Master Thesis

Unums and Posits: A Replacement for IEEE 754 Floating Point?

by

Andreas Schärtl

Matrikel-Nr.: 21915251

Supervision:
Prof. Oliver Keszöcze

2021-08-03

Unums and Posits: A Replacement for IEEE 754 Floating Point?

Masterarbeit im Fach Informatik

vorgelegt von

Andreas Schärtl

geboren am 01. Mai 1992 in Nabburg

angefertigt am

Lehrstuhl für Informatik 12
Hardware-Software-Co-Design

Department Informatik
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer: **Prof. Oliver Keszöcze**
Abgabe der Arbeit: **03. August 2021**

Erklärung. Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Declaration. I confirm that I have written this thesis unaided and without using sources other than those listed and that this thesis has never been submitted to another examination authority and accepted as part of an examination achievement, neither in this form nor in a similar form. All content that was taken from a third party either verbatim or in substance has been acknowledged as such.

Andreas Schärtl, Erlangen, 2021-08-03

Arithmetic with real numbers is a foundations of computing and programming. While the most common implementation of real arithmetic, IEEE floating point, has proven itself practical for many applications, floats leave some things to be desired. Hard to understand rounding rules and many special cases one has to keep in mind have lead to various problems. Novel unum arithmetic promises to solve some if not all problems introduced by float arithmetic. Of particular interest are (1) the recently introduced posits as drop-in replacements for floats and (2) valids as a new iteration on traditional interval arithmetic.

This thesis first reviews developments in computer arithmetic on the reals, in particular related to recent developments on unum arithmetic. We contribute with a detailed discussion and definition of the valid number format. We also provide a flexible implementation of both the posit and valid formats which opens the gates for various interesting experiments. In final evaluation, we compare accuracy and usability of both float and new unum arithmetic based on various benchmarks and applications from different domains.

Contents

1 Introduction

If there is one thing computers should be good at, then it has to be numbers. Performing accurate and fast arithmetic is what allows digital computers to outclass the long passed "human computers" of previous times [1]. It will be hard to find anyone interested in going back to pen and paper. But reality is that implementing arithmetic on computers can be anything but trivial.

Computer Arithmetic Pitfalls

Computer arithmetic is about implementing arithmetic on digital computers. But today's computer arithmetic is riddled with hidden traps and surprises. Programmers have to get used to many pitfalls or often choose to ignore them. In this introduction, we look at contemporary definitions and implementations of computer arithmetic. While the introduced formats (integers, fractions, fixed points and floats) will be familiar to any reader, the focus here is on pitfalls and compromises introduced by these formats.

1.1 Unsigned Integers: You can Only Have so Many Bits

We will start this discussion by looking at the natural numbers $\mathbb{N}$, that is numbers

$$0, 1, 2, \ldots, \tag{1.1}$$

as natural numbers make up the building blocks from which we can then construct more advanced number systems. Writing down some quantity n with a series of N digits

$$n = d_{N-1} \ \cdots \ d_1 \ d_0 \tag{1.2}$$

is familiar to any reader. In fact, "positional number systems" that split up quantities or numbers into individual digits have been around for pretty much as long as there has been human civilization [2, pp .195]. Formally, the value of n is then given by sum

$$n = \sum_{i=0}^{N-1} d_i \, B^i \tag{1.3}$$

where B is the base of the number system [3, pp. 8]. In day to day interactions, we will usually pick $B = 10$. For example, we get

$$1234 = 4 \cdot 10^0 + 3 \cdot 10^1 + 2 \cdot 10^2 + 1 \cdot 10^3. \tag{1.4}$$

Given an arbitrary number of digits, we see that we can use this notation to write down any quantity $n \in \mathbb{N}$.

Binary computers are better at working with $B = 2$ where digits are limited to either `0` or `1`. But implementing natural numbers on computers is not as simple as choosing base $B = 2$. When implementing some given number system on digital computers, we face various restrictions. In particular, memory size and computation time are realities we have to deal with. Perhaps the main differences when moving from math to computer arithmetic are limitations and errors.

Fixed Width

Computers mean compromise. A first compromise computer designers may make is that width N of a given number is not arbitrary. Rather it is fixed to a certain number of bits. Many programmers will be familiar with the C programming language and its various types

$$\texttt{uint8\_t}, \texttt{uint16\_t}, \texttt{uint32\_t}, \texttt{uint64\_t} \tag{1.5}$$

that define natural numbers 8, 16, 32 or 64 bits in size [4]. This is no frivolous choice made by programming language designers. Rather it reflects the underlying computer architecture optimized for fixed width numbers [5, A1-40]. Software libraries that work with numbers of arbitrary length do exist [6]. But emulating arithmetic arithmetic in software is slow compared to optimized hardware. With this in mind, we can now define the unsigned integers used to represent $\mathbb{N}$ on computers.

Defintion 1. *An unsigned integer n of type U_N is a bit string*

$$n = d_{N-1} \cdots d_1 \, d_0 \tag{1.6}$$

of fixed length N that represents some value in $\mathbb{N}$, The value of n is given by

$$n = \sum_{i=0}^{N_1} d_i \, 2^i \quad \textit{[3, pp. 21]}. \tag{1.7}$$

Overflow

Fixing width N limits the range of a given unsigned integer type. In particular, limiting N can result in overflow. This is the case when the result of some operation requires more digits than available by the given type. Overflow is a serious problem as it can be difficult to diagnose yet be the cause of dangerous malfunctions [7]. Programmers have learned to live with this constraint. Even modern programming languages expose overflow as a leaky abstraction to the developer. As such, overflow remains a concern that requires serious attention or elaborate analysis tools [8].

Computer arithmetic deals with values of fixed size N. In consequence, even a simple format such as U_N has dangerous pitfalls. It is the first of many compromises we have to keep in mind.

1.2 Signed Integers: Duplicate Value Confusion

The natural numbers $\mathbb{N}$ include zero, but all values are non-negative. To represent the integers $\mathbb{Z}$, that is the numbers

$$\ldots, -2, -1, 0, 1, 2, \ldots, \tag{1.8}$$

all we have to do is to extend a given natural number with an explicit sign. We add either a plus (sometimes omitted) or minus sign to state that number is positive or negative. Implementing integers on computer hardware could be as simple as adding an additional sign bit. `1` indicates a negative and `0` a positive value [3, p. 54].

Defintion 2. *A sign-magnitude integer k of type M_N is a bit string*

$$k = d_{N-1} \cdots d_1 \, d_0 \tag{1.9}$$

of fixed length N that represents some value in $\mathbb{Z}$ [3, pp. 54]. Absolute value $|k|$ is given by

$$|k| = \sum_{i=0}^{N-2} d_i \, 2^i \tag{1.10}$$

and value of k is given by

$$k = (-1)^{d_{N-1}} \, |k| \tag{1.11}$$

Above definition really just sticks an additional sign bit to the familiar unsigned integers U_N. For example, consider the following M_4 integers

$$k_0 = \texttt{0101} = +101_2 = 5 \tag{1.12}$$
$$k_1 = \texttt{1101} = -101_2 = -5 \tag{1.13}$$

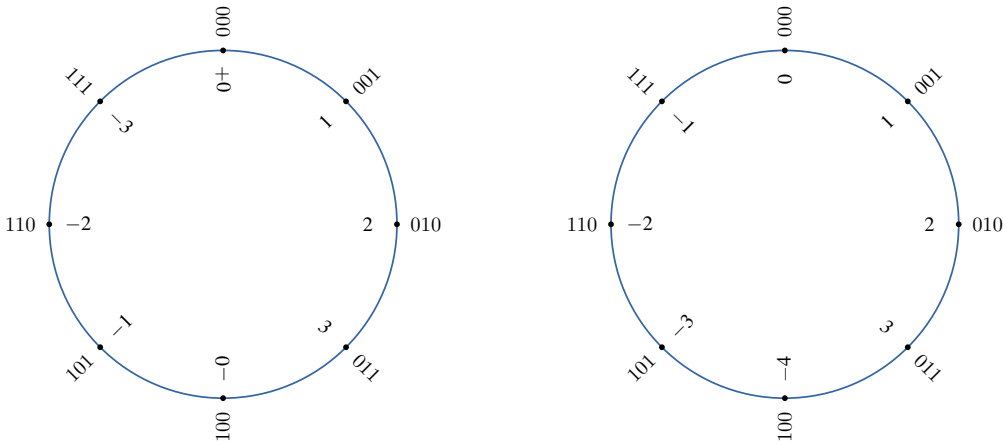
which have the same absolute value but different sign. Figure 1.1a shows the range of values provided by the M_4 type. Differentiating between positive and negative values with a dedicated sign bit is intuitive. But it also has various drawbacks.

Redundant Bit Patterns

One problem that arises from the M_N format is that it contains two representations of zero. For example, both

$$\texttt{0000} \text{ and } \texttt{1000} \tag{1.14}$$

are valid representations of zero in the M_4 format. This particular case is the first example for a whole class of problems related to computer arithmetic, namely redundant values. As we are already constrained in the range of values we can represent, we should at least be economical with the use of all N bits. We should not waste bit patterns on redundant values.



(a) Values provided by the M_4 type, a sign-magnitude integer with one sign bit and three bits magnitude.

(b) Values provided by the I_4 type, a signed integer with one sign bit. Negative values are encoded with the two's complement.

Figure 1.1: Two ways of representing signed integers with a total of four bits. While the simple sign-magnitude form might be more intuitive at first, it results in two representation of zero which makes the format hard to work with.

Two's Complement

Redundant values can make it difficult to define equality and arithmetic. As algorithms needs to consider more special cases, complexity grows [3, pp. 54]. Various solutions to this particular problem been proposed. Today the most common solution in use is the so called two's complement.

Defintion 3. *Given some unsigned integer n, the two's complement* $\text{Twos}(n)$ *of n is given by*

$$\text{Twos}(n) = (\neg n) + 1 \tag{1.15}$$

in unsigned integer arithmetic where $\neg n$ *is the bitwise negation of n [3, pp. 55].*

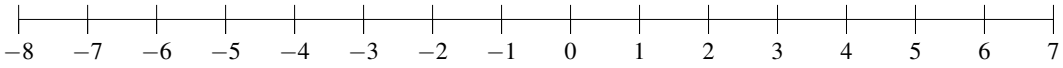
With the two's complement in hand we can define signed integers. They are today's canonical format for representing values in $\mathbb{Z}$ in computer arithmetic.

Defintion 4. *A signed integer k of type* I_N *is a bit string*

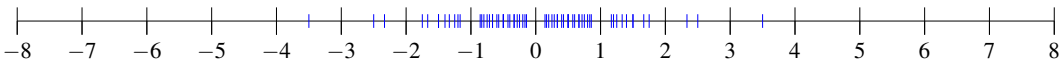
$$k = d_{N-1} \cdots d_1 \, d_0 \tag{1.16}$$

of fixed length N that represents some value in $\mathbb{Z}$*. If sign bit* d_{N-1} *is* `0`*, then k is positive and its value is identical to the* M_N *integer with the same pattern. If the sign bit is* `1`*, then k is negative and its absolute value is the two's complement* $\text{Twos}(k)$ *interpreted as an unsigned integer [3, pp. 55].*

Signed integers have only one representation of zero and arithmetic is easy to implement in both software and hardware. The two's complement solves our problem in this



(a) Range of the signed I_4 type. Integers only represent full integer stops on the number line and each step is evenly spaced.



(b) Range of a $R_{3,3}$ fraction which uses three bits for both numerator p and denominator q. Blue lines indicate some fractional value $x \notin \mathbb{Z}$ not representable with plain integers.

Figure 1.2: Comparing a plain signed integer with a fractional type. While the integer is distributed with even steps on the number line, the fractional type has most values clustered close to zero.

particular case. Still, it was a point worth investigating as redundant bit patterns come up in various formats, confusing implementors and users alike. We will see that the more complicated the number format, the harder it is to avoid duplicate patterns.

1.3 Integer Fractions: Jagged Accuracy

Integers can already get us very far, but for numeric applications, we need to be able to represent fractional values as well. Notice the gaping holes in Figure 1.2a which plots the range of the I_4 type. As of now, we can only represent fixed steps with nothing between. What we are missing is so called dynamic range. Our first attempt to solve this problem will be to use fractions, that is we will store fractional value

$$x = \frac{p}{q} \tag{1.17}$$

split up into nominator p and denominator q. Coupled with a dedicated sign bit, this allows us to represent ratios and not just integers.

Defintion 5. *An integer fraction x of type $R_{N,M}$ is a bit string*

$$x = s \underbrace{p_{N-1} \cdots p_1\ p_0}_{\textit{Nominator } p} \underbrace{q_{M-1} \cdots q_1\ q_0}_{\textit{Denominator } q} \tag{1.18}$$

of fixed length $1+N+M$ that represents some value in $\mathbb{Q}$. Bit string x is made up from sign bit s, unsigned integer numerator p and unsigned integer denominator q. The value of x is given by

$$(-1)^s \frac{p}{q} \quad \textit{[3, pp. 171]}. \tag{1.19}$$

Jagged Accuracy

Representing $\mathbb{Q}$ with fractions seems promising, but introduces a new problem, jagged accuracy. Figure 1.2b illustrates what exactly we mean by jagged accuracy. Instead of an even distribution of values on the number line, most of the types' dynamic range clusters close to zero. The steps between each discrete point on the number line differ in size.

Integer fractions have applications in cryptography as encryption often deals with finding divisors for big numbers. They also see use in number theory [3, pp. 177]. For general purpose computations however, integer fractions are seldom used. Even so, introducing them is useful because they introduce the concept of jagged accuracy. A property one has to keep in mind when working with such arithmetic types.

1.4 Fixed Point Numbers: Incompatible Magnitudes

Fixed point numbers are an alternative to integer fractions. Fixed point numbers do not store rational values as actual fractions with nominator and denominator. Rather, they use a representation that separates integer and fractional part. Instead of writing

$$\frac{1234}{100}, \tag{1.20}$$

we can use the more succinct representation

$$12.34 \tag{1.21}$$

which uses a decimal point. This notation translates nicely to the binary world of computer arithmetic.

Defintion 6. *A fixed point number x of type $Q_{N,M}$ is a bit string*

$$x = \underbrace{k_{N-1} \cdots k_1\, k_0}_{\textit{Integer Part } k}\ \underbrace{q_0\, q_1 \cdots q_{M-1}}_{\textit{Fraction } q} \tag{1.22}$$

of fixed length $N+M$ that represents some value in $\mathbb{Q}$. Bit string x is encoded using the two's complement and consists of integer part k and fraction q. If the most significant bit (sign bit) of x is 0, the value of x is positive and given by

$$x = \sum_{i=0}^{N-1} k_i\, 2^i + \sum_{i=0}^{M-1} \frac{q_i}{2^{i+1}}. \tag{1.23}$$

If the sign bit is 1, value x is negative and given by applying above Equation 1.23 to the two's complement of bit string x [3, pp. 183].

Example of Fixed Point Value

Maybe intimidating at first, Equation 1.23 merely splits up the bit string into integer part k and fraction q. As an example, consider fixed point number x of type $Q_{4,4}$, viz.

$$x = \texttt{0 0101 0110}. \tag{1.24}$$

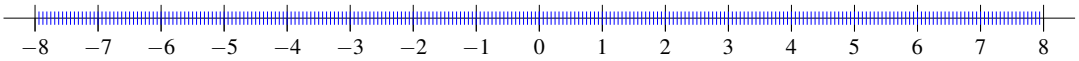


Figure 1.3: The $Q_{4,4}$ data type. Unlike the fractional type, values are evenly spaced.

The value of x is then given by

$$x = \underbrace{1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0}_{\text{Integer Part } k} + \underbrace{\frac{0}{2^1} + \frac{1}{2^2} + \frac{1}{2^3} + \frac{0}{2^4}}_{\text{Fraction } q} \tag{1.25}$$

which of course is very similar to how we encode decimal fraction such as

$$12.34 = 1 \cdot 10^1 + 2 \cdot 10^0 + \frac{3}{10^1} + \frac{4}{10^2}. \tag{1.26}$$

Advantages of Fixed Point

Implementing arithmetic for fixed point numbers only requires rudimentary integer operations. As such, software that implements fixed point numbers can be fast and efficient. Fixed point arithmetic used to be commonplace when computer hardware was missing built-in support for rational arithmetic. Low end embedded systems still use fixed point for the same reason [3, pp. 211].

Equal Distribution

Compared to integer fractions, fixed point numbers evenly distribute values on the number line. Figure 1.3 illustrates this for a low resolution fixed point type. Compared to integer fractions, fixed point numbers are equally accurate no matter the range. This is great if all numeric variables in a given algorithm are similar in magnitude. But it also means that fixed point numbers are not particularly flexible. For any given operation, all operands need to stay within the same level of magnitude. As an example, multiplying values

$$1.2 \cdot 2^{-10} \cdot 3.4 \cdot 2^{10} \tag{1.27}$$

with big difference in magnitude requires either (1) the use of big N and M (wasting space) or (2) accepting rounding error. Users always need to be aware of these pitfalls and the problems caused by them [9]. The problem with fixed point numbers is that they lack flexibility in respect to magnitude.

1.5 Floating Points: Scaled Fixed Point Numbers

So far we have looked at two was of representing $\mathbb{Q}$, integer fractions and fixed point numbers. But reality is that today's computers do not use either format for rational arithmetic. Instead floating point numbers are the dominant format and have been for a long time [2, pp. 225, 3, pp. 84].

Scientific Notation

The core idea behind floating point is that they represent numbers in a format akin to the familiar scientific notation. Scientific notation represents numbers as a fraction raised

to some scale. For example, the Avogadro Constant

$$N_A = 6.02214076 \cdot 10^{23} \quad [10] \tag{1.28}$$

as used in science would be quite awkward to write without its exponent. Another property of floating points is that some bit patterns are reserved for special values. In particular, floating points reserve bit patterns for (1) not a number (*NaN*) when a given result cannot be computed (e.g. as a result to division by zero) and (2) infinity to indicate that a value is too big ($+\infty$) or too small ($-\infty$) to represent with the limited numbers of bits.

IEEE 754 Standard

Floating point numbers have been around for pretty much as long as there have been digital computers [2, pp. 214]. Today when we talk about floating points (or "floats"), we usually mean IEEE 754 floating points. IEEE 754 [11] standardized the format and was released as early as 1985. A recent update to the standard appeared in 2007 [12].

Defintion 7. *A floating point number x of type $F_{M,E}$ is a bit string*

$$x = s \underbrace{m_{M-1} \cdots m_1 \, m_0}_{\textit{Mantissa } m} \underbrace{e_{E-1} \cdots e_1 \, e_0}_{\textit{Exponent } e} \tag{1.29}$$

of fixed length $1 + M + E$ that represents some value in $\mathbb{R}$ or alternatively any of the three special values NaN, $+\infty$ or $-\infty$. Bit string x consist of sign bit s, unsigned integer exponent e and mantissa m.

Aside from the familiar sign bit s, the value of some floating point x is split into a so called mantissa or fraction m and exponent or scale e. Referring back to the example in Equation 1.28, we can split up constant N_A into fraction and scale, viz.

$$N_A = \underbrace{6.02214076}_{\text{Fraction}} \cdot \underbrace{10^{23}}_{\text{Scale}} \tag{1.30}$$

and that is exactly what floating points are doing. Floating points really are scaled fixed point numbers [3, pp. 81].

Defintion 8. *Given floating point x of type $F_{M,E}$, the value of x is given by*

$$x = (-1)^s \cdot 1.m \cdot 2^{B-e} \tag{1.31}$$

if x does not represent a special value. Constant B is the "bias" fixed for type $F_{M,E}$.

Appendix A illustrates Definition 8 with an example if necessary. The advantage of floating point compared to fixed point numbers is that floats are flexible. Numbers of different magnitudes can happily live together in the same format. Surely this flexibility is one of the reasons why floats are popular with computer architects old and new.

Floating points are the de-facto standard for real arithmetic. They perform okay for most applications, though their jagged distribution and rounding behavior can be difficult to understand. Rounding in particular is ever present in floating point arithmetic. This rounding error can accumulate as each step introduces new inaccuracies. Errors caused by rounding can be hard to track down, requiring complicated numerical analysis.

1.6 We Live with Machines that Lie

We reviewed various number formats. Tedious as it may have been, we made out various classes of errors that haunt computer arithmetic. Beginning with signed integers, we accepted that numbers in computer arithmetic are limited by some fixed number of N bits. This unfortunate reality results in a limited range of expressible values. Next up, defining signed integers proved to be rather tricky, requiring the use of the two's complement. Clever engineering is required to avoid confusing and wasteful special and duplicate values. As a third format, we looked at integer fractions, an intuitive way of expressing values in $\mathbb{Q}$. Yet in practice, the distribution of values on the number line was too jagged to be useful. Fixed point numbers $F_{N,M}$ solve this problem, but are limited in range and magnitude. Users have to do quite a lot of thinking about which parameters N and M to pick. Finally, we looked at floating point numbers. While in active use today, they combine all problems discussed before. They are limited in range by a fixed number of bits. They have many duplicate patterns for the same value (zero, *NaN*, infinity). They are flexible in that they can adapt their magnitude but using them can be confusing or even dangerous as rounding is always present.

While today, floating point arithmetic is quick, its results can be deceiving as rounding error accumulates. The more steps involved in a given computation, the harder it is to reason about the accuracy of the result. While computers are very fast, they also produce lots of errors. Indeed, "[computers] lie all the time, and at incredibly high speeds." [13, p. xiv]. We should try to do better.

2 Unum Arithmetic

Computer arithmetic is riddled with error. We want to do better. This chapter takes a look at ideas presented by John Gustafson on this topic. In particular we will review the ideas behind so-called unum arithmetic. Unum arithmetic aims to provide a replacement for today's floating point arithmetic that is more accurate and honest about rounding. That is, unum arithmetic tries to be explicit about rounding error rather than hide it.

Three Types

By now there are three iterations on unum arithmetic; we will refer to them as Type I, Type II and Type III [14, pp. 21-28]. Each type is quite different in design but all versions share similar goals and features. This chapter reviews each type of unum arithmetic.

Heated Debate

Early versions of unum arithmetic were the target of serious criticism. In particular, William Kahan, known for work on the IEEE 754 floating point standard, has been rather vocal about his dismissal of Type I and Type II arithmetic [15, 16]. We will refer back to these criticisms as we introduce the different formats. Certainly from an outsiders point of view, discourse between Gustafson and Kahan appears quite heated. We opt for a more calm approach.

