values are placed at the top of the circle. For that side of the circle, the values are linearly interpolated based on their decoded value between the maximum and zero at the bottom of the circle. This allows users to see the distribution of values in the space.

Finally, we support log view. Like linear view, the it view the negative minimum and positive maximum values as the maximum for that side of the circle. Values are interpolated between the maximum and zero according to the log of the value. This distribution better represents the intention of the data types – to provide reasonable accuracy at a wide range of values.

This works fairly well for smaller values of n and es. However, because posits can express a large dynamic range, points tend to clump together around 0 at larger values of es and n, we also provide histogram representation for better summary of details in this setting. Histogram bars are positioned along the circumference of the circle. Any points in the region of the circle that the bar spans are counted as part of that bin. The position and width of the bar represent the range of values it contains. The width changes based on the setting of n, which controls the total number of points on the circle. Especially interesting are sections without a histogram bar, indicating a complete lack of coverage by the chosen datatype.

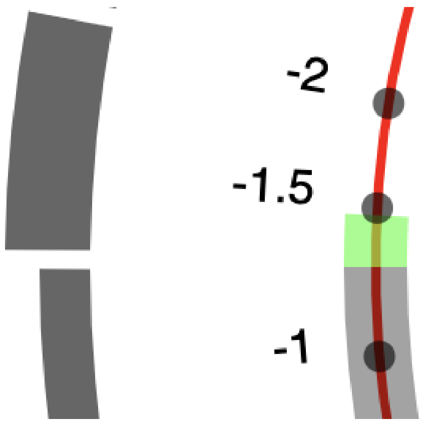


Fig. 5: Circle Segment

We provide a separate number line, seen in figure 4, which does not appear in the original visualization. The number line allows for focusing on a range of data, and provides additional information not shown on the circle—specifically, the rounding information. The data depicted on the number line is selected by brushing. Position of points on the number line has the same spacing options as points on the circle, with the maximum and minimum points as the maximum and minimum points in the range rather than zero and either the negative minimum or the positive maximum. We also use ticks with small arrowheads drawn on the number line to show rounding "tie points" and the direction they round in to visualize rounding behavior, which was another one of our goals and was not part of the initial visualization.

3.1.2 Size

The height of the histogram bar is related to number of values in the bin, although we made a couple of design decisions that are worth noting about the size.

First, the value that the maximum-sized bar represents changes based on the maximum count for that setting of *n*, *es*, and distance scaling. In particular, this means that comparing bars across settings of n and es is not a valid comparison. Additionally, for settings that produce bins with small numbers of, we set the maximum-sized bar to be smaller than the full available height, because otherwise the bars felt quite intrusive

Second, we set a minimum size for a non-zero bin. We did this, because for certain settings like linear scaling with high n or es, 100s of values were concentrated in one bin, so bins with one item in them were almost imperceptible slivers. Ultimately, this minimum size is still small enough that users can see which ranges have a higher density of values (and therefore higher precision).

3.1.3 Color

We use color in two ways. The first use is to reinforce the bisection of the circle into positive and negative values. We also encode the ordinal representation by bitfield: sign, regime, exponent and fraction. A legend on the side clarifies this encoding. We chose the color pallete based on the scheme in Gustafson's work. The two uses of colors here do not interfere because they are used in two distinct visual regions of the graphics: comprising the circle and surrounding it.

3.2 Interactivity

3.2.1 Sliders

One of our main goals with interactivity was to show how the parameters (n and es) affect the datatype. To do this, we provide two sliders for modifying the parameters which define a set a posits – n and es. N is the total number of bits in the posit, while es is used to determines the number of regime bits and therefore range. When a user slides either n or es, value dots are either added, subtracted or translated.

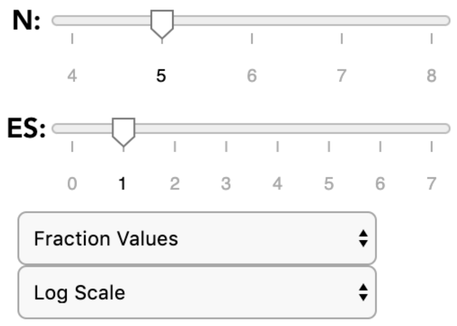


Fig. 6: Controls

We animate the dots so that users can visually track differences in the ordinal, linear or log magnitude as the precision or range changes.

3.2.2 Buttons

We use buttons for actions that dramatically change the visualization. The first button switches between fractional value and bitstring for each dot on the circle. Each of these modes is useful for understanding the connection between position and value. The second button switches between log scale, linear scale and ordinal scale. Ordinal scale is the default for two reasons: it corresponds to the original graphics in Gustafson's work and it instantly informs the user that they will be examining different interpretations of binary values. Switching to linear and log scales more clearly shows the differences in value distribution. Users will most likely toggle between these modes as their main method of exploration as they try to learn the relative advantages and disadvantages of data types.