2.1 Unum Type I: Variable Accuracy Floating Points

In 2015, John Gustafson presented the Type I unum number format in "The End of Error" [13] meant as a replacement for IEEE floating points. Unum numbers are a superset of IEEE floats that track inaccuracies introduced during operation. As a superset to IEEE floats, unums inherit sign, exponent and mantissa fields. The most notable change compared to floats is the introduction of a so-called u-bit. The u-bit (uncertainty bit) indicates whether the represented value is exact or only an approximation. Exact unums (u-bit set to `0`), are floating point values that represent a concrete value x on the number line. Uncertain unums (u-bit set to `1`) represent an open interval (x, y) between neighboring values x and y on the number line. Figure 2.1 illustrates this with an example.

U-Bit

The idea here is that Type I unums allow the computer to be honest about results. Consider the example in Figure 2.1 again. Say we wanted to represent real value $1/2$ in this format. If we were using traditional floating points without u-bit, we would have to round the result to either 1 or 2, neither of which is correct. In contrast, unums do not have to lie. Instead of returning a rounded result, they can be honest and instead return

$$1^u = (1,2) \tag{2.1}$$

which is no doubt a correct statement. The u-bit allows us to be honest about result. When the result is uncertain, unum arithmetic does not pretend that it is not.

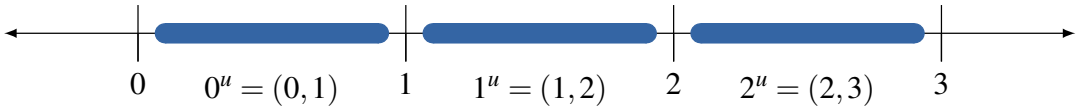


Figure 2.1: Simplified representation of unums on the number line. A given unum can either represent a value exactly (u-bit set to 0, e.g. 1, 2, 3) or it can represent an interval between neighbors on the number line. In that case, the u-bit is set to 1, e.g. for $0^u = (0,1)$ or $1^u = (1,2)$.

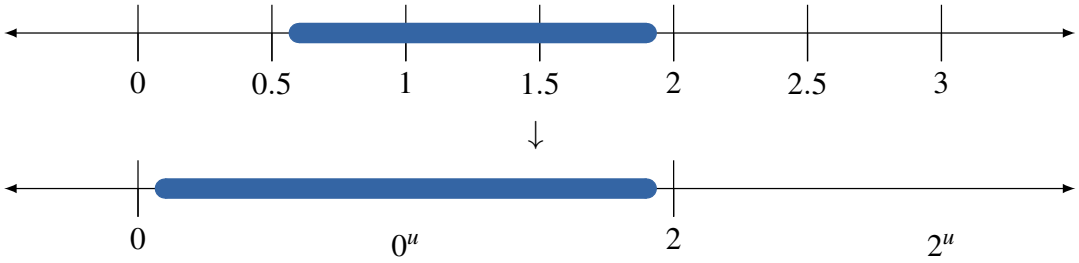


Figure 2.2: Unums adapt their accuracy to fit the result without traditional rounding. In this simplified example, a computation with a high resolution unum type should return a result of $x = (0.5, 2)$. However, as unums can only represent concrete values or intervals of neighboring values, result x is not expressible in this type. To avoid rounding to a concrete value which introduces untracked error, x is converted to a lower resolution environment where x is expressible as $x = 0^u = (0,2)$.

Variable Width

What is peculiar about the design of unums compared to other contemporary number formats is that Type I unums are of variable width. With Type I unums, individual fields such as mantissa and exponent vary in size. This is sometimes advertised as a particularly efficient way of storing numbers as high resolution is only required for some but not all values [13, pp. 40]. More importantly, variable width is necessary to approximate arbitrary intervals (x, y) for any combination of x and y. If the result of some computation does not fit exactly into a concrete value x or an interval of neighbors (x, y), the bit width can be reduced, resulting in a coarser grid on the number line. The grid is adapted until we can find a matching set up neighbors (x, y) to return. Figure 2.2 illustrates this with a simplified example.

Interval Arithmetic

Type I unums can adapt their resolution to approximate arbitrary intervals. Instead of returning rounded values, unums return a bound on the result. This makes Type I arithmetic similar to traditional interval arithmetic. Interval arithmetic has been around for a long time and provides intuitive rules such as

$$[a,b] + [c,d] = [a+c, b+d] \tag{2.2}$$

for addition. This allows us to do arithmetic on intervals and as such keep track of er-

ror [17]. Recent developments include a standardized variant of interval arithmetic based on IEEE floating point [18]. But the big problem with interval arithmetic is that the bounds on results quickly grow to be so large that they are not at all useful [13, pp. 69, 19]. This has hindered adoption of interval arithmetic in the past.

Criticism

The use of a dedicated u-bit to track error is novel, but even so Type I unums have been the subject of heavy criticism. The most common argument being that they are unfit for implementation on silicon. Variable width means that not all values will fit in fixed registers, a cause of much concern [14, p. 22]. William Kahan of IEEE floating point fame provides a more elaborate criticism. He notes that many of the advertised features of Type I arithmetic are no by no means guarantees [15]. It is certainly true that Type I unums can be more space-efficient and accurate than floats. But this is not always the case. And if unums cannot offer any guarantees, users still have to apply numerical analysis as is the case for floats. Difference being that analysis of floating point arithmetic is a well-understood field while numerical analysis for unums is not [15, 16].

In summary, Type I unums are a superset of traditional IEEE floating points. The introduction of a u-bit removes error from computation in that the result is not a rounded value. Rather it is a best-effort bound on the result. The big downside is that Type I arithmetic uses variable bit widths. While this can result in quite efficient storage, it means that Type I unums are unpractical as they are difficult to implement in real hardware. And without efficient hardware implementations, unums will never be a replacement for floating points.

2.2 Unum Type II: Projective Reals and Sets of Real Numbers

Projected Reals

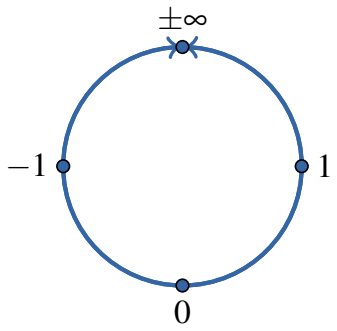


Figure 2.3: The projected reals wrap the number line on a circle.

Type II arithmetic aims to provide similar features as Type I unums without variable width storage [20]. Inspiration for Type II arithmetic comes from the "projectively extended real numbers" or "projected reals" [21]. The projected reals wrap around the real number line on a circle. Here, negative and positive infinity are not unreachably far away; rather they occupy a concrete point $\pm\infty$. The result is that we move from the well-known number line to a number circle (Figure 2.3).

Type II Unums

Based on the projective reals, we now have to differentiate between two components that make up Type II arithmetic. On one hand, we have the Type II unums themselves. A Type II unum is a point on the number circle that represents either a concrete value x or an interval (x,y) of neighboring values x and y. Implemented with a u-bit, Type II unums have a lot in common with their Type I predecessor. What is unique about Type II unums is that the steps between grid points on the number circle are arbitrary. They are

not determined by the format. Rather users of a Type II environment decide on what the individual steps should be. The points chosen between 1 and $\pm\infty$ are called the "u-lattice" and this u-lattice determines the values on the other three quadrants. It is up to developers to decide on a scaling that makes sense for the given application domain.

SORNs

The second component that makes up Type II arithmetic are the so called "SORNs" which is short for "sets of real numbers." A SORN is a set of arbitrary Type II unums. That is, if x, y and z are Type II unums, then

$$\{\},\{x\},\{y\},\{z\},\{x,y\},\{x,z\},\{y,z\},\{x,y,z\} \tag{2.3}$$

are all examples of possible SORNs. Because SORNs include the empty set, there is no need for a special "not a number" value. Rather the result of illegal operations such as division by zero is the empty set. SORNs are a genuine improvement on Type I unums. While Type I unums can only represent intervals, SORNS represent arbitrary sets.

Unums Are Pointers

Type I unums are a superset of floating points. To calculate the value of a given Type I unum, we compute the product of fraction and scale. Type II unums are different in that their bit strings do not tell us much without knowing the exact values on the u-lattice. Indeed Type II unums are more like pointers to the u-lattice. Similar, to implement SORNs that capture n-bit Type II unums, the suggested way is to employ bit strings 2^n bits in length. Each bit represents one concrete Type II unum. As the size of SORNs is exponential in n, it goes without saying that SORNs get quite large even for small n.

Implementation Is Difficult

Type II unums can be a nice way of representing numbers in a given domain [22]. The big problem is that doing actual down to earth arithmetic is hard. Given two Type II unums x and y, to compute $x+y$, both values are first converted to SORNs $\{x\}$ and $\{y\}$. Arithmetic is then done in terms of SORNs, which requires table lookups. These tables get huge even for small values of n. The result of lookup $\{x\}+\{y\}$ is either (1) a concrete value convertible back to a Type II unum or (2) some value only expressible with a SORN. Which means that Type II arithmetic is not closed. The result of arithmetic operations on Type II unums is not always just another Type II unum, but rather it could be a SORN as well. It is left up to users to deal with this ambiguity.

While Type II unums offer interesting properties, implementation is difficult. SORNs and associated lookup tables exponential in size are problems impossible to ignore. Be that as it may, ordering numbers on the projected reals is an interesting idea. SORNs can represent arbitrary sets for a given type and Type II unums maintain the u-bit. That is, Type II unums can track whether results are correct or merely guesses. Type III arithmetic, which we will look at next, inherits much from Type II unums.

2.3 Unum Type III: Posit Arithmetic

With Type I and Type II arithmetic out of the way, it is finally time to look at the most recent development, Type III arithmetic. It was first introduced by John Gustafson in 2017 [23, 24]. Just as Type II arithmetic is split into plain unums and SORNs, Type III describes an interplay of so called "posits", "quires" and "valids". As we will see, posits

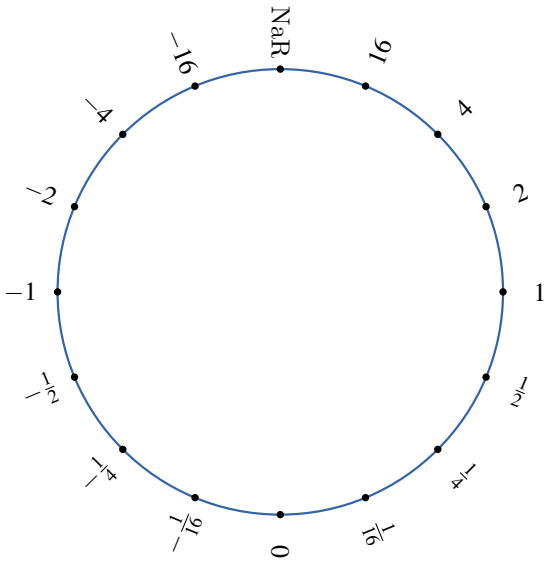


Figure 2.4: The $P_{4,1}$ type. A posit with width $N = 4$ and exponent size $ES = 1$. Values on the right side of the circle are positive while values on the left side have negative sign.

are a drop-in replacement for the familiar floating points. Quires are an aid for computing more accurate results with posit arithmetic. Finally, valids are the closest to Type I and Type II unums. Valids include the *u*-bit and can keep track of arbitrary error.

Posits Are Like Floats

We begin with an introduction of posits. Posits are like floating points in that a given posit encodes a product of scale (the exponent) and fraction (the mantissa). Posits represent concrete values exactly; any given posit type samples the number line just as floats do. There is no *u*-bit in a posit, instead posits round to nearest after each arithmetic operation. Surely most readers will find this to be quite odd. Unums promised to be the end of error, yet with posits we are back to rounding and as such error.

Drop-In Replacement

Indeed posits have more in common with floats than they do with previous iterations of unums. This does have advantages. While the encoding of posits is totally different from that of floating points, posits are meant to be a drop-in replacement for floats. It should be easy to replace floating point arithmetic with the posit alternative. The supposed advantage being that posits can be more accurate than floats in that for example a 32 bit posit can beat a 64 bit float. Posits wants to "[beat] floating point at its own game" [24]. If we can replace 64 bit floating point computations with 32 bit posit arithmetic, that means less required storage, less load on memory and caches; ultimately it could mean faster and more efficient arithmetic.

Projected Reals And Posits

What posits do have in common with Type II unums in particular is that we think of posits as values on the projected reals. All concrete values of a given posit type can be sorted on a circle where zero is at the bottom and $\pm\infty$ is at the top. Figure 2.4 illustrates this with an example. Early literature on posit arithmetic refereed to the point on top of the circle as "complex infinity". More recent literature refers to it as "*NaR*", as in "not a real", similar to *NaN* familiar from IEEE floating points [23, 25]. *NaR* is the only special

value of a given posit type and only one unique bit pattern is reserved for it. In particular, posits have no way of representing $+\infty$ or $-\infty$, something floating point can do.

2.3.1 Binary Format

Posits are fairly well defined [23, 24, 25] and various implementations of the format exist [26]. Regardless, it is useful to first review the basic format and semantics of the posit format as posits are a major part of this thesis.

Defintion 9. *A posit p of type $P_{N,ES}$ is a bit string*

$$p = p_{N-1} \cdots p_1 \; p_0 \tag{2.4}$$

of fixed length N that represents some value in $\mathbb{R}$ or special value NaR. A given posit p is a two's-complement number and can be split into up to four fields: Sign bit S, regime R, exponent E and fraction F.

Sign, Regime, Exponent, Fraction

While floating point types are parameterized by the length of mantissa and exponent, posit types $P_{N,ES}$ are parameterized by total length N and a so called exponent size ES. But encoding the value of a given posit bit string p is more involved than with floats. While posits are a fixed size format in that the width of a given posit is always N bits, individual fields within those N bits can differ in length. To decode each field, we have to first pay attention to the most significant bit.

- Most significant bit p_{N-1} is the sign bit S. Sign bit S set to `1` indicates that the value of p is negative. Otherwise the value is positive.

If sign bit S is set to `1`, we have to apply the two's-complement before decoding any of the remaining three fields. Parsing the remaining $N-1$ bits from left (most significant) to right (least significant bit), we can make out a maximum of three more fields that might be contained within a given posit p.

- Followed by the sign bit comes regime R. All regime bits except the last are identical to sign bit S. The last regime bit differs in that it is exactly $\neg S$.

  The regime is not interpreted as an integer; rather the length of the bit string determines the value of the regime. Say if $R =$ `0001`, then R consists of four bits and as such we arrive at a regime $R = 4$. Regime R can be both positive or negative, depending on whether the first bits in the regime are `0` or `1`, respectively.

  Regime R plays an important role in determining the scale of p, acting as a kind of "super-exponent" raised to a big power of two.

- If the regime did not consume all bits of p, then up to ES-many bits make up the exponent E. Similar to floating points, E is interpreted as an unsigned integer that represents an exponent value. Unlike floating point, posits have no bias that is subtracted from integer E.

- All remaining bits are part of fraction F. Fraction F works like the mantissa in floating points. It represents some binary fraction with an implicit leading one.

  For example, if $F = \texttt{1010}$, then the value of F is

  $$1.1010_2 = 1 + \frac{1}{2} + \frac{0}{4} + \frac{1}{8} + \frac{0}{16}. \tag{2.5}$$

  If fraction F is zero or not present (pushed away by regime or exponent), the value of fraction F is $F = 1$.

The binary format of posits is a novel design. While the total width N of a given posit is fixed, fields inside the posit differ in length.

2.3.2 Value of Posits

Useed

Before we can compute the value of a given posit p, we have to make one more definition.

Defintion 10. *Given a posit p of type $P_{N,ES}$, then constant*

$$\mathcal{U} = 2^{2^{ES}} \tag{2.6}$$

is the useed of p [23, p. 9].

Useed $\mathcal{U}$ plays an important role in determining the dynamic range of a given posit type. For the most commonly used exponent size $ES = 2$ we get

$$\mathcal{U} = 2^{2^2} = 16. \tag{2.7}$$

Equation 2.6 looks harmless, but $\mathcal{U}$ quickly grows to be quite huge, even for small choices of ES. For example, for $ES = 3$ we get an useed of $\mathcal{U} = 256$ and for $ES = 4$ we already get $\mathcal{U} = 65536$. Notably the current draft of the posit standard fixes $ES = 2$ for all posit types [25, p. 6], resulting in a standard useed $\mathcal{U} = 16$.

Defintion 11. *The value of a given non-NaR posit p can be computed in terms of*

$$p = (-1)^S \cdot \underbrace{\mathcal{U}^R \cdot 2^E}_{Scale} \cdot \underbrace{F}_{Fraction} \tag{2.8}$$

where R is the regime, E the exponent and F the fraction of p [23, p. 13].

Scale and Fraction

Just like the value of floating points comes down to a product of exponent and mantissa, the value of a given posit is a product of scale and fraction. The scale of a given posit consist of both useed $\mathcal{U}$ raised to regime R and exponent E raised to the power of 2. While decoding the individual fields of a given posit can quite involved, computing the concrete value from these fields is straight-forward.

Rounding Rules

Posits only have one special value, *NaR*, which occupies one unique bit pattern. *NaR* is used as a return for illegal operation such as division by zero, but never for any legal operation. In particular, posit values never overflow to infinity. If the result of a computation is too big to represent in the current posit environment, the biggest representable posit (called *maxpos*) is returned instead. For example, in the $P_{4,1}$ environment (Figure 2.4), computing $16+4$ returns $16+4=16$ because $maxpos=16$. A comparable floating point environment would return $+\infty$ as a result. The argument for this decision is that rounding to infinity results in infinite error rather than some finite error. As such rounding to *maxpos* is more correct that rounding to $+\infty$ [23, p. 7]. Posits never round to zero either. In cases where rounding to zero might occur with floats, posits round to *minpos* instead. *minpos* is the smallest non-zero value contained in the given posit type. The argument here is similar; it is supposed to be better to round to *something* rather than round to *nothing* [23, p. 7].

2.3.3 Posits in a Nutshell

In summary, posits are a reinvention of traditional floating points. Posits represent concrete values on the number circle, not intervals as the u-bit is omitted. Any given posit has a fixed length N and can be split into up to four different fields. Only one unique bit pattern is reserved for special value *NaR*. Unlike floats, rounding favors limits *minpos* and *maxpos* rather than zero or infinity. Posits are meant to be a drop-in replacement for floats that can solve the same problems as floats but with less bits, resulting in more efficient computer arithmetic.

Unsurprisingly, posits pop up again and again throughout this thesis. Typically we will use low resolution types as the limited number of concrete values makes for easier to understand examples. Appendix B plots some of these low resolution types handy for review.

2.4 Unum Type III: Quires

Posits round very much like floating point numbers do, a true *return of error* as far unum arithmetic is concerned. But Type III arithmetic has two tricks up its sleeve to ensure that results are reasonably accurate, quires and valids. We will first look at quires, which are well-defined, easier to understand and smaller in scope.

Defintion 12. *Given a posit type $P_{N,ES}$, the associated quire is a $16N$-bit wide fixed-point number of type $Q_{M,K}$ where $K=8N-16$ and $M=8N-16-1$ [25, p. 7].*

Accurate Intermediate Format

The quire is a high precision accumulator for intermediate operations. Parameters M and K are mostly arbitrary, chosen such that for the majority of computations, the given quire is accurate enough to circumvent intermediate rounding error. That is, the quire is meant to circumvent excessive rounding introduced by multi-step computations [23, pp. 81]. Instead of rounding posits after each step, as many operations as possible should be moved to the more accurate quire. But the quire is only an intermediate format. In

computations, posit arguments are first converted to a quire representation. Arithmetic is done in the quire and then finally the result is converted back to a posit. Because the quire has a much higher resolution than its associated posit type, intermediate results are much more accurate.

Accumulator

Calling the quire an accumulator is no accident. For each supported posit type, only one associated quire is required on the arithmetic unit. As such, mathematical expressions need to be rewritten to use an accumulator-like interface. This does mean extra work for programmers switching from floating point to posit arithmetic. But Gustafson argues in favor of this explicit conversion to and from quire as hidden conversions can be confusing [23, pp. 81]. The idea behind the quire is not new. Similar ideas which inspired Gustafson have previously been discussed as a "fused dot product" [23, pp. 80-81].

Fused Dot Product

In summary, quires are an intermediate format for posit arithmetic. They can improve accuracy as they cut down on accumulated rounding error. Admittedly, quires are no perfect solution as even the quire is finite in size and operations cannot always be rewritten to use a single accumulator. Even so, the quire is a fundamental part of Type III arithmetic and must not be ignored.

2.5 Unum Type III: Valid Arithmetic

What we omitted so far are valids. Valids are the final building block of Type III arithmetic and were first introduced together with posits in 2017 [23]. Indeed, a short summary of valids is found on the first page of aforementioned introduction.

> *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 [23, p. 1].*

To the best of our knowledge, this "separate document" does not exist at this point. What we do have is a rough sketch of the binary format [23, pp. 23]. There also exists a prototype `SigmoidNumbers` [27] implementation worked on in 2016. `SigmoidNumbers` implements an interval arithmetic that differentiates between open and closed endpoints. But as `SigmoidNumbers` predates posit arithmetic, it is better understood as an experimental precursor to Type III arithmetic rather than a concrete implementation of valids.

Binary Format

For starters, a valid of type $V_{N,ES}$ consists of two $P_{N,ES}$ posit endpoints, both with an additional u-bit. The union of posit and u-bit is called a "tile", take two and you have a valid (Figure 2.5). Compared to plain posits, valids already feel more like original unums as they make use of the u-bit.

Do note that the original literature actually defines $V_{N,ES}$ valids to consist of $P_{N-1,ES}$ tiles. We find this to be quite confusing which is why we take the liberty to slightly change the definition in our thesis. Rest assured that this change is only a formality. All of our findings apply just as well to the original definition.

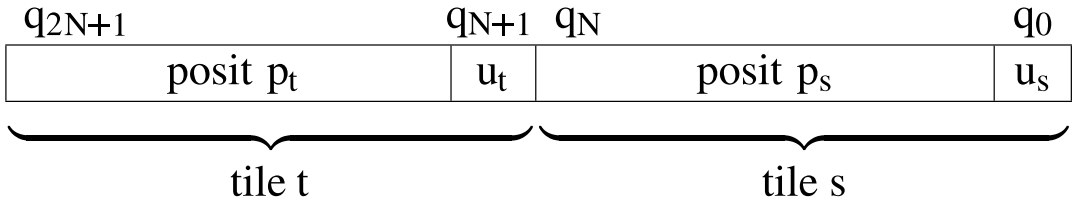


Figure 2.5: Valid $V_{N,E}$ binary format. Any such valid consists of $2N+2$ bits q_{2N+1} down to q_0. Each of the two tiles t and s consists of a $P_{N,E}$ posit p and a dedicated uncertainty bit u.

Valids Are Intervals

Although details on the format are left open at this point, it is clear that valids represent open or closed intervals. With valids, we should be able to regain the ability to represent both concrete posits p as well as intervals (p,q) of arbitrary posits p and q. A feature we lost with posits, coming from Type I and Type II arithmetic. That said, we can already tell that valids are a regression compared to SORNs. As valids have only two endpoints, they will never be able to represent arbitrary sets of posit values.

2.6 Summary

By now we should have developed a feeling for the different types of unum arithmetic. Type I unums are numbers of variable size that adapt themselves to accommodate rounding error. Type II unums and the associated SORNs are based on the projective reals in that they represent values or sets on a number circle. While Type I and Type II arithmetic offer interesting properties, implementation is difficult.

Type III Arithmetic

Finally, Type III arithmetic is divided into posit, quire and valid arithmetic. Posits want to be a drop-in replacement for floats. Algorithms written for floating point arithmetic should be trivial to port to use posits instead. Quires on the other hand are an aid for accurate posit arithmetic. But the quire is only a temporary register, for storage the posit format is preferred. Finally, valids appear to be intervals of two posit endpoints. Unlike posits and quires, valids contain the u-bit so familiar from Type I and Type II arithmetic.

2.6.1 Problem Statement

Float Replacement

Wee see that posits could provide an alternative to traditional floats. But posits are also the odd one out in the unum family. They suffer from rounding error and do not contain the u-bit. Indeed many problems identified in Chapter 1 also apply to posits. Posits are limited in bit width, so only a fixed finite number of values can be represented. While posits do not have duplicate patterns, their complex encoding can make them difficult to understand. Perhaps worst of all, rounding is ever present with posit arithmetic. The first open question we have to ask is whether posits can be a genuine replacement for floating points that is actually better.

Valids Are Left Vague

The second question is related to valids. Because surely valids are the true successor to Type I and Type II unums. Yet very little is available on them. As such it is an open

question how to design and take advantage of the format. In the following section, we will investigate the properties of a canonical valid arithmetic. We will see that valids have potential to be useful, but understanding them in detail is not the easiest task.

3 A Definition of Valids

Contributions

This chapter focuses on the valid format. We contribute with a detailed discussion of valid tiles and the valid format itself. We identify special values that arise from the current design. Finally, we introduce algorithms for valid arithmetic not previously found in literature. What we arrive at its a working definition of valids that can be used for experiments. A building block for future developments.

Valids As Debugging Environment

The big challenge is that valids are not exactly well defined. What we can infer is that valids represent some kind of interval arithmetic with posit endpoints. The domain of valids is more one of exploration and development. Gustafson himself has made the following comment aimed in this direction in an online discussion.

> *I haven't written them up yet, but _valids_ are what you want if you [...] want software that can gracefully and mathematically handle the results that make floats generate a NaN. Think of the valid computing environment as the numerical debugging environment for posits. It's slower and ultra-careful and rigorous, but once you get your algorithm to the point where it never tries to color outside the lines, then switch to posits and go FAST [28].*

Valids are not meant to be the go-to format for arithmetic, rather they are a programming or debugging tool.

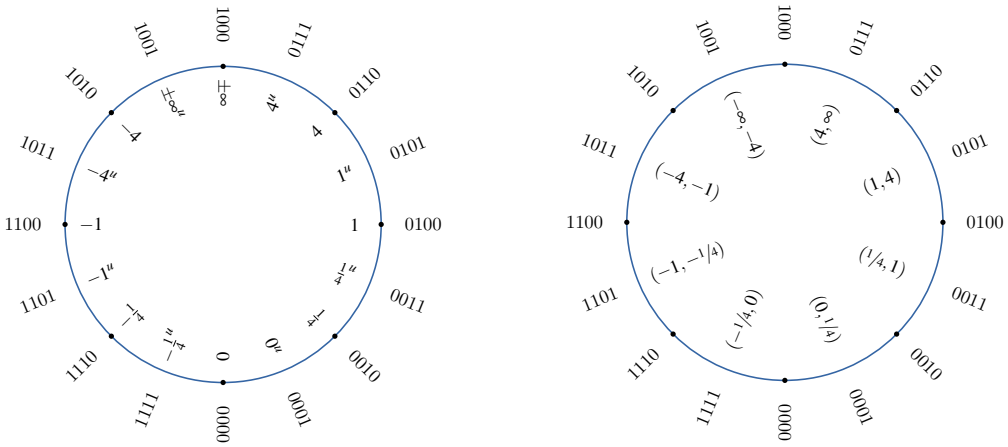
3.1 Attempting Cell Arithmetic

From literature we do know that valids are made from two tiles where any given tile consists of a posit with an additional u-bit [23, p. 23]. A detailed discussion of valid tiles is missing however. As a starting question, we wondered whether it might be possible to design useful arithmetic around just these plain tiles. To make the distinction between (1) tiles as building blocks of valids and (2) tiles as objects on their own explicit, we differentiate between (1) tiles and (2) "cells". While tiles are building blocks of valids, cells are stand-alone tiles.

Defintion 13. *A* cell *c of cell type $C_{N,ES}$ is a bit string of length $N+1$ consisting of (1) a posit p of length N with exponent size ES and (2) an additional u-bit. Cells c with u-bit set to `0` represent the value of p exactly. Cells with u-bit set to `1` represent the interval (p,q) where q is the successor of p.*

Cells Are Like Type I Unums

As an example, Figure 3.1 shows all possible cells that use $P_{3,1}$ posits as an underlying posit type. We see that cells are very reminiscent of Type I unums. Any given cell can



(a) Cells represented either with their concrete value p or their interval neighbor p^u. Note that p and p^u have matching bit patterns except for the final u-bit.

(b) Cells in interval notation. Of particular interest should be the intervals next to $\pm\infty$. If we approach from the positive side, we arrive at $+\infty$, if we approach from the negative side we arrive at $-\infty$.

Figure 3.1: $C_{3,1}$ tiles made from $P_{3,1}$ posits. Adding an additional u-bit means that we can represent intervals $p^u = (p,q)$ between neighboring posits p and q.

represent either value p exactly or alternatively an interval (p,q) of neighboring values p and q. And just like Type II unums, cells are laid on a circle inspired by the projected reals. On the downside, unlike Type I unums, cells do not have any way to adapt their accuracy. As such, cells cannot approximate arbitrary intervals. Looking at Figure 3.1 again, the given cell type can represent the interval $(1,4) = 1^u$, but it cannot approximate the interval $(0,4)$ because 0 and 4 are not direct neighbors. Cells are like Type I unums, but more limited.

3.1.1 Cell Comparisons

Cells discretize the number circle with points p and intervals (p,q) in between. The first operation we will define on cells are the comparison operations. These will also prove useful later when we define comparisons on valids.

Defintion 14. *Two cells c and d of cell type $C_{N,ES}$ are equal (i.e. $c = d$) if and only if the bit patterns of c and d are identical. In particular, special value $\pm\infty$ is equal to itself.*

Comparing cells for equality is identical to comparing posits or plain integers for equality as there are no redundant bit strings that represent the same value or interval. Defining the less-than operation is a bit more involved, but we find that it is also quite intuitive.

Defintion 15. *Given two cells c and d of cell type $C_{N,ES}$, we say that c is less than d (i.e. $c < d$) if and only if the bit string of c is less than the bit string of d when compared like signed integers. The only exception is special value $\pm\infty$, which is less than no value.*

Above definition may feel a bit technical. An easy way of understanding Definition 15 is that cell c is less than cell d if c is further on the left side of the circle than d. What remains is the special value $\pm\infty$ which is both positive and negative. In consequence, we cannot reasonably say that it is greater or less than any other value.

3.1.2 Cell Arithmetic is not Closed

With their familiar Type I unum feel, one might be tempted to define cell arithmetic as an arithmetic that can stand on its own. But when attempting such a thing, one very quickly runs into a problem. The result of cell operations often does not fit inside another cell.

Consider the following example of a canonical cell arithmetic based on $P_{3,1}$ posits (Appendix B). It seems obvious that, say,

$$0+\frac{1}{4}=\frac{1}{4} \tag{3.1}$$

holds in this environment. In this case, cell arithmetic is identical to posit arithmetic as the result is already perfectly representable in just plain posits. But where in plain posit arithmetic we have to be satisfied with

$$1+1=1 \tag{3.2}$$

as results are rounded to the nearest representable posit, we could instead return

$$1+1=(1,4) \tag{3.3}$$

in cell arithmetic. We cannot return the correct result of 2, but rather than hiding rounding error, cell arithmetic can at least be honest. The honest answer in this case is that the result is somewhere between 1 and 4. Cell arithmetic allows us to correctly capture the result. But unfortunately things do not always work out in our favor. Consider the case of

$$(-4,-1)+(1,4)=(-4+1,-1+4)\stackrel{?}{=}(-3,3) \tag{3.4}$$

where we are adding intervals and as such are forced to apply the rule

$$(a,b)+(c,d)=(a+c,b+d) \tag{3.5}$$

adapted from Type I unums and traditional interval arithmetic [13, p. 113]. The problem here is that the resulting interval of $(-3,3)$ fits in none of the available cells. Cells also lack the ability to dynamically scale their resolution to accommodate such results. No matter at which one of the 2^4 cells from Figure 3.1 we look at, none of them strikes us as an acceptable result for above computation.

This example shows that cell arithmetic cannot be closed. When adding two cells we may not have any way of representing the result without being completely wrong. One can find many such examples for other arithmetic operations as well. As such we have to accept that cells on their own are probably not particularly useful.

3.2 Valids Defined

Defintion 16. *A valid v of valid type $V_{N,ES}$ is a bit string of length $2N+2$ consisting of two tiles $t = (p_t, u_t)$ and $s = (p_s, u_s)$ where each individual tile consists of a $P_{N,ES}$ posit endpoint p and a u-bit. Valid v represents an interval on the tiled number circle. That is, v is a set that contains all cells between start p_t and end p_s when traveling counter-clockwise on the number circle. The individual interval endpoints are open if the respective u-bit is set and closed if the respective u-bit is not set.*

Examples

The easiest way to understand above definition is to think of some examples, some of them illustrated in Figure 3.2. Pick any two endpoints on the number circle and you can make a valid that includes those endpoints and everything between. Because endpoints are so important, we use the notation

$$v = \{t; s\} \tag{3.6}$$

for valid v composed of start tile t and end tile s. We use curly braces to indicate that the bounds can be either open or closed, depending on the values of t and s. Also do note the use of a semicolon which differentiates above notation from traditional interval notation (x, y). For example, we get

$$\{0;4\} = [0,4] \quad \{1;4^u\} = [1,4) \quad \{1^u;4\} = (0,4] \quad \{1^u;4^u\} = (1,4) \tag{3.7}$$

where we use a raised u to state that the u-bit is set for the given endpoint.

3.2.1 Regular and Irregular

Note that so far, starting point t has always been smaller than endpoint s. We call these valids regular. What is maybe a bit unexpected are valids that are not regular, that is valids where start t is bigger than end s. We call these valids irregular.

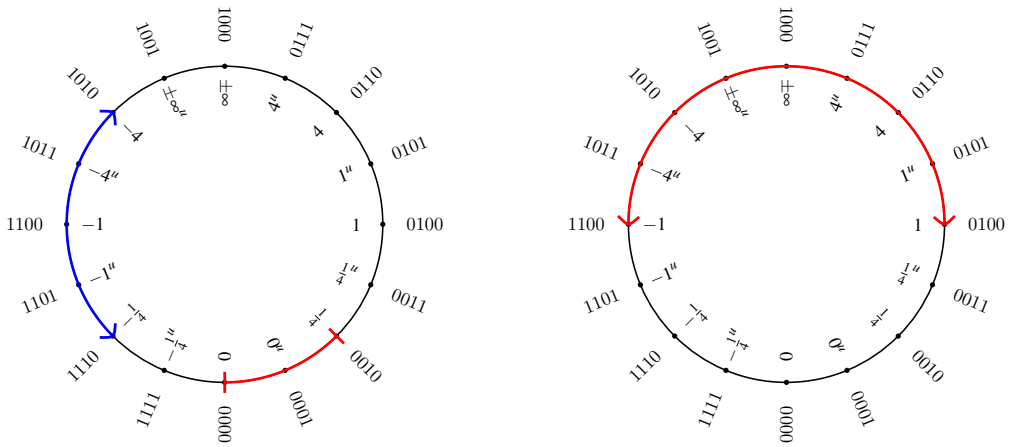
Defintion 17. *Valid $v = \{t; s\}$ with start tile t and end tile s is called regular if $t \leq s$. If on the other hand $t > s$ holds true, v is called irregular.*

Both regular and irregular valids are allowed, though we find that regular valids are much easier to work with. It should also be apparent that any irregular valid has to contain special posit value *NaR*.

3.2.2 Valid Set Operations

A slightly different way of looking at valids is that a given valid is a set of cells, delimited by tile endpoints. While it is convenient to write valids as intervals, we can also list the elements (the cells) in the valid. For example, for $v \in V_{3,1}$ (Figure 3.2) we get

$$v = [-1,4) = \{-1, (-1,-1/4), -1/4, (-1/4,0), 0, (0,1/4), 1/4, (1/4,1), 1, (1,4)\}. \tag{3.8}$$



(a) Regular intervals. The blue one on the left represents open interval $(-4,-1/4)$ while the red valid on the bottom right represents closed interval $[0,1/4]$.

(b) Irregular interval $(1,-1)$. We begin at startpoint 1 and travel counter-clockwise until we reach endpoint -1. In particular, this valid contains special cell $\pm\infty$.

Figure 3.2: Some $V_{3,1}$ valids on the number circle. A closed endpoint is indicated by a straight line while an open endpoint is represented by parentheses at the end.

Valids are really just sets and as such it comes natural to use standard set operations on them. For example, we can write

$$1 \in v \tag{3.9}$$

to indicate that cell 1 is part of valid v. Operator $\in$ is not the only set operation we can use on valids. We can also take the inverse v^{-1} of a valid v.

Defintion 18. *Given a valid $v \in V_{N,E}$, the inverse v^{-1} is the valid which contains all cells $c \in C_{N,E}$ which v does not contain.*

To illustrate this with a short example, if we take a look at above $v = [-1,4)$ (Equation 3.8) again, we can see that its inverse is

$$v^{-1} = [4,-1) = \{4,(4,\infty),\pm\infty,(-\infty,-4),-4,(-4,-1)\}. \tag{3.10}$$

We see that valids can be understood as sets of cells and as such it comes natural to use typical set operations in conjunction with valids.

3.2.3 Special Valids

While many valids $v = \{t;s\}$ define intervals on $\mathbb{R}$, certain special valids are of particular interest. In total, we found four classes of valids worth extra discussion. Note that for the most part we did not exactly *invent* these special valids. Rather they were *discovered* in that they naturally arise out of the limited information we have on the valid format.

Real Set $(-\infty, \infty)$

If we want to say that v represents some finite real value, but we are not sure at all about which one exactly, we use the real set represented by valid

$$v = (-\infty, \infty) = \{NaR^u; NaR^u\}.$$

Not A Real $\pm\infty$

Posits have one special value: *NaR*. While valids are supposed to be useful without an exceptional value like *NaR*, valid

$$v = \{NaR; NaR\}$$

naturally arises as a representation of *NaR* or perhaps $\pm\infty$. We could declare this particular valid as illegal. But as that would waste a precious bit pattern and introduce a special case to all operations, we prefer not to.

Empty Set $\varnothing$

Empty set $\varnothing$ is the valid that contains no cells. Given any non-*NaR* posit p, we define

$$v = (p, p) = \{p^u; p^u\}$$

to represent $\varnothing$. All representations of the empty set are equal to each other, regardless of p. Posit p needs to be non-*NaR* because $\{NaR^u; NaR^u\}$ represents the real set $(-\infty, \infty)$.

Full Set $\circ$

Full set $\circ$ is the inverse of the empty set, that is valid $\circ$ contains all cells on the number circle, including $\pm\infty$. For any non-*NaR* posit p, we get

$$v = [p, p) = \{p; p^u\}$$

to represent the full set $\circ$. This works out because we start at concrete value p and then travel the number circle until we reach the cell just before p. That is one round trip on the number circle which includes every possible cell. We can also represent $\circ$ as

$$w = (q, q] = \{q^u; q\}$$

where we start ever so slightly off from q and travel the number circle all the way until we reach point q. Full set $\circ$ is different from the real set $(-\infty, \infty)$ in that $\circ$ contains $\pm\infty$ while the real set does not.

3.2.4 Binary Format Reviewed

Wasted Patterns

This definition captures the basic encoding of valids as it arises from the binary format. It is by no means perfect. For starters, we waste a lot of bits on redundant representations for both the empty set $\varnothing$ and on the full set $\circ$. It is tempting to use duplicate bit patterns to represent additional real interval values. But in favor of a definition that is easy to understand, we prefer not to.

What To Do With *NaR*?

Another inconvenience is special value $\pm\infty$. Valid $\pm\infty$, that is $[\pm\infty, \pm\infty]$, is a tile we have to deal with as it arises out of the binary format. As we do not want to declare $\pm\infty$ an illegal state, we keep it as part of our definition.

3.3 Valid Comparisons

Before we start with the scary part, arithmetic, we first define the comparison operations. In particular, we will now look at the check for equality and the less-than operator on valids.

Defintion 19. *Two valids v and w of valid type $V_{N,ES}$ are equal (i.e. $v = w$) if and only if v and w contain the same cells.*

In practice, this means that two valids $v, w \in V_{N,ES}$ are equal if any of the following conditions is met.

- Both v and w are bitwise identical, that is they represent the same interval down to the endpoints.
- Both v and w represent the empty set, that is $v = w = \varnothing$. This special case is necessary because there are multiple ways of expressing the empty set.
- Both v and w represent the full set, that is $v = w = \circ$. Again, as there are multiple ways of expressing the full set, a special case is necessary.

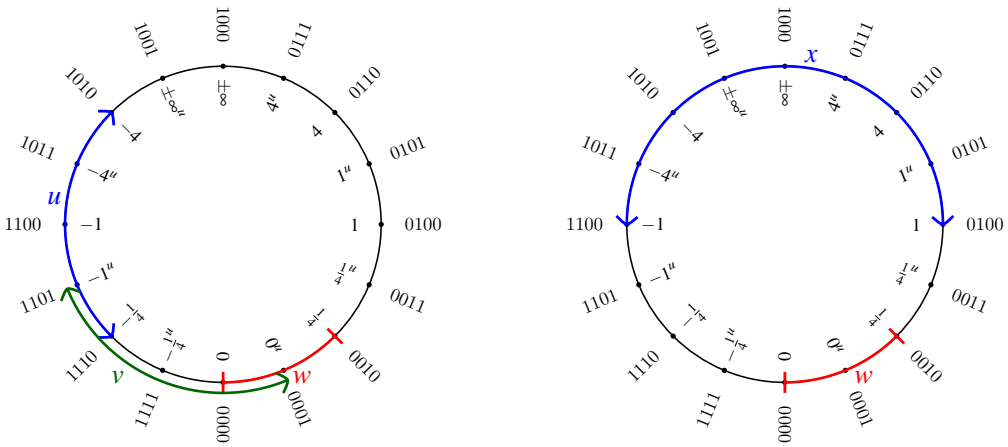
NaR And Equality

We do not define any special cases related to $\pm\infty$. Complex infinity is only equal to itself and nothing else. This is identical to how equality is treated for *NaR* when talking about posits. With Definition 19 we get an easy to understand definition for valid equality. The less-than operation on the other hand involves more creativity. In particular, comparing with valids that contain $\pm\infty$ is something to keep in mind.

Defintion 20. *Given two valids v and w of valid type $V_{N,ES}$, we say that v is less than w (i.e. $v < w$) if and only if any cell $c \in v$ is less than any cell $d \in w$, that is*

$$v < w \quad \Leftrightarrow \quad \forall\, c \in v\,.\,\forall\, d \in w\,.\, c < d \tag{3.11}$$

Both regular and irregular cases are illustrated in Figure 3.3. For most cases above definition is pretty straight forward. We run into an unfortunate situation if we compare with a tile that contains $\pm\infty$. Complex infinity has both positive and negative sign, so



(a) Valid u in blue on the left, valid v in green on the bottom left and valid w on the bottom right. Only $u < w$ holds true as all cells in u are smaller than all cells in w. Overlapping valids such as u and v are never less than each other.

(b) Valid w in red on the bottom right and irregular valid x in blue on the top. While x contains elements smaller than w, it also contains elements greater than w. In particular, x contains special value $\pm\infty$ and as such any less-than comparison with x has to return false.

Figure 3.3: Visualizing the comparison operators on $V_{4,1}$ valids. Any less-than comparison can only be true if both intervals are regular.

clearly it can never be less than anything, nor can it be greater than anything. We have to accept that $v < w$ is never true if any of v or w contain the special value $\pm\infty$. A corollary is that less-than comparisons with irregular valids are always false.

3.4 Attempting Addition Based on Type I Rules

Our first shot at defining addition on valids did not succeed. But realizing why exactly it failed is well worth the effort. This first attempt is based around the definition of addition for Type I arithmetic [13, p. 113].

3.4.1 Porting Type I Addition

Type I addition is defined in terms of a table that covers all possible cases [13, p. 113]. We adopted these rules for use with valid arithmetic. Table 3.4 lists the individual rules as we ported them to valids. We can make some observations here.

- In most cases, we will be looking at intervals (a,b) with $a,b \in \mathbb{R}$. Here we apply

$$[a,b] + [c,d] = [a+c, b+d] \tag{3.12}$$

  familiar from Type I unums and traditional interval arithmetic [29, p. 1048]. Type I

x	$[\pm\infty$	$(-\infty$	$(\infty$	$(a$	$[a$
$[\pm\infty$	$[\pm\infty$	$[\pm\infty$	$[\pm\infty$	$[\pm\infty$	$[\pm\infty$
$(-\infty$	$[\pm\infty$	$(-\infty$	$[\pm\infty$	$(-\infty$	$(-\infty$
$(\infty$	$[\pm\infty$	$[\pm\infty$	$(\infty$	$(\infty$	$(\infty$
$(c$	$[\pm\infty$	$(-\infty$	$(\infty$	$(a+c$	$(a+c$
$[c$	$[\pm\infty$	$(-\infty$	$(\infty$	$(a+c$	$[a+c$

(a) Table for evaluating left tile x.

y	$\pm\infty]$	$-\infty)$	$\infty)$	$b)$	$b]$
$\pm\infty]$	$\pm\infty]$	$\pm\infty]$	$\pm\infty]$	$\pm\infty]$	$\pm\infty]$
$-\infty)$	$\pm\infty]$	$-\infty)$	$\pm\infty]$	$-\infty)$	$-\infty)$
$\infty)$	$\pm\infty]$	$\pm\infty]$	$\infty)$	$\infty)$	$\infty)$
$d)$	$\pm\infty]$	$-\infty)$	$\infty)$	$b+d)$	$b+d)$
$d]$	$\pm\infty]$	$-\infty)$	$\infty)$	$b+d)$	$b+d]$

(b) Table for evaluating right tile y.