3.2.3 Brushing

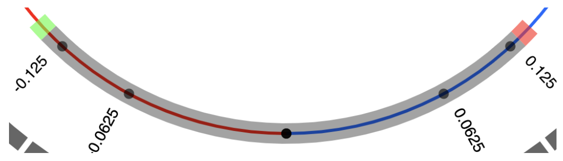


Fig. 7: Brush

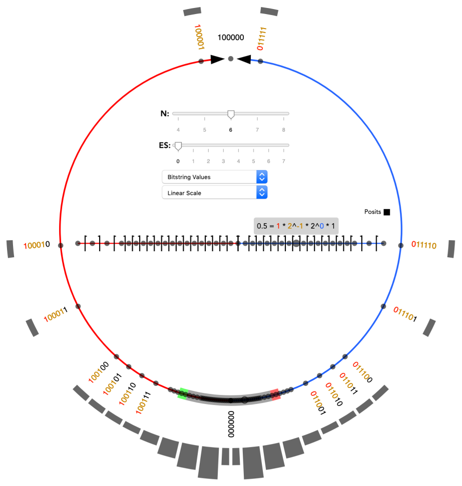
Users can brush along the circle to select a range of values. This provides a way to closer examine a trimmed range of values. The brush has two handles to select a range, but is unable to include positive or negative infinity. Users can also drag the middle of the brush stroke in order to shift the selection without changing its magnitude.

3.2.4 Tooltips

When a user hovers over a dot, a tooltip pops up showing the formula which derives the value. The formula is color-coded according to the bitfield color encoding described in section **??**. While the formula is the same for each dot, this visualization is intended to explain the fundamentals of data types. This redundant encoding reinforces the connection between the data type encoding and its value.

4 Discussion

Most of the audience feedback that we received was along the lines of "cool" or "nice", rather than actionable feedback. However, we can observe the reactions of the audience to different elements of our visualization. Generally, audience members initially acted overwhelmed by the available information. After a brief overview of IEEE-754/posits, they typically moved from the poster to the live visualization. This implies that the interactive elements of the visualization are most important. The most used operations of the visualization were (in order):


N: 6

ES: 0

Bitstring Values

Linear Scale

0.5 = 1 * 2^-1 * 2^0 * 1

Fig. 8: Our visualization with linear scaling, showing the increased precision of posits around 0.

1. Changing N and es
2. Toggling between log and linear mode
3. Dragging the brush

From these findings, we hypothesize that reducing the initial amount of available information in the visualization would be valuable. Having a short overview of posits hover over the visualization upon loading would be more helpful than the text-based description currently above it, which users tended to simply scroll past (or completely ignore, since the user before them had already scrolled past). Ordinal mode is most useful as a default because it provides some context for how binary values are translated into magnitudes. However, after its initial transformation to a log or linear scale, it is less valuable and perhaps should not be given the same control priority as other magnitude modes. Figure 8 gives an example of linear scaling mode.

5 Future Work

While this visualization highlights differences between float16, bfloat16 and posits, our approach could be applied to any variable range or precision datatype. A natural extension would be to add support for the full range of IEEE-754 float ranges, from 16-128 bits. Once a numerical library is written, little visualization code needs to change in order to support these alternate formats. In order to prevent an overwhelming amount of information being presented, we would most likely provide a checklist where users could select which datatypes to include in the visualization at a single instant.

An additional visualization which would be nice to have is the effect of different rounding modes on the number line. Currently, we choose a single reasonable rounding mode for each type, but formats such as IEEE-754 offer several rounding modes to mitigate different types of aggregate error. A limitation of current rounding visualization is that we show the rounding boundary itself, but do not indicate the reason for the placement. If we show all rounding modes simultaneously, one option would be to use color or pattern to indicate which rounding boundary applies to which rounding mode. Otherwise, a simple drop-down menu or radio button could select the rounding mode, adjusting the number line as necessary.

References

[1] Ieee standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, Aug 2008.

[2] Z. Carmichael, H. F. Langroudi, C. Khazanov, J. Lillie, J. L. Gustafson, and D. Kudithipudi. Deep positron: A deep neural network using the posit number system. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1421–1426. IEEE, 2019.

[3] R. Chaurasiya, J. Gustafson, R. Shrestha, J. Neudorfer, S. Nambiar, K. Niyogi, F. Merchant, and R. Leupers. Parameterized posit arithmetic hardware generator.

[4] Google. Using bfloat16 with tensorflow models, Apr 2019.

[5] I. Y. John L Gustafson. Beating floating point at its own game: Posit arithmetic. Supercomputing Frontiers, March 2017.

[6] E. M. Schwarz, M. Schmookler, and S. D. Trong. Fpu implementations with denormalized numbers. *IEEE Transactions on Computers*, 54(7):825–836, July 2005.