Figure 3.4: Prototypical addition table for valid addition $v+w = \{a;b\}+\{c;d\} = \{x;y\}$ based on previous work in The End Of Error [13, p. 113]. As valids are built from posit endpoints and posit arithmetic introduces rounding error, a naive implementation of above tables results in incorrect results.

arithmetic prides itself in being a superior version of traditional interval arithmetic. But as a matter of fact it is made from very similar rules.

- Special value $\pm\infty$ has priority over everything else. If it shows up when computing a bound, that bound will then also be $\pm\infty$.
- Adding endpoints $\infty+\infty$ and $-\infty+-\infty$ behaves well, returning infinity for a given endpoint in the result.
- Adding endpoints ∞ and $-\infty$ is something we cannot do. We have to fall back to special value $\pm\infty$ in the final result to indicate error.
- A bound will only ever be closed if both arguments also represent closed bounds.

One difference compared to Type I addition is that we have to handle special value $\pm\infty$ which does not exist in a Type I environment. Type I unums can fall into illegal states, in which cases the unum environment raises an exception [13, p. 113]. We decide against exceptions here as that would add another layer to valid arithmetic. Instead we return a silent error in the form of $\pm\infty$.

3.4.2 Posit Rounding Introduces Error

The rules listed in Figure 3.4 are intuitive and easy to follow column by column. The big problem is that each individual operation in Figure 3.4 is done in posit arithmetic. This introduces rounding error. Type I arithmetic does not have this problem as unums can adapt accuracy as necessary. But posits do no such thing. Consider the following example

$$(1/16, 2) + (1, 4) \tag{3.13}$$

in the $V_{4,1}$ environment (Appendix B). Think about how this problem might be solved with the rules as defined so far. According to the rules in Figure 3.4, above sum results

in a left bound

$$x = (1/16 + 1 \tag{3.14}$$

and right bound

$$y = 2 + 4). \tag{3.15}$$

The notation used here, based on The End of Error, is a bit odd but should be intuitive. We have to keep in mind that we are computing left and right bounds of a given result. As such the parentheses (open or closed) are part of the result.

The real issue is the following. Posit arithmetic introduces a problem here as we arrive at $x = 1/16 + 1 = 1$ and $y = 2 + 4 = 4$, both rounded to nearest. Which means that the final result, computed based on the rules in Figure 3.4, is

$$(1/16, 2) + (1, 4) = (1, 4) \tag{3.16}$$

which is not correct. Evaluating Equation 3.13 with perfectly accurate interval arithmetic tells us that the result is anywhere inside interval

$$(1/16 + 1, 2 + 4) = (17/16, 6). \tag{3.17}$$

Surely the right answer in a $V_{4,1}$ environment should be

$$(1/16, 1) + (1, 4) = (1, 16). \tag{3.18}$$

Because posit arithmetic introduces rounding error, a naive definition of addition based on Type I rules does not result in acceptable results. We have to be a bit more creative.

3.5 Error Interval Notation

To find a working definition of valid arithmetic, we first have to think about what intervals really mean. When we write

$$(x, y) \tag{3.19}$$

that represents the interval starting just after point x on the number line and ending just before y. We find that we can rewrite any such interval to use only closed bounds. Put in concrete terms, we can rewrite

$$[x, y] = [x, y] \tag{3.20}$$
$$[x, y) = [x, y - \varepsilon] \tag{3.21}$$
$$(x, y] = [x + \varepsilon, y] \tag{3.22}$$
$$(x, y) = [x + \varepsilon, y - \eta] \tag{3.23}$$

Figure 3.5: Addition of posits p and q can yield a result that was rounded down when the exact result $p+q$ is not perfectly representable. If we want to be exact, then we can rewrite this sum as $(p +_p q)+\varepsilon$ where ε indicates that the result is to the right on the number line.

Figure 3.6: Addition of posits p and q can yield a result that was rounded up when the exact result $p+q$ is not perfectly representable. If we want to be exact, then we can rewrite this sum as $(p +_p q)-\varepsilon$ where ε indicates that the result is to the left on the number line.

where ε and η are positive non-zero errors. Choosing Greek letters such as ε is no accident as it draws inspiration from infinitesimals found in early calculus [30]. But it is important to keep in mind that errors ε and η are not infinitesimal, they are only unknown. This *error interval notation* fits nicely with our understanding of regular valids. Valid

$$v = (p,q) = \{p^u; q^u\} = [p+\varepsilon, q-\eta] \tag{3.24}$$

represents some value that appears some uncertain amount ε after start p and some η before end q.

3.5.1 Error Interval Posit Arithmetic

We can also extend standard posit arithmetic to use error interval notation. This allows us to keep track of rounding information, not unlike the u-bit from unum arithmetic. As an example, consider the $P_{4,1}$ type (Appendix B). Posit arithmetic will return

$$4 +_p 2 = 4 \tag{3.25}$$

because the accurate result $4+2=6$ cannot be represented and as such the result is rounded to closest point 4. We use the $+_p$ notation to indicate that addition is done in term of posit arithmetic and not perfectly accurate math. In particular, we rounded the correct result 6 down to 4. Rounding down to p means that the result is actually something

ever so slightly more than p, that is

$$4+2=4+\varepsilon \tag{3.26}$$

rather than just a plain 4 (Figure 3.5). This also works the other way around. If we return a result that was actually rounded up to q, we better write the result as $q-\varepsilon$. For example, this is the case for

$$\frac{1}{2}+\frac{1}{4}=1-\varepsilon \tag{3.27}$$

in the $P_{4,1}$ type (Figure 3.6). This notation is not unlike what is provided by the u-bit. But error interval notation will prove useful when computing more complicated sums as error can be resolved back to interval bounds. For now we can sum up above rules as

$$p+q=\begin{cases} p+_p q & \text{if } p+_p q \text{ yields exact result} \\ (p+_p q)+\varepsilon & \text{if } p+_p q \text{ is rounded down} \\ (p+_p q)-\varepsilon & \text{if } p+_p q \text{ is rounded up} \end{cases} \tag{3.28}$$

for posit addition that keeps track of rounding.

3.5.2 Combining and Simplifying Errors

Error interval notation is useful because computing the sums and products of more complicated terms is straight-forward. For example, sum

$$(a+\varepsilon)+(b+\eta)=a+b+\varepsilon+\eta \tag{3.29}$$

can be simplified to

$$a+b+\varepsilon+\eta=a+b+\varepsilon \tag{3.30}$$

because errors ε and η are of no particular value. Rather they only tell us that the result is *somewhere* afterwards. In total, we get the following simplification rules:

$$\varepsilon_0+\varepsilon_1+\cdots+\varepsilon_n=\varepsilon \tag{3.31}$$

$$-\varepsilon_0-\varepsilon_1-\cdots-\varepsilon_n=-\varepsilon \tag{3.32}$$

$$\varepsilon_0+\cdots+\varepsilon_n-\varepsilon_{n+1}-\cdots-\varepsilon_m=\pm\varepsilon \tag{3.33}$$

Adding or subtracting any number of unknown errors can be reduced to just some positive or negative error. But when dealing with both positive and negative errors, we have no idea about the concrete result. The only honest thing we can say is that there is some error $\pm\varepsilon$ of uncertain sign. While the u-bit only knows two states (certain and uncertain), error interval notation can represent a total of four states. All illustrated in Figure 3.7.

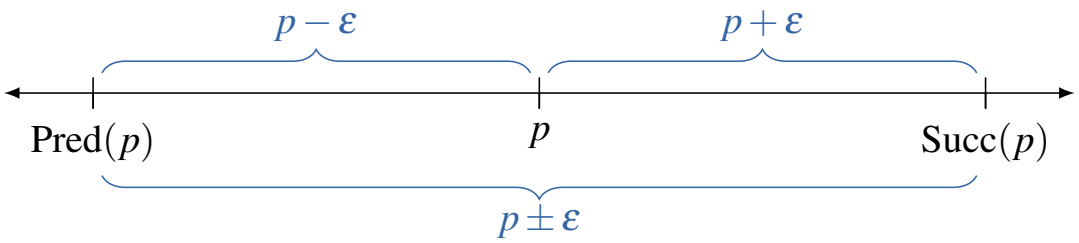


Figure 3.7: Error interval notation allows us to indicate that a given value (1) is exactly p or that it (2) comes before, (3) after or (4) either before or after p.

3.5.3 Resolving Error Intervals to Tiles

Error interval notation is only meant as an intermediate representation. In particular, it will prove useful in our definition of valid arithmetic. Given valid v rewritten in error interval notation

$$v = [s_{lo},\ s_{hi}], \tag{3.34}$$

we now need to think about we can resolve v back to a standard valid encoding. Depending on whether we are resolving back to left bound s_{lo} or right bound s_{hi}, the rules are slightly different. For the lower bound we get

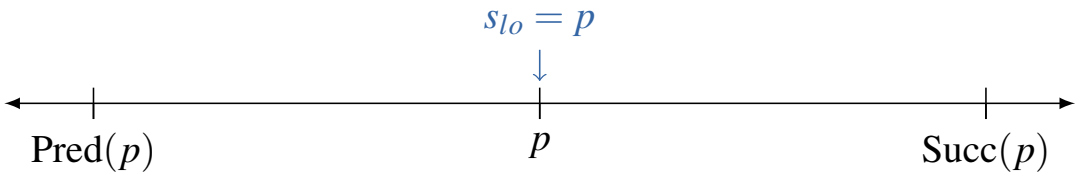
$$\text{Resolve}_{lo}(s_{lo}) = \begin{cases} [\,p & \text{if } s_{lo} = p \\ (\,p & \text{if } s_{lo} = p+\varepsilon \\ (\,\text{Pred}(p) & \text{if } s_{lo} = p-\varepsilon \\ (\,\text{Pred}(p) & \text{if } s_{lo} = p\pm\varepsilon \end{cases} \tag{3.35}$$

and for the upper bound we get

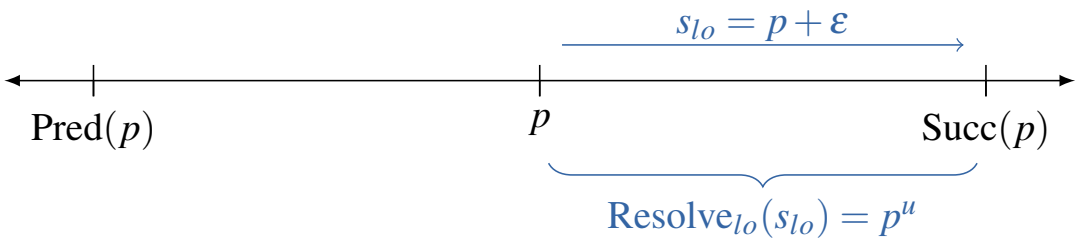
$$\text{Resolve}_{hi}(s_{hi}) = \begin{cases} p\,] & \text{if } s_{hi} = p \\ \text{Succ}(p)\,) & \text{if } s_{hi} = p+\varepsilon \\ p\,) & \text{if } s_{hi} = p-\varepsilon \\ \text{Succ}(p)\,) & \text{if } s_{hi} = p\pm\varepsilon \end{cases} \tag{3.36}$$

instead. Function Succ (and Pred) work as expected in that they return the successor (or predecessor) of given posit on the number circle.

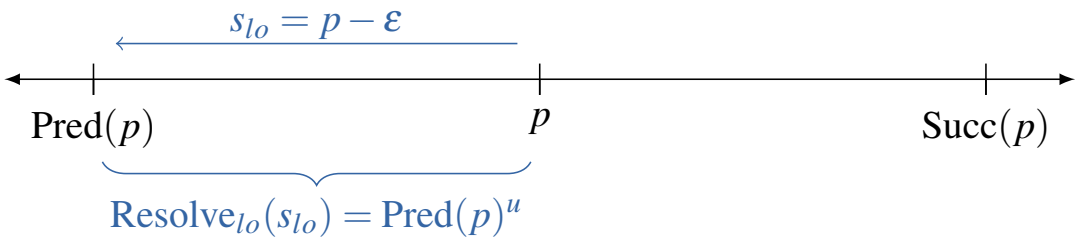
The easiest way to understand the Resolve functions is to look at the illustrations in Figure 3.8. In this figure, we focus on resolving left bounds with function Resolve_{lo}, but similar things apply to Resolve_{hi} as well. In general, we always use the smallest possible bound. When the result is uncertain, we have to be conservative and pick the greatest possible bound in the given situation.



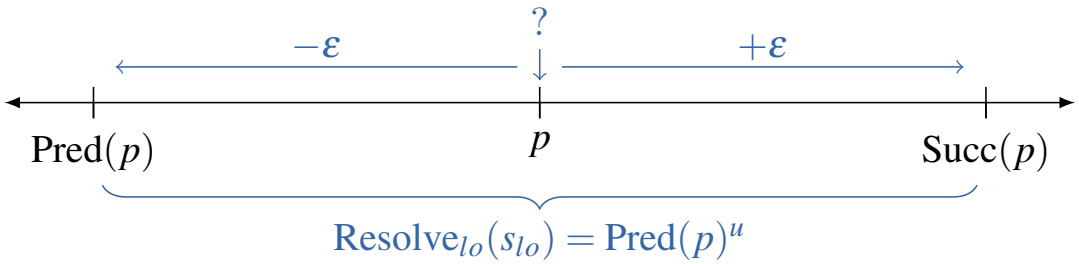
(a) Resolving exact left bound $s_{lo} = p$ results in a closed left bound $[p$. This case is simple as the result is perfectly representable by posit p.



(b) Resolving left bound $s_{lo} = p + \varepsilon$. The result is open interval $(p$, that is posit p with u-bit set to 1. The u-bit indicates the range between p and its right neighbor Succ(p).



(c) Resolving left bound $s_{lo} = p - \varepsilon$. The result has to be open interval (Pred(p), that is the predecessor of posit p with u-bit set to 1. Because the u-bit always refers to the successor of a given endpoint, we have to take a step back (function Succ) to resolve bound s back to a posit endpoint.



(d) Resolving left bound $s_{lo} = p \pm \varepsilon$. Because we are not at all sure about the concrete result, we have to be conservative and resolve s to (Pred(p) as that covers all possibilities $s = p - \varepsilon$, $s = p$ and $s = p + \varepsilon$.

Figure 3.8: Resolving error interval s_{lo} back to a left valid bound. Here we look at all possible cases, that is $s = p$, $s = p + \varepsilon$, $s = p - \varepsilon$ and $s \pm \varepsilon$.

3.6 Addition Based on Error Intervals

Armed with this knowledge, we can now define valid addition. Not only will the returned bounds be correct, but they will also be as small as is reasonably possible. A caveat is that this definition currently only works for regular valids. Irregular valids are a whole different beast, not covered by our definition of valid arithmetic.

Algorithm 1. *Given two non-special and regular valids v and w of matching valid type, this algorithm computes sum $s = v + w$ using error interval notation.*

1. *Rewrite v and w in error interval notation, viz.*

$$v = [a, b] \qquad w = [c, d]. \tag{3.37}$$

2. *Compute lower bound s_{lo} and upper bound s_{hi} with rules familiar from traditional interval arithmetic, that is compute*

$$s_{lo} = a + c \qquad s_{hi} = b + d \tag{3.38}$$

   in error interval notation that keeps track of rounding.

3. *Convert lower and upper bound back to valid tiles using the respective* Resolve *function. We get*

$$s = \{\ \text{Resolve}_{lo}(s_{lo})\ ;\ \text{Resolve}_{hi}(s_{hi})\ \}. \tag{3.39}$$

   as the final result.

To compute sum $s = v + w$, we first convert valids v and w to error interval representation. We do the math while tracking rounding information and then finally resolve both start and end back to a standard valid. Using an intermediate step allows us to ensure that the returned bound is only as big as it needs to be.

3.6.1 A Simple Example

To illustrate our definition of valid addition, we will take a look at two examples. Both examples take place in the $V_{4,1}$ environment (Appendix B). For the first example, we look at the challenging task of computing

$$1 + 1 \tag{3.40}$$

for which we will apply each of the three steps defined by Algorithm 1.

1. First we have to rewrite Equation 3.40 to error interval notation. We arrive at

$$1 + 1 = [1, 1] + [1, 1] \tag{3.41}$$

because number 1 is perfectly representable with the given posit and as such valid type.

2. We now apply the standard addition rule from interval arithmetic while keeping track of rounding information.

$$s_{lo} = 1 + 1 = 1 + \varepsilon \tag{3.42}$$
$$s_{hi} = 1 + 1 = 1 + \varepsilon \tag{3.43}$$

In either case we round the mathematically correct result 2 down to posit 1. Unlike with posit arithmetic, we do not discard this rounding information. Rather we keep it for later use.

3. Finally, we have to resolve s_{lo} and s_{hi} back to a standard valid. We get

$$\text{Resolve}_{lo}(1 + \varepsilon) = (1 \tag{3.44}$$
$$\text{Resolve}_{hi}(1 + \varepsilon) = \text{Succ}(1)) = 4) \tag{3.45}$$

and as such the final result is

$$1 + 1 = (1, 4). \tag{3.46}$$

This first example shows that our definition of valid addition can be honest about the result. With valids we do not have to present the user with a rounded value, rather we return the smallest possible bound on the result.

3.6.2 A More Involved Example

Admittedly, even a hypothetical cell arithmetic could return the previous result. So let us look at an example that genuinely requires the use of valids, viz.

$$(-4, 1) + [1, 4] \tag{3.47}$$

for which we will again run through all three steps of valid addition.

1. First we need to convert the open intervals to closed intervals. We get

$$(-4, 1) + [1, 4] = [-4 + \varepsilon, 1 - \eta] + [1, 4] \tag{3.48}$$

where ε and η are uncertain errors. We can already see that this example is going to be a bit more involved.

2. Nevertheless, the basic algorithm is the same as we now apply the addition rules of

traditional interval arithmetic.

$$s_{lo} = -4 + \varepsilon + 1 = -4 + 1 + \varepsilon \tag{3.49}$$
$$s_{hi} = 1 - \eta + 4 = 1 + 4 - \eta \tag{3.50}$$

Left sum s_{lo} and right sum s_{hi} need to be simplified before we can continue any further. We begin by running the standard posit addition while tracking rounding. We get

$$-4 + 1 = -4 + \tau \qquad \text{(rounded down from } -3 \text{ to } -4) \tag{3.51}$$
$$1 + 4 = 4 + \mu \qquad \text{(rounded down from 5 to 4)} \tag{3.52}$$

which plugged back into Equation 3.49 and Equation 3.50 yields

$$s_{lo} = -4 + 1 + \varepsilon = -4 + \tau + \varepsilon = -4 + \varepsilon \tag{3.53}$$
$$s_{hi} = 1 + 4 + \eta = 4 + \mu - \eta = 4 \pm \varepsilon \tag{3.54}$$

where we also simplified the individual errors to just one term each. For the left sum s_{lo}, we were able to collapse all errors into just $+\varepsilon$. In the case of the right sum s_{hi}, we cannot know the sign of the specific error. The result could be left or right of -4 on the number line.

3. This uncertainty is resolved when converting back to valid bounds in the final step. We get

$$\text{Resolve}_{lo}(s_{lo}) = (-4 \tag{3.55}$$
$$\text{Resolve}_{hi}(s_{hi}) = \text{Succ}(4)) = 16) \tag{3.56}$$

and as such the final result of this example is

$$(-4, 1) + [1, 4] = (-4, 16) \tag{3.57}$$

which given the low resolution valid type is the best answer we can give. It is also the smallest interval in that the result does not contain any cells which cannot be part of the result.

We finally managed to define valid addition. While our initial approach based on Type I rules failed, using error interval notation proved to be productive. Error interval notation allows us to return the smallest possible bound, including cases where we are unsure about the concrete error.

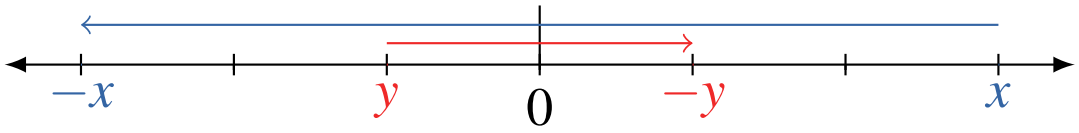
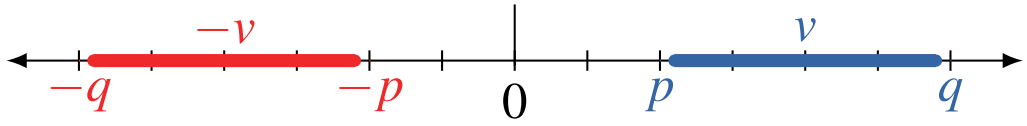
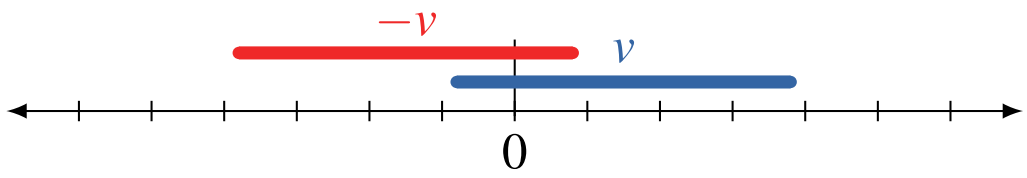


Figure 3.9: Taking the negation $-x$ of some real value x is equivalent to mirroring value x at the center of the number line.



(a) Taking the negation of regular valid v which moves both endpoints to the negative.



(b) Taking the negation of regular valid v which contains both negative and positive values.

Figure 3.10: Taking the negation $-v$ of some regular valids v means mirroring each endpoint at the center of the number line.

3.7 Subtraction Based on Addition

From $\mathbb{R}$ we know that subtraction can be implemented in terms of addition, that is

$$x - y = x + (-y). \tag{3.58}$$

We believe that it is possible to also apply this rule to valid arithmetic, at least for regular valids. For this to work however we first have to think about what it means to take the negation

$$-v \tag{3.59}$$

of some given valid v. To gain an intuition for this problem, let us first look at the real numbers. On $\mathbb{R}$, taking the negation of some value x is equivalent to mirroring value x on the number line (Figure 3.9). We find that we can apply this principle to valids as well.

Defintion 21. *Given regular valid $v = \{s\ ;\ t\}$ of some arbitrary valid type, we define its negation to be $-v = \{-t\ ;\ -s\}$.*

For negation of regular valid $x = \{s\ ;\ t\}$, we flip around and negate both tiles s and t. Figure 3.10 illustrates valid negation with two examples. With negation in hand, we will now be able to see that valid subtraction can be implemented in terms of valid addition.

Matching Rules

Valid arithmetic is an evolution of interval arithmetic and as such we again think about the standard interval arithmetic rules. In traditional interval arithmetic, subtraction of intervals $x = [a,b]$ and $y = [c,d]$ is defined to

$$x - y = [a,b] - [c,d] = [a-d, b-c] \quad \text{[29, p. 1048].} \tag{3.60}$$

Applied to valid arithmetic, we see that our definition of valid negation maps nicely to this formula as subtraction of valids $v = \{a;b\}$ and $w = \{c;d\}$ amounts to

$$v - w = v + (-w) = \{a\ ;\ b\} + \{-d\ ;\ -c\} = \{a-d\ ;\ b-c\}. \tag{3.61}$$

Subtraction of valids can easily be implemented in terms of addition. As such we do not have to think of a separate definition of subtraction. Instead we use our algorithm for valid addition to define valid subtraction. The relationship between addition and subtraction familiar from $\mathbb{R}$ also holds for valids.

3.8 Multiplication Based on Error Intervals

With addition and subtraction out of the way, we are left to work out multiplication and division. Just as we defined addition on regular valids using error intervals, we can also define multiplication. Again, we fall back to the core rules of interval arithmetic and Type I unums. In particular, we base our definition of valid multiplication on the interval rule

$$[a,b] \cdot [c,d] = [\min(ac,ad,bc,bd), \max(ac,ad,bc,bd)] \tag{3.62}$$

adopted from traditional interval arithmetic [29, p. 1049]. Interval multiplication amounts to evaluating candidates

$$ab,\ ad,\ bc,\ bd \tag{3.63}$$

and picking out the smallest and biggest one such that the returned interval contains all possible solutions [13, p. 128].

Algorithm 2. *Given two non-special and regular valids v and w of matching valid type, this algorithm computes product $m = v \cdot w$ using error interval notation.*

1. *Rewrite v and w in error interval notation, viz.*

$$v = [a,b] \qquad w = [c,d]. \tag{3.64}$$

2. *Compute all candidates*

$$ac,\ ad,\ bc,\ bd \tag{3.65}$$

in error interval notation that keeps track of rounding.

3. *Set m_{lo} to the minimum of all four candidates and m_{hi} to the maximum of all candidates, that is*

$$m_{lo} = \min(ac, ad, bc, bd), \tag{3.66}$$
$$m_{hi} = \max(ac, ad, bc, bd). \tag{3.67}$$

4. *Convert lower and upper bound back to valid tiles using the respective* Resolve *function. We get*

$$m = \{\ \text{Resolve}_{lo}(m_{lo})\ ;\ \text{Resolve}_{hi}(m_{hi})\ \}. \tag{3.68}$$

as the final result.

The main differences compared to Algorithm 1 for addition are found in Step 2 and 3. Here we first have to compute four candidates and then pick out the minimum and maximum. Aside from that, valid addition and multiplication work in very much the same way. Because error interval notation allows us to keep track of rounding, we can exploit posit arithmetic to do the heavy lifting.

3.8.1 Example

Again, we want to illustrate our algorithm with an example. We will keep it simple and compute

$$m = \frac{1}{16} \cdot 2 \tag{3.69}$$

in the $V_{4,1}$ environment (Appendix B).

1. We begin by applying Step 1, that is rewriting both arguments in terms of error intervals. Here, we get

$$[1/16, 1/16] \cdot [2, 2]. \tag{3.70}$$

2. As for the second step, we will now compute all candidates while keeping track of rounding information, viz.

$$ac = ad = bc = bd = \frac{1}{4} - \varepsilon. \tag{3.71}$$

3. In this simple example, all candidates are identical which also means that

$$m_{lo} = m_{hi} = \max(ac, ad, bc, bd) = \min(ac, ad, bc, bd) = \frac{1}{4} - \varepsilon. \tag{3.72}$$

4. All that is left is that we have to resolve error intervals m_{lo} and m_{hi} to get the valid result. We arrive at

$$m = \left(\frac{1}{16}, \frac{1}{4}\right), \tag{3.73}$$

the best possible result for our query in this low resolution environment.

We see that valid multiplication works in very much the same way as addition. The only difference between the two is that for multiplication, we have to compute and sort four candidates ac, ad, bc and bd. Converting to and from error interval notation remains identical. Just like with valid addition, multiplication should not return valids that are bigger than necessary.

3.9 Valid Division Based on Multiplication

The last remaining arithmetic operation is division. We spent less time on this operation than we would have liked. As such we cannot present a definition we are perfectly confident in. To compute quotient

$$\frac{v}{w} \tag{3.74}$$

of valids arguments v and w, we actually compute

$$\frac{v}{w} = v \cdot \frac{1}{w}, \tag{3.75}$$

that is we multiply quotient v with the reciprocal of dividend w. Computing the reciprocal of some valid x is a special case and can be achieved with reasonable accuracy by taking the reciprocal of the individual endpoints. That is if valid

$$x = \{t\ ;\ s\} \tag{3.76}$$

is split in tiles t and s, then the reciprocal of x should be given by

$$\frac{1}{x} = \left\{\frac{1}{t}\ ;\ \frac{1}{s}\right\}. \tag{3.77}$$

We are on thin ice here as above definition of valid division has not seen elaborate testing in the limited time allotted to this thesis. We present it mostly as a suggestion for future work on the topic. For what it is worth, above definition has served us well when valid experiments required division.

3.10 Summary

Starting with just the binary format, we presented an in-depth discussion of the valid format. We know that valids are intervals that represent sets of cells. Cells in turn are either concrete values p or intervals (r,s) of neighboring posits r and s. As there are no illegal states a valid can be in, we found it necessary to differentiate between regular and irregular valids. While regular valids are well-behaved intervals, irregular valids contain special value $\pm\infty$. From there, we defined equality and the less-than operation on valids. Finally, we introduced a limited definition of valid arithmetic. It should provide a good foundation for future research.

Valids are a necessary building block if we wish to provide an end of error to computer arithmetic. Indeed with valids, we solve some of the fundamental problems with computer arithmetic. Sure, a given valid type is still limited to some finite number of states. But the u-bit allows valids to be honest about the result. Valids do exhibit jagged accuracy as they are based on posit endpoints, but automatic tracking of rounding means that programmers might not have to think about such problems. The only downside of the core format is that valids contain duplicate patterns that represent the same values. As there are many ways to represent the empty and full set, some operation require special cases.

4 Implementation

With our definition of valids, we now have the full set of Type III formats available for use. But no format is any good if it cannot be used in real world applications and experiments. An implementation is required.

We contribute with a C++ implementation of Type III arithmetic based on the existing `aarith` [31] arbitrary precision number library. It provides a solid base for experimentation and benchmarks. This chapter introduces our implementation, describes the programming interface and internal design.

4.1 Existing Libraries

Since its introduction in 2017, various implementations of the posit and quire format have been worked on. Indeed the "Survey of Posit Hardware and Software Development Efforts" [26] lists more than 25 projects. Here we first take a look at four interesting software libraries.

- The `SoftPosit` [32] library written in C acts as a reference implementation. It implements posit and quire arithmetic. `SoftPosit` is officially endorsed by Gustafson et al. [26] and bindings for various other programming languages exist [33, 34, 35]. That said, only a small set of posit parameters N and ES is supported.

- `bfp` [36] is an early C++ implementation of posit arithmetic. It aims to provide a "human readable posit reference implementation" [36]. Indeed the `bfp` code is easy to understand. But as `bfp` is currently missing proper rounding it does not match the posit specification. Unlike many other C++ libraries, `bfp` does not use template classes. Parameters N and ES are only evaluated during runtime.

- `cppPosit` [37] is a more recent C++ implementation of posit arithmetic. It includes tuned implementations of standard posit types as well as a template posit class with support for arbitrary parameters N and ES. Quire support is missing.

- The `universal` [38] library is written in C++ and under active development with corporate backing. It supports a big number of template parameters N and ES for both posit and quire arithmetic. `universal` does include a `valid` class, but it only implements the rough binary format. Valid comparisons and arithmetic are not part of `universal`.

Each library comes with distinct pros and cons. `SoftPosit` is fast and officially endorsed, but limited to a small number of parameters N and ES. `bfp` is easy to understand,

but development efforts seem to have to stopped before reaching proper rounding support. Likewise, `cppPosit` provides good support for posit arithmetic, but quires and more advanced posit features are not in sight. `universal` is attractive because it sports corporate backing and is under active development. Compared to the alternatives, it is also the most flexible and feature-complete.

All introduced libraries focus on implementing Type III arithmetic. But our goal is to compare IEEE floating points with novel posits and valids. What we need is a flexible base on which we can build our tests and experiments. To provide this foundation, we extended the `aarith` library to support our definitions of Type III arithmetic.

- `aarith` [31] is a C++ library for arbitrary precision arithmetic. It implements integers, fixed point numbers and IEEE-like floating points with arbitrary parameters.

As part of this thesis, we added support for posit, quire and valid arithmetic to `aarith`. It is a fair to ask why one should bother with implementing posit arithmetic yet again. Already many implementations of posits and quires exist. But combining different formats in one library makes it easy to share code between experiments and benchmarks. Code written for `aarith` can be configured to run with a big number of different formats. Extending `aarith` lays down infrastructure that can pay off in the future.

4.2 Programming Interface

When implementing any number format there are two things to think about. On one hand, we have to think about the programming interface exposed to users. On the other hand, we have to think about the internal design of the library. We take a top-down approach here. We first discuss the user interface and then the internals implementing that interface.

Template Classes

`aarith` makes heavy use of C++ class templates [39, pp. 1-44]. A good example is the `aarith::floating_point<E, M>` type parameterized by exponent size `E` and mantissa size `M`. As `aarith` already uses templates for its integer and floating point types, it comes natural to use a similar approach for Type III arithmetic. In total, we extended the `aarith` library with the following three types. They serve as the library's programming interface exposed to users.

- Class `aarith::posit<N, ES>` provides a posit type parameterized by width `N` and exponent size `ES`.
- Class `aarith::quire<N, ES>` provides a quire type for use with the associated posit type of matching parameters.
- Class `aarith::valid<N, ES>` provides a valid type. Under the hood, it uses `aarith::posit<N, ES>` tiles as endpoints.

We only ever require parameters `N` and `ES`. Users only have to think about which posit environment they wish to work with.

Type Aliases

While we support arbitrary parameters `N` and `ES` of type `size_t`, we expect most users to fall back to standard posit types as defined in the current draft of the posit standard [25, p. 6]. For convenience, we provide type aliases

`posit8`, `posit16`, `posit32`, `posit64`

for the standard posit types as well as

`quire8`, `quire16`, `quire32`, `quire64`
`valid8`, `valid16`, `valid32`, `valid64`

for all associated quire and valid types. Instead of manually construing a given posit type, users can simply refer back to these helpful aliases.

Explicit Construction

Instantiating a given posit, quire or valid object is easy as we provide many conversion operators from existing C++ and `aarith` types. We differentiate between two cases: (1) Direct conversion and (2) explicit construction. Direct conversion means that the value of a given number is preserved as well as possible. For example, constructing a posit

```
posit16 p = posit16(15.92f);
```

will create a `posit16` that approximates float value `15.92` as accurate as is possible with the given posit type. This is intuitive and matches the behavior of standard C++ types, for example when converting a `double` to `int`. Explicit construction on the other hand means that we let users define the underlying fields of a given type. For example,

```
uinteger<32> bits = 0x12345;
posit32 q = posit32::from(bits);
```

constructs posit `q` from bit pattern `bits`. Meaning that bit pattern `bits` is imported as-is and as such `q` does not represent value `0x12345` (i.e. 74565). Rather it is a bitwise copy of `bits` into `q`. We use the `::from` naming convention consistently throughout all of our classes. As such users always know whether they are importing an existing number by direct conversion or by explicit construction.

Each unum type overwrites the typical arithmetic operations, making it easy to port existing code and write new one. The combination of easy to use but flexible class templates, convenient type aliases, explicit constructors and operator overloading makes for an straightforward interface.

4.3 Intermediate Representations Simplify Arithmetic

We will now investigate some under the hood details of our implementation. We start with a discussion on how we use intermediate representations to simplify posit arithmetic.

As an example we will look at posit multiplication, but similar things apply to other operations as well.

Float Multiplication

The current draft of the posit standard does not dictate any particular algorithm for posit multiplication [25]. What we can infer is that that multiplication of posits p and q should result in a product pq rounded to the closest concrete posit value. How to achieve this is left to the individual implementors. As posits have a lot in common with floating points, adapting algorithms for float multiplications is the natural choice. Many such algorithms exist [2, pp. 220–223, 3, pp. 103–105] and conceptually they are all very similar. Float multiplication $x \cdot y$ first splits arguments x and y into exponents e and fractions f, viz.

$$x = f_x \cdot 2^{e_x} \quad \text{and} \quad y = f_y \cdot 2^{e_y}. \tag{4.1}$$

We can then compute product $x \cdot y$ by multiplying the individual fractions and adding the respective exponents, that is

$$f_{xy} = f_x \cdot f_y \quad \text{and} \quad e_{xy} = e_x + e_y. \tag{4.2}$$

The final result is

$$x \cdot y = f_{xy} \cdot 2^{e_{xy}}. \tag{4.3}$$

Both fraction and exponent are stored as standard integers. As such we can use standard integer arithmetic to compute the two terms f_{xy} and e_{xy}. In a final step, the result from Equation 4.3 is converted back to a float encoding. Introducing an intermediate step simplified the whole process.

Posit Multiplication

We use a similar approach for posit multiplication. The only difference is that we cannot directly import the scale factor from a given posit as we did with floats. Because posits encode their scale factor as a product of (1) regime and (2) explicit exponent, we first combine both into a unified scale e. Fraction f works just as it does with floats and as such can be imported directly from the posit.

Intermediate Classes

Intermediate representations can be very useful and our software design reflects this. The internal `posit_parameters` class represents a posit decoded into parameters `scale` and `fraction`. Arithmetic is done in parameterized form and only converted back to plain posit for storage. This not only makes our code easier to understand, it also allows us to reuse various logic for all four arithmetic operators. In particular, splitting up a given posit into parameter and applying the necessary rounding is all handled by the `posit_parameters` class. Other libraries such as `universal` take a similar approach, also utilizing a dedicated class to represent a parameterized posit. The same applies to silicon. Hardware implementations of posit arithmetic may use a dedicated posit extraction phase that splits a given posit into scale and fraction [14, pp. 53-63].

Intermediate representations simplify our implementation of Type III arithmetic. Individual steps required to perform typical arithmetic operations are separated into dedicated classes. Structuring code in this way allows for easy reuse of complicated logic such as decoding and rounding.

4.4 Reusing aarith Datatypes

`aarith` already comes with support for many different arithmetic types. In particular, the current development version supports `integer`, `uinteger`, `fixed_point` and `floating_point` arithmetic. We were able to greatly take advantage of these existing classes. For example, the `posit` type is made up of just one `uinteger` of matching width. Typical bitwise operations, comparisons and the two's complement are all provided by `aarith`.

Quire Implementation

As for the `posit` and `valid` classes, most logic had to be implemented from scratch. But this was not the case for the `quire`. Recall that the quire is nothing but a fixed point accumulator for posit arithmetic. As `aarith` already provides a bare-bones implementation of fixed point numbers, the majority of operations are simply deferred to the `fixed_point` class. This goes a long way. For example, the logic for printing out the value of a given posit p first converts p to a quire representation and then reuses the code for printing fixed point numbers. We only had to write the annoying string conversion code once for fixed point. String conversion for the quire and as such for posits come for free. Throughout our implementation we reuse existing `aarith` code to great effect.

4.5 Keeping Track of Rounding

Rounding is an important part of posit arithmetic and getting it right can be tricky. In addition to getting correctly rounded posit results, we also need to be able to track rounding. Remember how our definition of valid arithmetic relies on error interval notation. To get it right, we need to be aware of rounding error introduced by posit arithmetic.

Additional Bits For Rounding

All posit arithmetic is done in parameterized form. But the underlying representation split in `fraction` and `scale` actually uses more bits than strictly speaking necessary to represent the plain posit. Just as quires circumvent the rounding problem by using high resolutions, our `posit_parameters` does all arithmetic with an increased number of bits. Converting back to plain `posit` means cutting off the superfluous bits. If the cut bits were all zero, we know that the result was accurate enough in terms of posit arithmetic. If the cut off bits were non-zero, we round the result as necessary. This is an approach adapted from code in the `universal` library which also uses additional bits to keep track of rounding.

Tracking Rounding

What `universal` does not do to our knowledge is that our implementation exposes rounding information to upper layers. In our `aarith`-based implementation, converting back from parameterized form to posit returns not just a plain `posit`, but also rounding information encoded in a `rounding_event`, viz.

```
enum class rounding_event
{
    NOT_ROUNDED,
    ROUNDED_DOWN,
    ROUNDED_UP
};
```

```
template<size_t nbits, size_t es>
posit<nbits, es> sin(posit<nbits, es> x) {
    return posit<nbits, es>(std::sin(double(x)));
}
```

Listing 1: Code from the `universal` arithmetic library for computing the sine of a given posit `x`. File `trigonometry.hpp`, revision `314b1c80`.

which can be used when necessary. For example, we take advantage of rounding information in our implementation of valid arithmetic. When rounding information is not required (e.g. for plain posit arithmetic), the `rounding_event` is discarded.

Rounding can be tricky to get right. An easy solution inspired by `universal` is to compute the result in increased resolution and then evaluate the cut off bits. Unlike `universal`, we do not always discard rounding information. This allows us to keep rounding into account when necessary, for example when implementing valid arithmetic.

4.6 Mathematical Functions

Mandated Functions

The draft posit standard lists a long number of mathematical functions a posit environment is expected to support [25, pp. 9-10]. The list includes simple operations such as the absolute value or floor and ceil of a given posit p. Our `aarith`-based implementation supports many of these operations. The draft also mandates various more advanced functions. This includes the square root, logarithms and trigonometric functions. Our implementation supports only a subset of these more advanced operations. In particular, our port of `aarith` supports the `sqrt`, `log`, `sin`, `cos` and `tan` operations on posits.

Delegating To Floats

Implementing such functions in an efficient and accurate manner is the topic for publications of their own. Notably, `universal` currently delegates trigonometric operations on posits to their floating point counterparts. Listing 1 illustrates this with an example from the `universal` code base. For low resolution posit environments, this should result in accurate and quick results. But when N is big, significant rounding error may be introduced. `SoftPosit` previously experimented with a similar approach to `universal`, but currently offers no support at all for the trigonometric functions. Despite being an officially sanctioned implementation of posit arithmetic, `SoftPosit` is currently missing most of the mathematical functions mandated by the draft standard.

Custom Code For Posits

Our `aarith`-based implementation favors custom algorithms that natively support posit arithmetic. This does mean added work for us. But because we want our implementation of Type III arithmetic to (1) be independent of floating point and (2) support arbitrary parameters N and ES, custom implementations are a necessity. In total, we implemented the following five functions:

- Function `sqrt(p)` computes the square root of a given posit `p`. We iterate using the Newton-Raphson method [40] until the result converges to a fixed value.

- Function `log(p)` computes the natural logarithm of posit argument `p`. To compute this function, we use Borchardt's algorithm [41], an iterative approach with a bounded number of maximum iterations.
- `sin(p)` computes the sine of posit argument `p` in radians. We use series

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots \quad \text{[42, p. 82]} \tag{4.4}$$

  to approximate the result. This is no perfect solution. Equation 4.4 involves the factorial function $y!$ which greatly limits the number of terms we can evaluate as $y!$ quickly grows to be very large.
- Function `cos(p)` computes the cosine of posit argument `p` in radians. Our implementation uses the equality

$$\cos(x) = \sin\left(\frac{\pi}{2} - x\right) \quad \text{[42, p. 45]}, \tag{4.5}$$

  deferring computation to our implementation of `sin`.
- Function `tan(p)` computes the tangent of posit argument `p` in radians. We use the equality

$$\tan(x) = \frac{\sin(x)}{\cos(x)} \quad \text{[42, p. 44]}, \tag{4.6}$$

  deferring computation to our implementations of `sin` and `cos`.

The main disadvantage of our approach has to be performance. Compared to highly optimized routines readily available for floating point arithmetic, our custom code will be much slower. But as we are interested in evaluating Type III arithmetic itself, proper implementations of the mathematical functions are required.

While our implementation already supports various mathematical functions, we are far from done. Many mathematical operations mandated by the draft standard are currently missing from `aarith`. Providing a full posit environment involves way more than just implementing plain arithmetic.

4.7 Testing Strategy

Test Driven Development

The `aarith` project makes heavy use of automated tests. They ensure that the implementation behaves as expected. We continued this good tradition, often working in a genuine test driven development fashion [43]. Test driven development typically means defining automated test cases *before* working on the actual implementation. A good idea in theory, it often can be difficult to apply in complex real world applications. Luckily, this is not the case for our `aarith`-based implementation. While `aarith` offers support for many

different formats, all operations are small in scope and well-defined . Writing a test case for addition is as simple as checking whether our code returns the correct result for the given arguments.

Exhaustive Tests

As our implementation of posit arithmetic is certainly not the first one, we have a wide array of existing libraries to check against. Indeed our port of `aarith` contains exhaustive tests for addition, subtraction, multiplication and division for the low resolution $P_{8,1}$ and $P_{8,2}$ types. Solutions provided by `SoftPosit` and `universal` serve as a reference. These tests are exhaustive in that they test every possible combination of arguments for the given type. Despite being exhaustive, running these tests is still reasonably quick. We find that on a contemporary workstation, each individual test takes less than ten seconds to complete. As such it is not unreasonable to execute such exhaustive tests with every test run. A worthwhile investment as exhaustive testing provides much needed protection against unexpected regressions.

Randomized Tests

We also run tests for higher resolution types. Unfortunately the exponential number of values to consider makes running exhaustive tests unfeasible in these cases. As an alternative, we opt for a randomized approach where we randomly pick a number of arguments and then check whether `aarith` matches the results provided by the `SoftPosit` reference. Combined with manually defined tests that check certain edge cases, we are quite confident that our implementation of posit arithmetic matches the results of `SoftPosit` and `universal`.

Valid Tests

Testing valid arithmetic is more difficult as no matching implementation is available. What we found useful during debugging was to generate CSV tables that list the results of all possible outputs for a given operation. We would then manually scan the table for obviously wrong results. For each mistake that caught our attention, we created a manual test case. In addition to these manual tests, we also provide some programmatically generated test cases. In particular, we run exhaustive tests where for each combination of posit arguments p and q we check whether valid arithmetic returns a reasonable bound on the result. Our current implementation of valids certainly matches all test cases we provide and does so for a great number of parameters N and ES.

In summary, projects like `aarith` really do lend themselves to exhaustive and automated testing. Exhaustive tests show that our posit implementation matches the `SoftPosit` and `universal` reference for low resolution types. Where exhaustive testing is not feasible, we favor randomized tests. Testing valid code proved more challenging as no reference is available.

4.8 Summary

This chapter first introduced various existing implementations of posit arithmetic. Even so, we decided to extend the `aarith` library with support for Type III arithmetic. It allows us to combine a big number of formats in one library. The programming interface exposed to users is composed of the three types `posit`, `quire` and `valid`. We overload the typical arithmetic operations, making them easy to use. While not yet complete, our implementation already provides a number of mathematical functions such as `sqrt`, `log`

and `sin`. All of which are missing even in the `SoftPosit` reference implementation. To ensure that our implementation returns correct results, extensive testing is used. This includes exhaustive tests for small resolution posit types.

5 Evaluation of the Implementation

Now we want to take a quick look at how our `aarith`-based implementation compares to some other existing implementations. In particular, we will compare our implementation to other projects such as `universal` and `SoftPosit`. We are interested both in performance as well as correctness. Performance is about how quickly the given implementations can run some benchmark. Correctness is about whether the returned result conforms to the spec. While our `aarith`-based implementation was never designed with particular performance goals in mind, we find that it is competitive when compared to an implementation with similar features.

Compared Libraries

The investigated libraries should already be familiar from Section 4.1. We compare our own `aarith`-based implementation to `SoftPosit` and `universal` as they match the most recent definition of the posit standard. We also include the `SoftPosit.jl` library [35] in our first benchmark. `SoftPosit.jl` is a wrapper around `SoftPosit` code written in the Julia programming language. While it builds on the foundation of `SoftPosit`, it supports more types than just plain `SoftPosit`.

Test Hardware

All benchmarks in this section were run on a workstation with an AMD EPYC 7302P 16 core processor and 128 GiB of RAM. We ran Linux kernel version 4.19 and all benchmark code was compiled with GCC version 11.1.0.

5.1 Standard Arithmetic: Add, Sub, Mul, Div

Test Setup

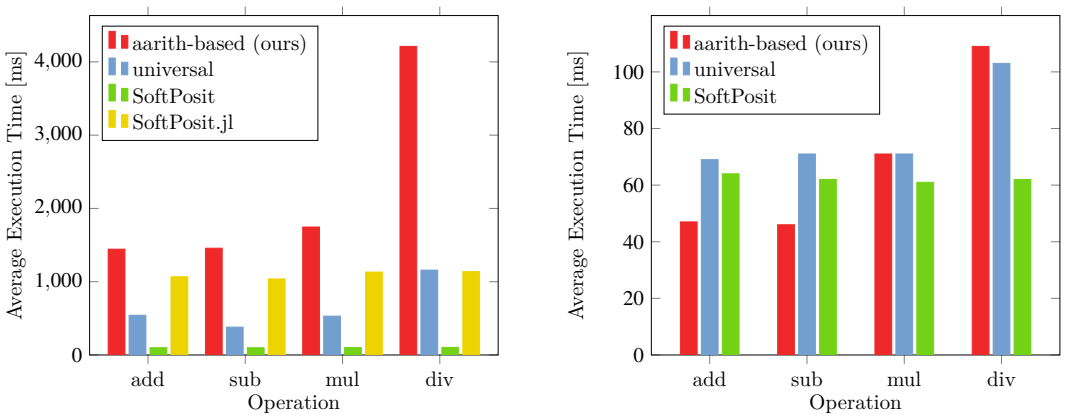
The first benchmark is simple, but important. Here we compare the results of all standard arithmetic operators for all possible combinations of $P_{8,1}$ posit arguments. In particular, we compared the results and performance of `aarith`, `universal`, `SoftPosit` and `SoftPosit.jl`. For this experiment, we would have preferred to use the standardized $P_{8,2}$ type. But as of writing, `SoftPosit` only supports 8 bit posits with exactly one exponent bit, limiting us to the $P_{8,1}$ type.

Correctness

All four implementations returned matching results; a good indication that the all libraries do correct posit arithmetic. As far as correctness is concerned, all libraries pass this test. But in terms of performance, there are notable differences between the compared implementations. Figure 5.1 plots our findings.

Performance Differences

`SoftPosit`, written in C, easily beats the C++ competition when optimizations are disabled during compilation. `SoftPosit` is a rather limited implementation of Type III arithmetic, each supported type can be manually tuned. As such it should not be surprising that `SoftPosit` beats the slower but more flexible implementations provided by us and `universal`. But in most real world applications, there is no reason to disable compiler optimizations. In this case, `SoftPosit` still performs well but the difference is



(a) Running the benchmark without compiler optimizations. Less is better.

(b) Running the benchmark with the highest optimization level. Less is better.

Figure 5.1: Comparing the runtime when exhaustively testing all four arithmetic operations in a $P_{8,1}$ environment. Tests were run a total of four times, we plot the average execution time over all four runs. Because optimizations are hard to control for the interpreted Julia code, `SoftPosit.jl` only appears in the unoptimized case.

way less pronounced. In fact, we find that our `aarith`-based implementation provides quicker addition and subtraction than `SoftPosit` in this case.

Both C++ libraries compare about equally well when compiler optimizations are enabled. While our `aarith`-based implementation provides quicker result for addition and subtraction, `universal` performs better when it comes to division. Finally, `SoftPosit.jl` performs about equally bad for all four operations. We find it likely that this is related to the high startup cost introduced by the Julia runtime [44, pp. 45-49]. However we did not investigate this point any further and as such cannot make a definite statement.

Compiler Optimizations

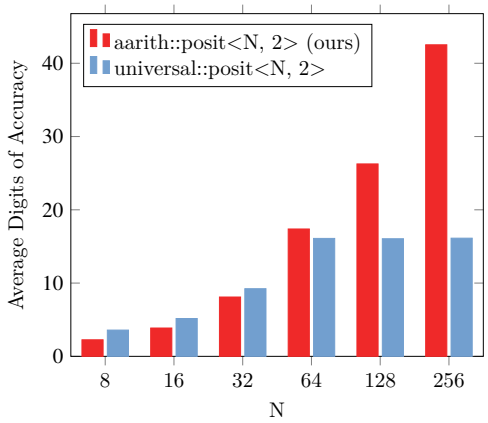
`aarith` greatly benefits from compiler optimizations. Without optimizations, `aarith` is quite the slug compared to `universal`. But enabling compiler optimizations levels the playing field, resulting in equal or better performance of our `aarith`-based implementation compared to `universal` and even `SoftPosit`.

5.2 Mathematical Functions

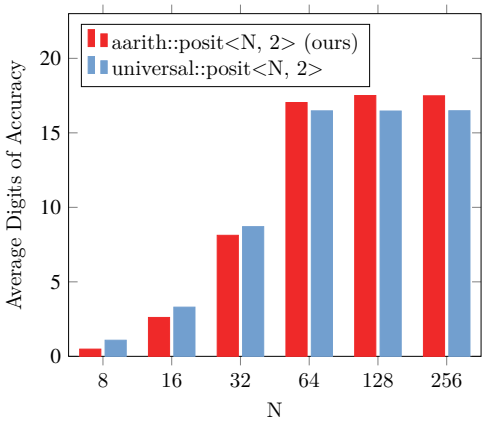
Our `aarith`-based implementation supports some of the required mathematical functions such as `sin`, `cos`, `tan` and `log`. Unlike other libraries in this comparison, we do not rely on the system's floating point unit to perform these operations, rather we provide implementations of our own. In this section, we take a look at how our naive implementations perform compared to the competition.

Test Setup

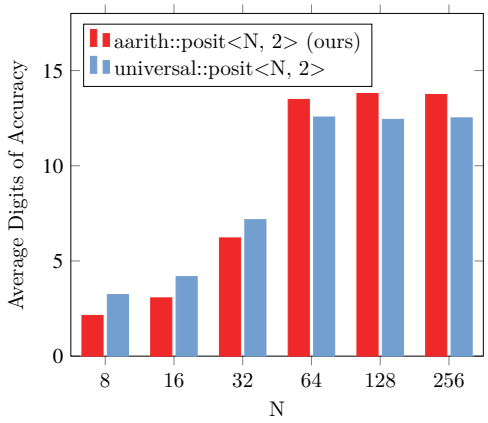
For this example, we compared the accuracy of functions `sin`, `cos`, `tan`, `log` and `sqrt` provided by `universal` and our `aarith`-based implementation with a high resolution reference provided by `apfloat` [45]. `apfloat` is an arbitrary precision floating point



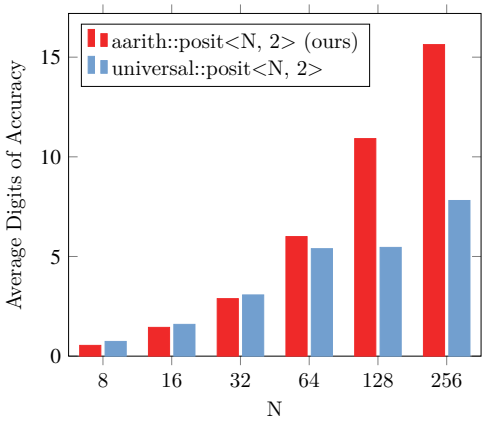
(a) Function `sin`.



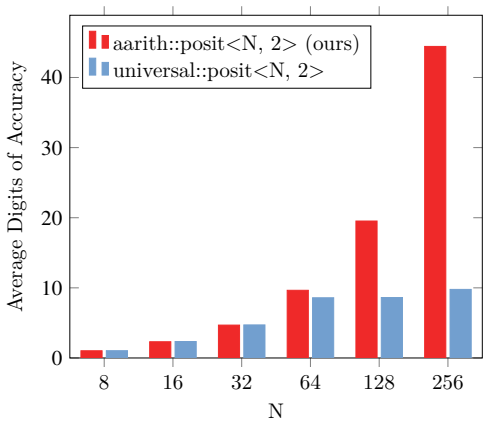
(b) Function `cos`.



(c) Function `tan`.



(d) Function `log`.



(e) Function `sqrt`.

Figure 5.2: Comparing the accuracy of our custom implementations of the mathematical functions with that of `universal`. In this test, we ran each function with 10,000 random arguments and compared their accuracy to a high resolution reference. Our `aarith`-based implementation is competitive and even superior in high resolutions. More is better.

library that includes custom implementations of above functions that scale to arbitrary accuracy. For each given function, we picked a random set of 10,000 argument in a sensible range, comparing the results of our `aarith`-based implementation and `universal` with the high resolution `apfloat` reference.

Performance Differences

Little can be said about performance aside that `universal` eclipses our `aarith`-based implementation. `universal` is multiple orders of magnitudes faster than our code when computing above benchmark. But these results are not at all surprising. Remember that `universal` implements the respective mathematical functions by deferring them to floating point arithmetic (Section 4.6). Deferring computation of the mathematical functions to the system's floating point libraries and hardware exploits decades of optimization work. As such it should not be surprising that our `aarith` based implementation performs much worse.

Correctness

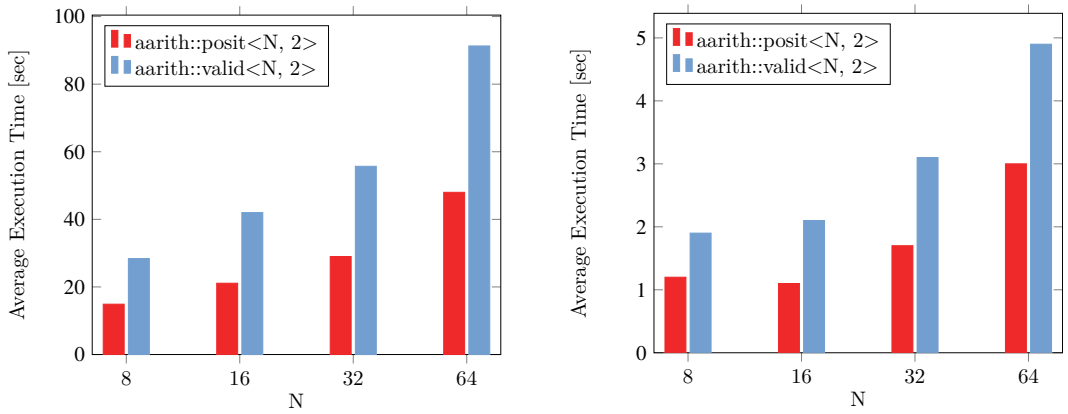
Comparing the returned results of our `aarith`-based implementation and `universal` with the `apfloat` reference reveals the big problem with implementations that rely on `double` arithmetic. Figure 5.2 plots the results for a variety of standard posit types. For the lower resolution types up to width $N = 64$, both our `aarith`-based implementation and `universal` perform about equally well. An indicator that `double` floating point arithmetic is roughly equivalent to 64 bit posit arithmetic. But once we reach higher resolutions, our implementations of `sin`, `log` and `sqrt` outperform the low resolution solutions provided by `universal`. 64 bit floating point simply is not enough when we reach bigger N. We see that our implementations of the mathematical functions can provide superior answers compared to `universal`.

On the downside, our implementations of `cos` and `tan` only show comparable but not superior results to `universal`. In our code, `cos` and `tan` defer computation back to `sin` and we believe this is the problem as accuracy is lost in intermediate steps. A superior solution would most likely not defer these functions to `sin` but rather find fitting approximations specific for `cos` and `tan`. Nevertheless, compared to `universal` our results are competitive.

Deterministic Functions

The current draft of the posit standard generally mandates no particular algorithms for implementing mathematical functions. It only requires implementations to return an result that is as accurate as possible, rounded according to the rules of posit arithmetic. Consequence being that given some mathematical function `f`, there only ever is one concrete solution `y = f(p)` for argument `p` that is strictly speaking acceptable. Our `aarith`-based implementation was not developed with this constraint in mind. Rather we favored easy to write implementations that provide approximations of the given function, ready for use in first experiments. But neither can `universal` make any such guarantees as it only delegates computation to the `double` type. It is up to future research to provide efficient and accurate algorithms for all mandated mathematical functions that are deterministic across implementations.

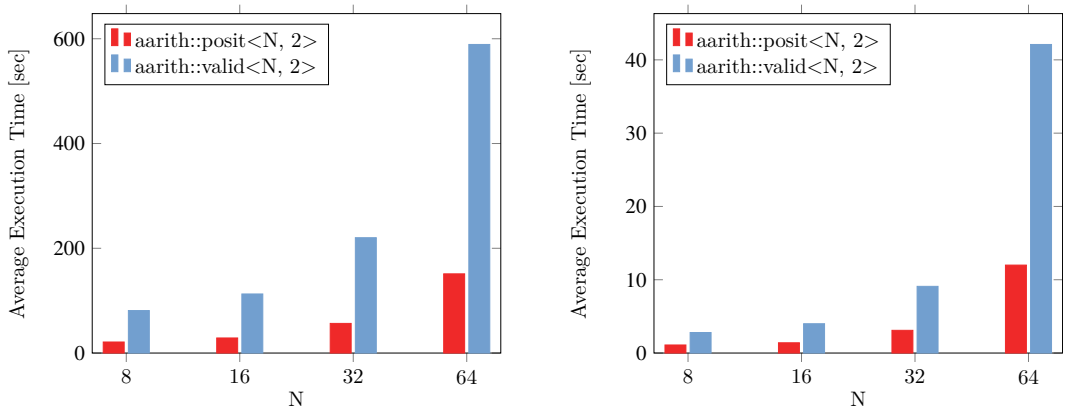
In summary, our implementations of the mathematical functions are orders of magnitude slower than their `double` counterparts. But despite their simple design, our implementations of `sin`, `log` and `sqrt` return more accurate results in high resolutions when compared to the competition. Functions `cos` and `tan` require further improvement.



(a) Running the benchmark without compiler optimizations. Less is better.

(b) Running the benchmark with the highest optimization level. Less is better.

Figure 5.3: Running one million randomized additions with both posits and valids of different widths N. Valid arithmetic about doubles the required time.



(a) Running the benchmark without compiler optimizations. Less is better.

(b) Running the benchmark with the highest optimization level. Less is better.

Figure 5.4: Running one million randomized multiplications with both posits and valids of different widths N. In the worst case, valid arithmetic takes about four times as long as posit arithmetic to complete the given operations.

5.3 Overhead Introduced By Valid Arithmetic

Test Setup

Another metric we were interested in is the overhead introduced by valid arithmetic compared to posit arithmetic. No doubt valid arithmetic involves more individual steps, but the question is to what extend this makes a difference. To find out, we set up a test where we ran a total of 1,000,000 randomized posit and valid additions and multiplications. The results for addition are plotted in Figure 5.3, the results for multiplication in Figure 5.4.

Performance Differences

Our results are encouraging. Valids are not particularly slower than comparable posit arithmetic. In particular, valid addition takes about two times as long as posit addition

while valid multiplication takes about four times as long as the equivalent operation on plain posits. Judging from these numbers, it actually appears to be the case that posit arithmetic is the dominating factor that determines the performance of valid arithmetic. Remember that valid addition involves two posit additions for left bound and right bound. Similarly, remember that valid multiplication involves evaluating four posit candidates from which we have to pick the minimum and maximum value. That fits exactly with our findings.

5.4 Summary

Overall, we are quite pleased with our implementation of unum arithmetic. Despite not being tuned for performance, compiler optimizations result in competitive performance with projects such as `universal`. Mathematical functions are the only real bottleneck. But we are willing to pay that price as our implementations of `sin`, `log` and `sqrt` can be much more accurate.

6 Evaluating Type III Unum Arithmetic

Finally we have the required tools to evaluate Type III arithmetic. This chapter describes various benchmarks and experiments based on our version of the `aarith` library. We try to find out how Type III arithmetic competes with traditional floating point. Where possible we investigate how valids can help in finding accurate results.

Standard Types

Both floating point and posit standards come with a suggested list of standard parameters to use with the respective types [11, p. 8, 25, p. 6]. Appendix C gives on overview of what we consider standard types ready for quick review. Throughout this evaluation, we typically focus on the standard types as they are most likely to be used by the practicing programmer. So for example a "16 bit posit" is a standard $P_{16,2}$ posit type as listed in Appendix C. When non-standard parameters are required, they are explicitly listed as such.

6.1 Problems From Unum Literature

The End of Error [13] proposes Type I unums and has various examples to back up its case. But how does Type III arithmetic compare to its original successor? As we will see, posits fail to provide the correct results as given by Type I arithmetic. Posits are more like traditional floating points than they are like the original unums.

Problem 1. *Iterate*

$$x_{i+2} = 111 - \frac{1130}{x_{i+1}} + \frac{3000}{x_i\, x_{i+1}} \tag{6.1}$$

starting with $x_0 = 2$ and $x_1 = -4$. The stable result is supposed to be $x_i = 6$, but floats fail to provide that answer [13, pp. 173].

Values Do Not Fit

Iterating Equation 6.1 with respective initial values $x_0 = 2$ and $x_1 - 4$ is supposed to yield convergence at $x_i = 6$. Figure 6.1 plots the first 50 iteration steps for various floating point and posit types. We see that both floats and posits do approach the correct result, but eventually jump to a wrong conclusion. The reason in either case is that neither floating points nor posits can represent the fractions in Equation 6.1 correctly. As an example, consider the first fraction,

$$\frac{1130}{x_{i+1}} \tag{6.2}$$

and assume that we have somehow arrived at the point where x_{i+1} reached the desired result of $x_{i+1} = 6$. While both numerator 1130 and denominator 6 are perfectly representable with either 32 bit floats or posits, fraction

$$\frac{1130}{6} = 188.\overline{3} \tag{6.3}$$

on the other hand is not. The result will not be correct and as such true convergence at $x_i = 6$ can never be reached. Type I unums adopt their accuracy to adapt specific needs, but posits are fixed in size and parameters. As such posits suffer from the same problems as traditional floating point arithmetic.

Valids As Warnings

The story hardly gets any better when attempting to solve this problem with valids. Figure 6.2 plots the results for four standard valid types. Again, the results approach the correct value, but at some point all valids diverge so much that computation has to halt as x_i reaches the full set $\circ$. Valids are no magic solution to this problem. However, considering that valids are more of a debugging tool, they did provide some insight. While both floats and posits happily converged at an incorrect value, valids tell us that something is amiss, requiring further investigation.

Problem 1 shows that posits can suffer from the same problems as floating points. Valids can warn the programmer about *something* going wrong, but finding the problem will still require conventional debugging. In this concrete experiment, Type III arithmetic is a genuine regression compared to Type I unums.

Problem 2. *Solve the following systems of equations.*

$$0.25510582x + 0.52746197y = 0.79981812 \tag{6.4}$$
$$0.80143857x + 1.65707065y = 2.51270273 \tag{6.5}$$

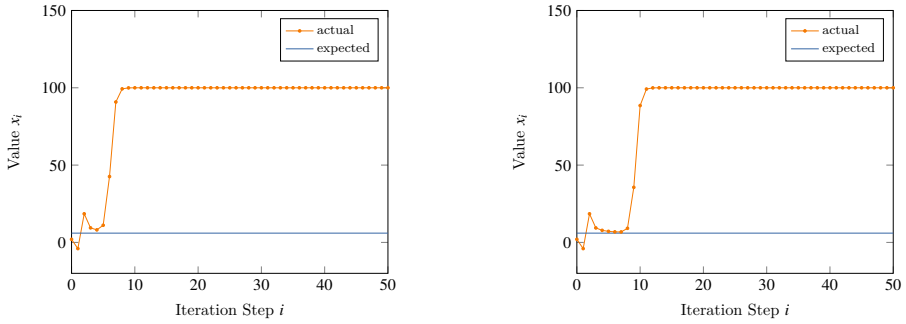
This problem is referred to as "Bailey's Numerical Nightmare" and the expected result is $x = -1$ *and* $y = 2$ *[13, pp. 184].*

We ran Problem 2 with the full set of standard floating point and posit types, from 8 up to 128 bits in width. The results are listed in Table 6.1. The table shows that neither low nor high precision environments provide satisfactory results. Again we have to conclude that posit arithmetic is a genuine regression compared to original Type I unum arithmetic. Neither floating points nor posits allow us to write down a naive solution for Problem 2 that results in acceptable results.

Detecting Errors Manually

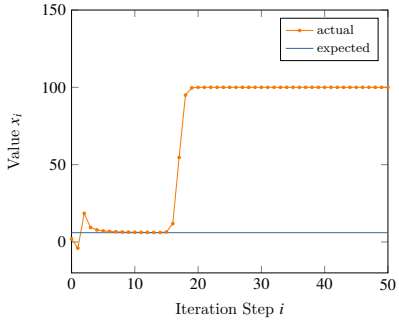
When solving linear equations, a common way to check the results is to plug in the computed solutions into the original equations. Plugging in the computed values for x and y as listed in Table 6.1 into Equations 6.4 and 6.5 yields vastly different results for most types. At the very least, programmers performing this additional sanity check will be alerted of potential errors. But even this can be deceiving. Solving above system of equations with 64 bit posits returns

$$x = 0.317808\cdots \text{ and } y = 1.358904\cdots. \tag{6.6}$$

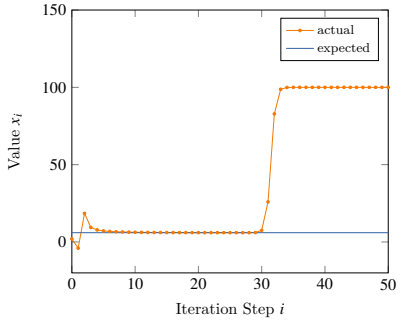


(a) Half precision float.

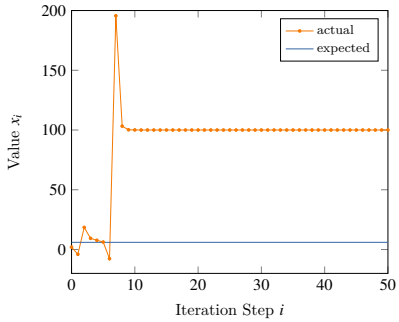
(b) Single precision float.



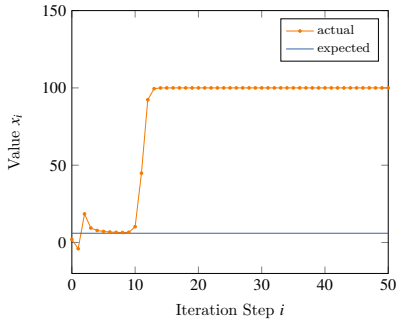
(c) Double precision float.



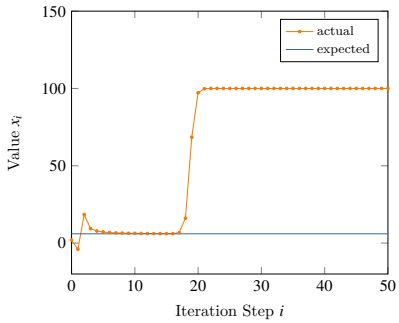
(d) Quad precision float.



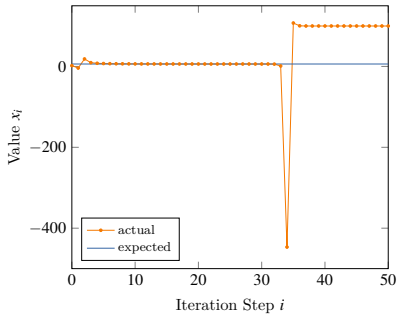
(e) $P_{16,2}$ posit.



(f) $P_{32,2}$ posit.

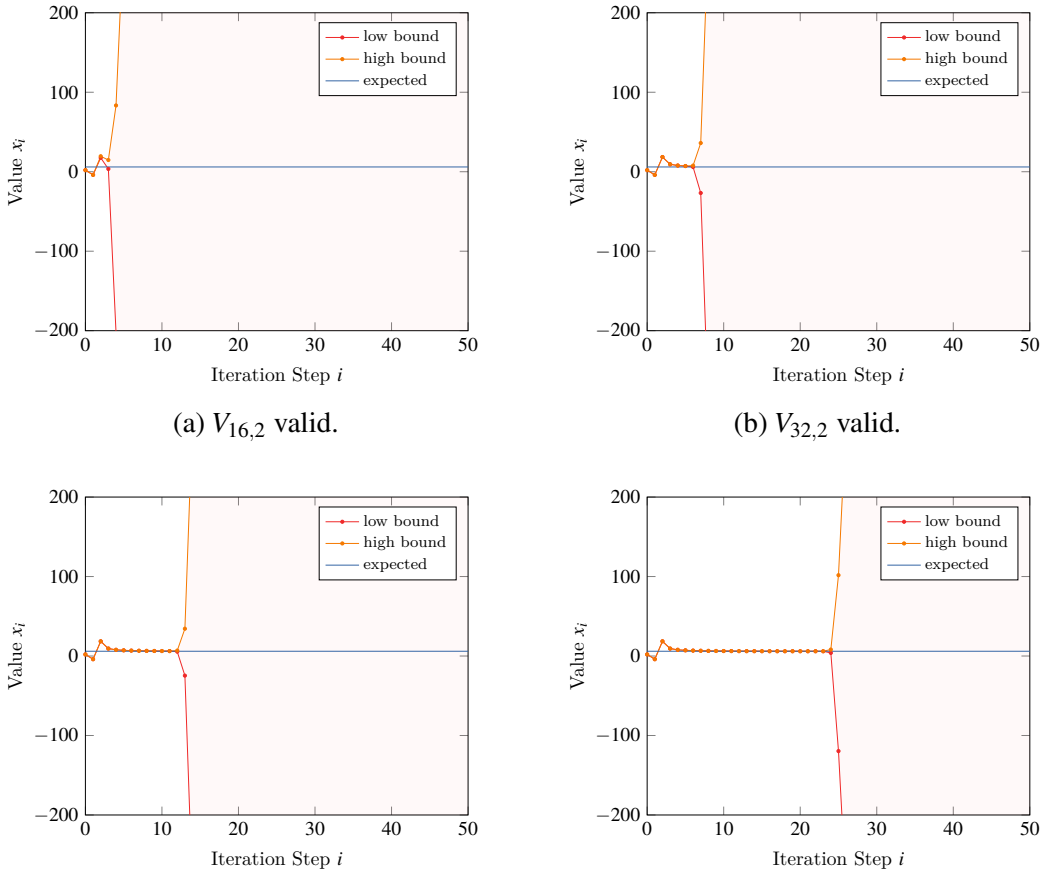


(g) $P_{64,2}$ posit.



(h) $P_{128,2}$ posit.

Figure 6.1: Iterating Equation 6.1 with different data types. The correct result is $x_i = 6$, indicated by a blue line. but even high resolution floating point and posit types eventually converge at the wrong result.



(a) $V_{16,2}$ valid.

(b) $V_{32,2}$ valid.

(c) $V_{64,2}$ valid.

(d) $V_{128,2}$ valid.

Figure 6.2: Iterating Equation 6.1 with valid data types of different lengths. The results are disappointing as computation suddenly returns a huge bound and then aborts, indicated by the sudden jump in upper and lower bounds. The expected result $x_i = 6$ is plotted in blue.

Type	x	y	Type	x	y
expected	-1	2	expected	-1	2
$P_{8,2}$	*NaR*	*NaR*	quarter precision	4	2
$P_{16,2}$	4	0	half precision	0	2
$P_{32,2}$	4	0	single precision	*NaN*	*NaN*
$P_{64,2}$	$0.317808\cdots$	$1.358904\cdots$	double precision	0	$1.333333\cdots$
$P_{128,2}$	$0.323285\cdots$	$1.359995\cdots$	quad precision	$0.3232856\cdots$	$1.359996\cdots$

Table 6.1: Results of Bailey's Numerical Nightmare (Problem 2) computed with various floating point and posit types. Neither floating points nor posits come close to giving reliable answers.

Type	x_1	x_2	Type	x_1	x_2
expected	$-0.02001\cdots$	$-33.3133\cdots$	expected	$-0.02001\cdots$	$-33.3133\cdots$
$P_{8,2}$	0	-32	quarter precision	-16	-16
$P_{16,2}$	$-0.04165\cdots$	$-33.2812\cdots$	half precision	$-0.02081\cdots$	$-33.3125\cdots$
$P_{32,2}$	$-0.02001\cdots$	$-33.3133\cdots$	single precision	$-0.02001\cdots$	$-33.3133\cdots$
$P_{64,2}$	$-0.02001\cdots$	$-33.3133\cdots$	double precision	$-0.02001\cdots$	$-33.3133\cdots$

Table 6.2: Finding the roots of polynomial given by Equation 6.9 (Problem 3) with various floating point and posit types.

Plugging in these values for x and y into the original system leaves us with

$$0.25510582\cdot 0.317808\cdots + 0.52746197\cdot 1.358904\cdots = 0.7978449\cdots \tag{6.7}$$
$$0.80143857\cdot 0.317808\cdots + 1.65707065\cdot 1.358904\cdots = 2.50650388\cdots \tag{6.8}$$

which looks awfully close to the expected results, even though solutions x and y are totally wrong. One can make a case here that the 64 bit posit type is actually the most dangerous because it gives a wrong sense of security. A less controversial conclusion has to be that Problem 2 is solvable with neither floating point nor posit arithmetic. Without thinking about the particular formats and proper numerical analysis, Problem 2 is not solvable in either format. A regression compared to original Type I unums.

Problem 3. *Given polynomial*

$$f(x) = 3x^2 + 100x + 2, \tag{6.9}$$

find the roots of f, that is find those $x_{1,2}$ such that $f(x) = 0$ using the well-known solution

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}. \tag{6.10}$$

The correct solutions are $x_1 = -0.0200120\cdots$ and $x_2 = -33.313321\cdots$. Type I unums

Type	x_1	x_2
expected	$-0.0200120\cdots$	$-33.313321\cdots$
$V_{8,2}$	$(-8, 13)$	$(-56, -16)$
$V_{16,2}$	$(-0.05212\cdots, -0.03128\cdots)$	$(-33.40625, -33.25)$
$V_{32,2}$	$(-0.02001\cdots, -0.02001\cdots)$	$(-33.31332\cdots, -33.31332\cdots)$
$V_{64,2}$	$(-0.02001\cdots, -0.02001\cdots)$	$(-33.31332\cdots, -33.31332\cdots)$

Table 6.3: Finding the roots of polynomial given by Equation 6.9 (Problem 3) with valid arithmetic. The relatively big bounds for the 8 bit types indicate that given computation could benefit from more accuracy. On the other hand, the bounds for 32 bit arithmetic (and above) return very tight bounds, hinting at that sufficient accuracy has been reached.

handle this well as they can adapt their resolution to represent small values [13, pp. 181].

Equation 6.9 looks like harmless high school math, but the difficulty here is that the roots of f are difficult to represent. Table 6.2 shows the resulting values for various floating point and valid types. Attempting to solve this problem with short 8 or 16 bit types will not always return correct results. That said, even at 32 bits, both floats and posits perform well. Floating points have a slight advantage as the standardized half precision float beats the standard 16 bit posit. The issue we want to illustrate with this example is that developers must not underestimate the requirements in respect to accuracy set by a problem. It would be nice if there were an automated way to detect such inaccuracies.

Valids Are Good For Short Computations

Perhaps valids can help, which is why we also solved Problem 3 using valid arithmetic. The resulting values are listed in Table 6.3 and in this particular case, valids do shine. While the low resolution $V_{8,2}$ type results in quite big bounds, the higher resolution types return increasingly small bounds. Indeed, one could argue that in this particular case, valids made complicated error analysis redundant. We can simply keep increasing the number of bits until we arrive at bounds we deem satisfactory. The reason why valids work so well here is that computing roots x_1 and x_2 does not involve a lot of steps. As such the valid bounds do not grow to unacceptable levels, rather they represent a good estimation of the result. We see that for debugging individual steps or short computations, valids can be helpful.

Summary

Type I unums were supposed to be a silver bullet to rounding errors in computer arithmetic. But reality is never that easy. Now at its third iteration, unum arithmetic has become fast and easy to implement. But this comes at the cost of suffering from similar problems as floating points: Problems 1 and 2 were solvable with neither floating point nor posits. Similarly, Problem 3 suffered from rounding errors on low resolutions. But it is not all bad. In particular, for short computations such as Problem 3, valids can be a useful tool. And for what it is worth, posits are competitive, performing about as well as floating points.

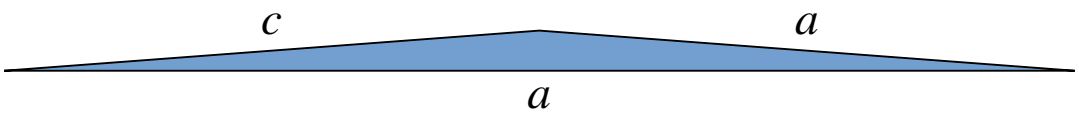


Figure 6.3: A not particularly thin "very thin triangle problem" (Problem 4). Computing the area of such triangles with floating point can be surprisingly difficult.

6.2 Problems From Posit Literature

Naturally posit literature comes with examples of its own. We re-evaluated some of these examples. While we generally were able to reproduce previous experiments, our results can also be looked at in a different light.

Problem 4. *Solve the "Very Thin Triangle Problem" [23, pp. 75] which asks us to find the area of some triangle with sides a, b, c where two of the sides b and c are barely longer than half of the longest side a (Figure 6.3). The problem was originally proposed as an example where floats perform very poorly.*

For our discussion, we first reproduced the thin triangle problem for a triangle with sides

$$a = 7 \text{ and } b = c = 7 + 3 \cdot 2^{-111} \tag{6.11}$$

as described in posit literature [23, pp. 75]. Note that sides b and c are only slightly longer than 7. Representing such values is hard, even with standard 128 bit data types. Running this computation with standard floating point and posit types results in completely wrong and unusable results. However, Gustafson points out that this particular version of the thin triangle problem can be solved with the $P_{128,7}$ posit type [23, pp. 75]. This is true, but it remains questionable whether this solution is a practical one. A 128 bit posit with $ES = 7$ results in a posit type with an useed of

$$\mathcal{U} = 2^{2^7} = 340282366920938463463374607431768211456 \approx 10^{38} \tag{6.12}$$

which no doubt is very large indeed. So large in fact that it is hard to imagine hardware vendors to ever support this particular posit type as a standard feature, leaving us with software arithmetic as the only alternative on general purpose systems.

Parameters Are Power

What Problem 4 shows is that there is power in letting users pick precision and parameters. While hardware will probably continue to be limited to a small number of standard resolution types, augmenting those default types with arbitrary software arithmetic can be useful in concrete applications.

Problem 5. *The "LINPACK Benchmark" can be solved* accurately *with posit arithmetic [24, pp. 85]. Reconstruct this experiment and look at why exactly this is the case.*

LINPACK is a popular measure of floating point performance run on clusters and workstations alike [46]. At its core, LINPACK solves big systems of linear equations, a typical

workload in the field of scientific computation and simulation. We could use a posit port of LINPACK as a performance measure, but this is not the focus here. Rather we will discuss the accuracies involved in this benchmark.

Fused Dot Product

Gustafson et al note that the LINPACK benchmark run with double precision floating points actually returns inaccurate results. Instead of the mathematically correct value 1, the results are values ever so slightly off from 1. This is not the case for posits as even 32 bit posit arithmetic can solve this problem accurately [23, pp. 88]. The reason for this is that a posit port of the LINPACK benchmark can take advantage of the quire dot product to maintain accuracy. The quire is introduced as a fundamental aspect of posit arithmetic, it is natural to use it for this application.

Floating Point Quires

For comparison, we ported a version of the LINPACK benchmark to use naive posit arithmetic ourselves, that is posit arithmetic without the use of quires. Without quires, posit arithmetic suffer from similar problems as floats, returning inaccurate results, even for high resolutions. The quire is advertised as a fundamental part of posit arithmetic and as such must not be overlooked. But the elephant in the room is that quires are not exclusive to posits. Indeed it is easy to imagine a quire type based on floating points that works in very much the same way. In respect to Problem 5, we have to conclude that quires can provide surprisingly accurate results. However we also must not forget that the fused dot product is not exclusive to posit arithmetic.

Summary

In general, we were able to reproduce previous examples from posit literature. But either example can also be looked at in a different light. It is true that the thin triangle problem is solvable with an $ES = 7$ posit type. But it is unreasonable to expect hardware makers to ever support this format as a standard feature. Similarly, LINPACK can be computed accurately with quires, but the idea of quires or an extended fused dot product must not be constrained to posits.

6.3 Decimal Loss of a Unary Function

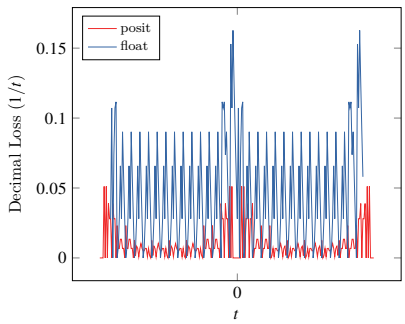
We now compare the accuracy of a unary function in low resolution floating point and posit environments. These examples have previously been used as a selling point for posit arithmetic [24, pp. 78]. While posits do perform better in this benchmark, erratic distribution of error means that neither format provides strong guarantees.

Problem 6. *Compare the accuracy of some basic mathematical function. A good starting point is Section 4 in "Beating Floating Point at its Own Game" [24, pp. 78] which introduces the concept of decimal loss as a metric for accuracy.*

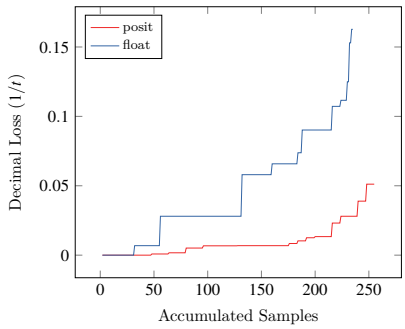
Test Setup

In these examples, we evaluate the so called "decimal loss" introduced by evaluating a unary function. Decimal loss as defined by Gustafson et al is a metric for comparing the accuracy of some computed value y provided by computer arithmetic with an expected value x [24]. Appendix D gives a more detailed refresher on the topic. In respect to the actual tests, we will present our findings for the unary reciprocal function, viz.

$$f(t) = \frac{1}{t}. \tag{6.13}$$



(a) Values on the horizontal sorted, zero is at the center.



(b) Values on the horizontal sorted by decimal loss.

Figure 6.4: Plotting decimal loss of unary function $f(t) = 1/t$ with 8 bit types. Posits results in red, floating point results in blue.

For each value t provided by the given float or posit data type, we compute the decimal loss introduced when computing reciprocal $y = 1/t$. Standard 64 bit `double` arithmetic provides reference x. While previous work is limited to only 8 bit types, we extended the experiment to include standard 16 types as well. We find this is of particular importance as 16 bit floats actually see real world applications [47]. Something not necessarily the case for the more limited 8 bit types used in previous publications.

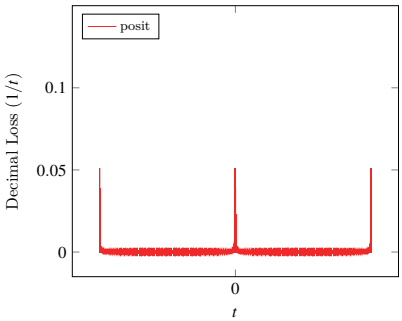
Figure 6.4 plots the result for the 8 bit types while Figure 6.5 illustrates the results when run with 16 bit types. When values t on the horizontal are ordered by the introduced decimal loss, we see a steeper raise for floating points than for posits. This matches previous findings by Gustafson et al [24, p. 79]. In general, posits exhibit less decimal loss for both 8 and 16 bit types when computing the unary reciprocal.

Erratic Loss

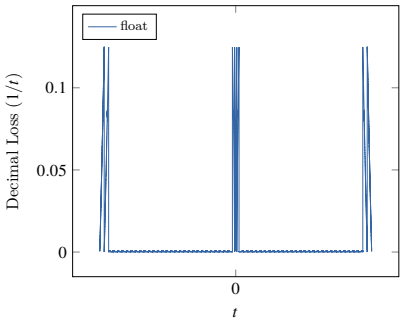
We also plot the results sorted by value t with $t = 0$ at the center. Decimal loss is clustered close to zero and infinity. Both floats and posits show a drastic peak for decimal loss in this area. Despite this, on a whole the resulting plots are quite erratic. We can make little assumptions about the decimal loss of the result. Designing accurate algorithms still requires constant vigilance as decimal loss is distributed in a mostly erratic fashion. In absolute numbers, posits do perform better. Figure 6.5c shows this especially well as it is limited to those 10,000 values with lowest decimal loss. Decimal loss grows less quickly with posits than it does with floats.

Summary

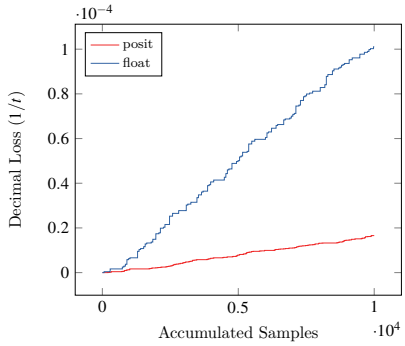
Based on previous work by Gustafson et al, we reproduced and extended experiments that evaluate decimal loss. While it is true that posits perform better than floats, the bigger picture is more complicated. Both floats and posits show erratic error, meaning a programmer using either format can make little assumptions about the level of introduced error.



(a) Values on the horizontal sorted, zero is at the center.



(b) Values on the horizontal sorted, zero is at the center.



(c) Values t on the horizontal sorted by decimal loss. This plot only shows the first 10^4 values, i.e. those values with lowest decimal loss.

Figure 6.5: Plotting decimal loss of unary function $f(t) = 1/t$ with 16 bit types. Posits results in red, floating point results in blue. We see drastic peaks close to zero and infinity.

6.4 Closure and Accuracy

We expect the standard operations $+$, $-$, $\cdot$ and $\div$ to be closed, that is given two arguments x and y of some type, the result of any binary operation $x \circ y$ should again be of the same type. On $\mathbb{R}$, this holds for most operations.

We could cheat here and argue that the result of, for example, any `float` division is again a value of type `float`. But this is hardly productive. What we are interested in are non-special values that allow us to continue computation. In this way, perfect closure is achievable with neither floats nor posits as both formats contain special values such as *NaR* or *NaN*. What we can compare is how often special values pop up as a result of standard math operations.

6.4.1 Closure

We decided on an empiric approach where for each binary operation $\circ$, we evaluate

$$x \circ y \tag{6.14}$$

with our `aarith`-based implementation and then plot the result. While this approach is relatively straight-forward to implement, it is limited to small sizes. Trying out every possible combination of arguments x and y for a 32 type would result in a plot with

$$2^{32} \cdot 2^{32} = 2^{64} \tag{6.15}$$

pixels, an amount no memory currently available to us can store. As such we limited ourselves to 8 and 16 bit types. Previous work published by Gustafson is limited to only 8 bit posit types [23, pp. 63].

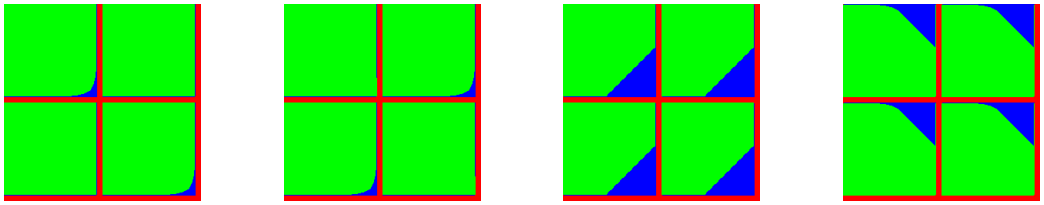
Floating Point Closure

Figure 6.6 plots closure for 8 and 16 bit floats. (1) Red pixels indicate that a given result is *NaN*, (2) blue pixels indicate either positive or negative infinity and (3) green pixels represent standard values in $\mathbb{R}$. For the higher resolution 16 bit type, special values are generally less pronounced. But no matter the resolution, a good portion of all available bit patterns is spent on special values. Unum literature would perhaps call these cases "wasted" [23, p. 44].

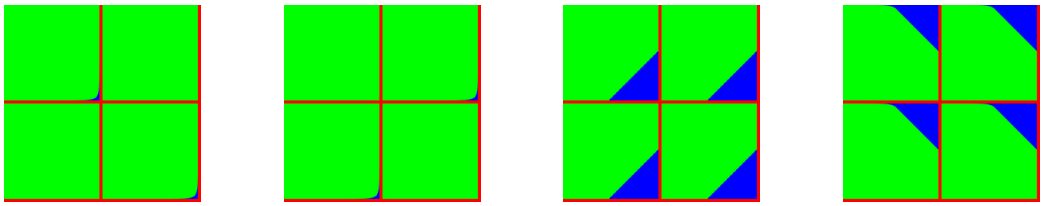
Posit Closure

The equivalent posit plots in Figure 6.7 give a different picture. Almost all pixels are green, we only get a slight red border in the very few cases where the result of a given binary operation is *NaR*. This is the case only when any of the operands themselves are *NaR* or when division by zero is attempted. Plots for the 8 and 16 bit types look exactly the same. No matter the size, the behavior is identical. We might count this as a plus for the posit format. No matter the parameters, behavior is identical as far as closure is concerned.

Just from looking the these two sets of plots, we might get the idea that posits are closed in more cases and as such are a better format. Of course this is an oversimplification. What is actually happening is that posits cannot alert users of values too big or small to represent. While posits indicate failure with ∞, posits can do no such thing.

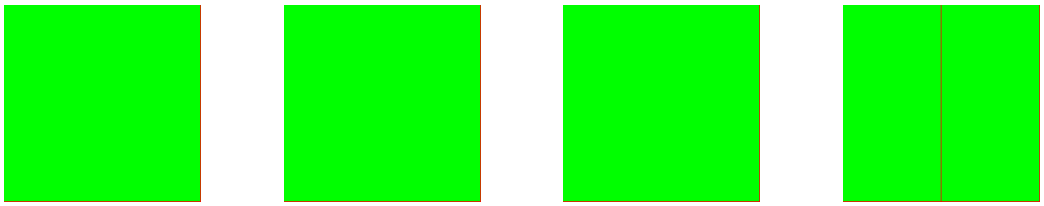


(a) $x+y$, 8 bits (b) $x-y$, 8 bits (c) $x \cdot y$, 8 bits (d) $x \div y$, 8 bits

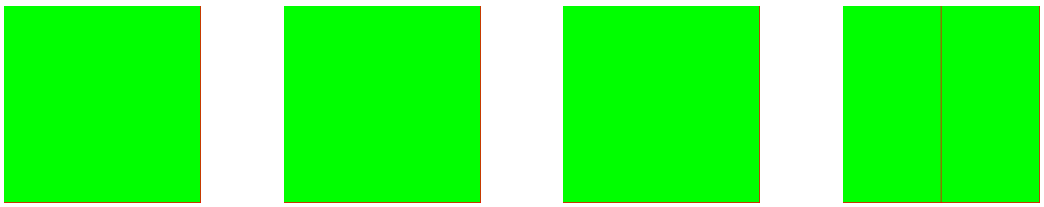


(e) $x+y$, 16 bits (f) $x-y$, 16 bits (g) $x \cdot y$, 16 bits (h) $x \div y$, 16 bits

Figure 6.6: Closure of binary operations $x \circ y$ where x and y are standard floating point types. Green values represent a normal result, red indicates *NaN* and blue infinity.

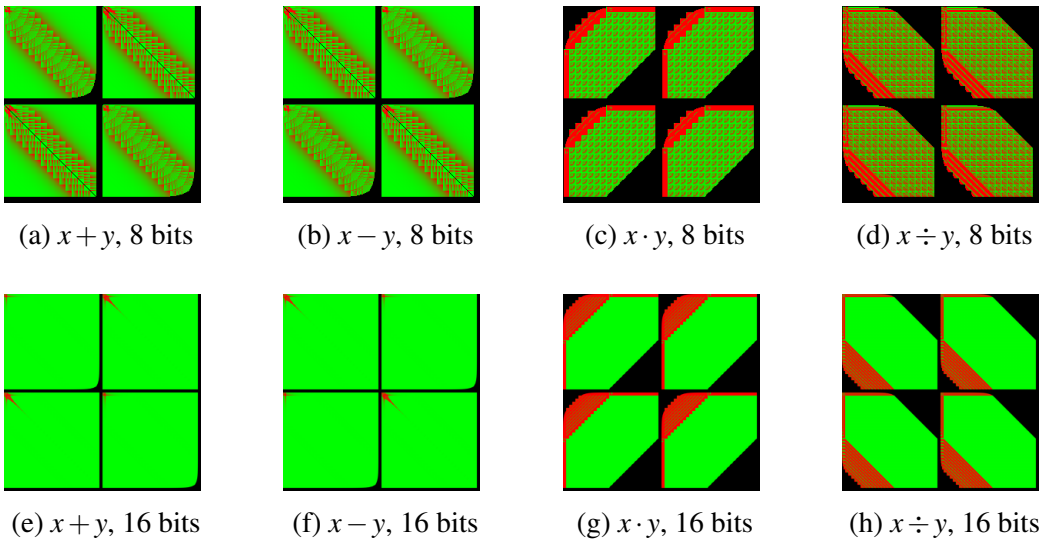


(a) $p+q$, 8 bits (b) $p-q$, 8 bits (c) $p \cdot q$, 8 bits (d) $p \div q$, 8 bits



(e) $p+q$, 16 bits (f) $p-q$, 16 bits (g) $p \cdot q$, 16 bits (h) $p \div q$, 16 bits

Figure 6.7: Closure of binary operations $p \circ q$ where p and q are standard posit types. Green values represent a normal result, red indicates *NaR*.



(a) $x+y$, 8 bits (b) $x-y$, 8 bits (c) $x \cdot y$, 8 bits (d) $x \div y$, 8 bits

(e) $x+y$, 16 bits (f) $x-y$, 16 bits (g) $x \cdot y$, 16 bits (h) $x \div y$, 16 bits

Figure 6.8: Accuracy of binary operations $x \circ y$ where x and y are standard floating point types. Shades of green represent relatively accurate results, shades of red are less accurate. Values in black are either *NaN* or infinity; in these cases computing decimal loss will not yield useful results.

6.4.2 Accuracy

Even perfect closure is not particularly useful should the computed result be wrong. For this reason, we reproduced and improved upon previous work [23, pp. 63] which plots the decimal loss of all four arithmetic operations. Unlike previous work, we plot the results not just for 8 but also for 16 bit types. The resulting plots are pretty, but drawing definite conclusions from them is anything but easy.

Float Accuracy

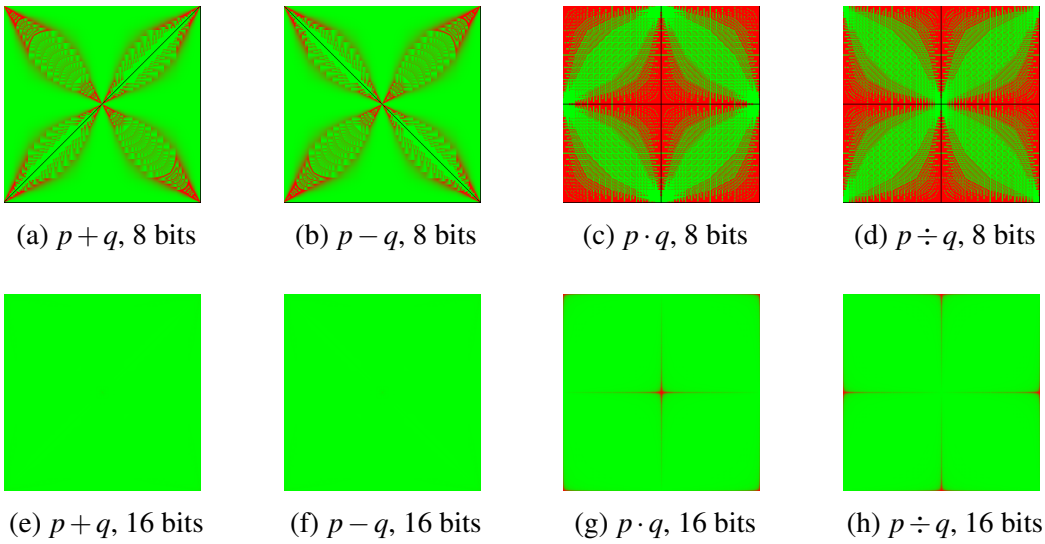
Figure 6.8 plots the decimal loss of all four arithmetic operations for both 8 and 16 bit floating point types. As computing decimal loss for special values *NaN* or infinity does not exactly tell us much, the respective pixels were left black. If we look at just the non-special values shaded on a spectrum between green (relatively accurate) and red (relatively high error), we see that (1) the higher resolution types are more accurate and (2) while there are some patterns, distribution of decimal loss is mostly erratic.

Posit Accuracy

Moving to posit arithmetic in Figure 6.9, we can make similar observations. Most notably, the 16 bit type is more accurate than the low resolution 8 bit type. So much more accurate in fact that the plot is almost entirely green. All plots for floats and posits of either 8 or 16 bits in size share the same scaling. As the 16 bit posit plots are mostly green, we know that they show the least amount of decimal loss.

Summary

While the resulting accuracy plots are no doubt pretty, it is difficult to draw real world conclusions from them. The first impression must be that both floats and posits show erratic behavior. A second observation is that posits do have cases where they are deeper in the read than floats. As posits do not return infinity for a finite result, but this comes at the cost of great inaccuracies.



(a) $p+q$, 8 bits (b) $p-q$, 8 bits (c) $p \cdot q$, 8 bits (d) $p \div q$, 8 bits

(e) $p+q$, 16 bits (f) $p-q$, 16 bits (g) $p \cdot q$, 16 bits (h) $p \div q$, 16 bits

Figure 6.9: Accuracy of binary operations $p \circ q$ where p and q are standard posit types. Shades of green represent relatively accurate results, shades of red are less accurate. Values in black are *NaR*; in these cases computing decimal loss will not yield useful results.

6.5 Commutativity, Associativity and Distributivity

We have certain expectations about the standard arithmetic operators $+$, $-$, $\cdot$ and $\div$ known from $\mathbb{N}$ or $\mathbb{R}$. In particular, we may expect associative, commutative and distributive properties. With computer arithmetic, many of these properties do not hold. The question for our evaluation has to be whether Type III arithmetic is any better than traditional floating points.

Properties On $\mathbb{R}$

Most readers will be familiar with the properties mentioned above; nevertheless Appendix E provides a quick review of their definitions. On $\mathbb{R}$, addition $a+b$ is commutative and associative just as multiplication $a \cdot b$ is commutative and associative. Combining both operations, multiplication distributes over addition in the sense that

$$a \cdot (b+c) = ab+ac \tag{6.16}$$

and so on. In the interest of time, our discussion omits subtraction and division.

6.5.1 Properties of Floating Point and Posit Arithmetic

As both IEEE floating point numbers as well as posits represent values in $\mathbb{R}$, one would expect their operators to obey the same laws as in $\mathbb{R}$. But there is no free lunch. While commutativity holds, associativity and distributivity does not.

Commutative Property.

Both floats and posits offer addition and multiplication operations that are commutative. In either case, to compute the sum or product of operands a and b, we first split a and b into scale k and fraction m, that is

$$a = m_a \cdot 2^{k_a} \quad \text{and} \quad b = m_b \cdot 2^{k_b}, \tag{6.17}$$

do the math with those parameterized forms,

$$a + b = m_a \cdot 2^{k_a} + m_b \cdot 2^{k_b} \tag{6.18}$$

and then convert back to floating point or posit representation. Because all parameterized operations are integer operations and integers addition and multiplication is commutative [48], so is floating point and posit addition and multiplication. Note that this only shows commutativity. It does not mean that the results will be correct or particularly accurate. Commutativity only implies that no matter the order of operations, the result will be identical.

Associative Property

The story is a different one when it comes to associativity. Neither floating point nor posit arithmetic can ensure this property [49, p. 30]. Consider the case of

$$(x + y) + z \tag{6.19}$$

where x, y and z are floats or posits. Above sum is not guaranteed to be identical to

$$x + (y + z) \tag{6.20}$$

because in either case the sum inside the parenthesis are evaluated first. Rounding takes place at different steps and as such the results are not guaranteed to be the same. Neither floats nor posits ensure the associative property for addition or multiplication.

Distributive Property

Addition and multiplication on floats or posits are not distributive either. [50, pp. 6]. Splitting up some term

$$a \cdot (b + c) = ab + ac \tag{6.21}$$

means that instead of evaluating $b + c$ first, products ab and ac are evaluated and rounded first. The result cannot be guaranteed to be identical.

Neither floats nor posits maintain the associative and distributive properties of addition or multiplication. Programmers will always have to keep these constraints in mind when implementing numeric algorithms based on floating point or posit arithmetic. In this

regard, posits are no better than floats.

6.5.2 Properties of Valid Arithmetic

While posits are no better than floating points, what about the new valid format? As a supposedly mathematically rigorous system, can valids do any better than traditional floating point arithmetic? Unfortunately, we find that just like floats and posits, valids do not maintain associativity and distributivity for addition and multiplication.

Commutative Property.

Valid addition and multiplication is commutative because the underlying posit addition and multiplication is as well. Valid addition amounts to evaluating interval arithmetic rule

$$v+w=(a,b)+(c,d)=(a+c,b+d) \tag{6.22}$$

which is the same as

$$w+v=(c,d)+(a,b)=(c+a,d+b) \tag{6.23}$$

and as such valid addition is commutative. The same applies for multiplication. In valid multiplication, we have to compute four candidates

$$v\cdot w=(a,b)\cdot(c,d)\rightarrow\{ac,ad,bc,bd\} \tag{6.24}$$

which are the same four candidates even if we flip v and w around, viz.

$$w\cdot v=(c,d)\cdot(a,b)\rightarrow\{ca,cb,da,db\}. \tag{6.25}$$

We see that valid multiplication is commutative.

Associative Property

Valid addition and multiplication are not associative. One can find many examples for why this is the case. Here we present two simple examples in a $V_{4,1}$ environment (Appendix B). We begin with addition. If associativity were to hold, we would get

$$a+(b+c)=(a+b)+c. \tag{6.26}$$

However, if we pick

$$a=4,\quad b=1,\quad c=1 \tag{6.27}$$

we see that the results differ as

$$a+(b+c) = 4+(1+1) = 4+2 = (4,16) \tag{6.28}$$
$$(a+b)+c = (4+1)+1 = (4,16)+1 = (4,\infty). \tag{6.29}$$

The order of evaluation changes the result. Of course in either case the perfectly accurate result $4+1+1=6$ is contained in the returned interval. But the returned intervals are not identical and as such valid addition is not associative. In a similar fashion, if associativity were to hold for valid multiplication, we would get

$$a \cdot (b \cdot c) = (a \cdot b) \cdot c \tag{6.30}$$

but again this is not the case. Picking

$$a = 0.25, \quad b = -2, \quad c = (2,16) \tag{6.31}$$

results in different outcomes depending on the order of evaluation. We get

$$a \cdot (b \cdot c) = 0.25 \cdot (-2 \cdot (2,16)) = 0.25 \cdot (-\infty,-4) = (-\infty,1) \tag{6.32}$$
$$(a \cdot b) \cdot c = (0.25 \cdot (-2)) \cdot (2,16) = -0.5 \cdot (2,16) = (-16,-1). \tag{6.33}$$

Note that in either example, one result contains infinity while the other does not. Depending on the order of evaluation, we may get small bounds (Equations 6.28, 6.33) or huge bounds that contain infinity (Equations 6.29, 6.32).

Distributive Property

On $\mathbb{R}$, we can distribute multiplication over addition, that is

$$a \cdot (b+c) = ab + ac. \tag{6.34}$$

However for valids, if we pick

$$a = 0.25, \quad b = -2, \quad c = 16 \tag{6.35}$$

in an $V_{4,1}$ environment, we get different result for the left and right terms, viz.

$$a \cdot (b+c) = 0.25 \cdot (-2+16) = 0.25 \cdot (4,16) = (1,4) \tag{6.36}$$
$$a \cdot b + a \cdot c = 0.25 \cdot (-2) + 0.25 \cdot 16 = -0.5 + 4 = (2,4). \tag{6.37}$$

Just like with associativity, distributivity does not hold on valids.

While valids do not perform worse than traditional floating point arithmetic, they hardly are more rigorous as far as associativity and distributivity is concerned. Perhaps worse, the size of returned intervals can vary greatly depending on the concrete order of evaluation.

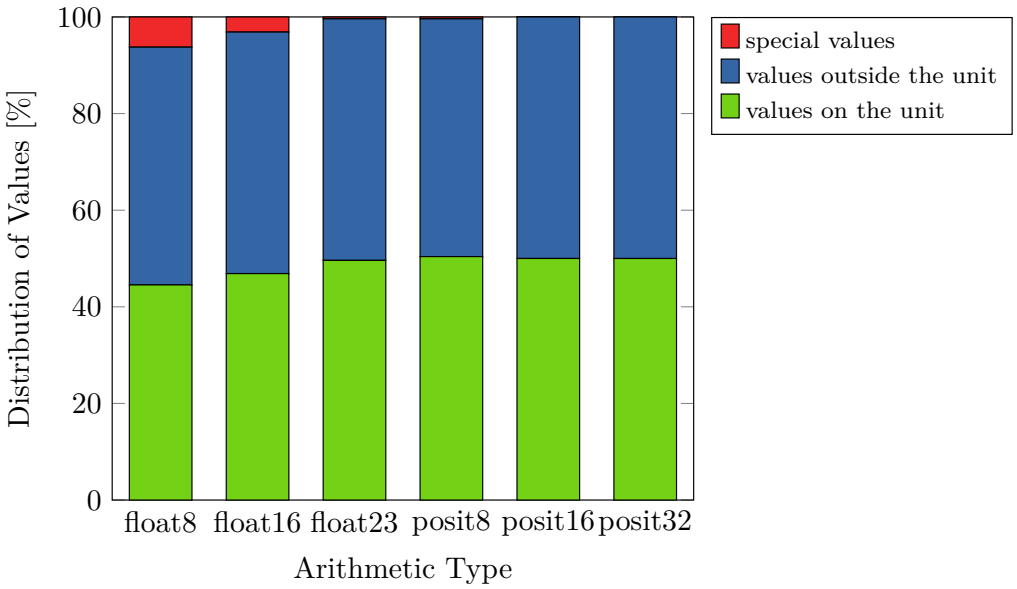


Figure 6.10: Distribution of values on the unit $[-1,1]$, outside the unit or special (i.e. *NaR*, *NaN* or infinity). For higher resolutions, floating points and posits are distributed in about the same way.

6.6 Exploiting the Unit Interval?

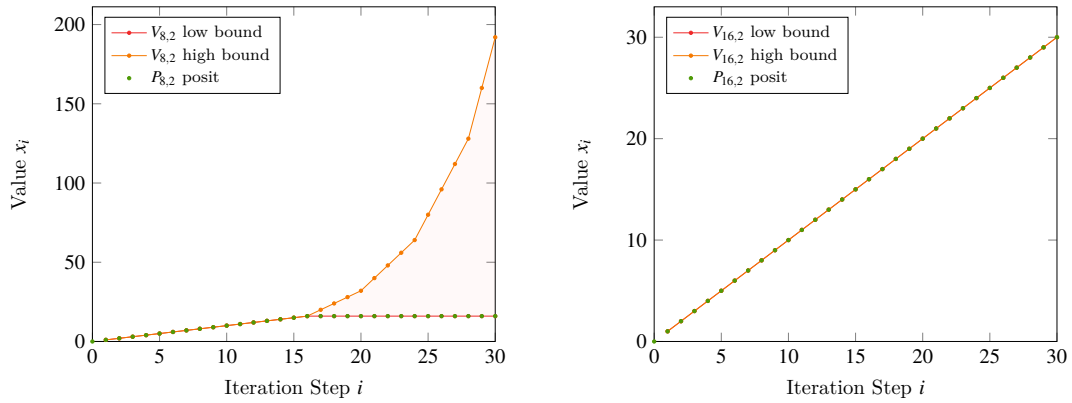
Posits should be especially accurate on the unit interval $[-1,1]$. When applications do lots of computations close to zero, posits should perform well. If on the other hand we do math with huge unscaled numbers, perhaps floating points can be a better choice. We want to investigate this hypothesis with a real world experiment.

Problem 7. *Decision Tree Learning is a simple but effective way of machine learning [51, pp. 697]. As all involved math is run on probabilities $p \in [0,1]$, we expect posits to be a natural fit. Implement decision tree learning and compare the performance of floats with that of posits.*

To experiment with decision tree learning, we ported an existing open source implementation, `bowbowbow/DecisionTree` [52], to support arbitrary `aarith` types. Unfortunately, the results acquired from our experiments are not very decisive. Even 8 bit posit and floating point types provide the same exact result as 64 bit types in all standard benchmarks and examples. For what it is worth, it is interesting to see that low resolution types perform so well in this task. But we are interested in comparing floats with posits. Both floats and posits perform the same, neither format is really superior in this case.

Equal Distributions

Reality strikes once we analyze the distribution of values for both floating points and posits. Figure 6.10 shows the share of (1) real values on the unit, (2) real values outside the unit and (3) special values, that is *NaR*, *NaN* and infinity. It is true that posits spend about half of all available values on the unit. But for floats the story is pretty much the same. For both the 16 and 32 bit floating point types used in real world systems, about



(a) Iteration using the $V_{8,2}$ valid type results in a huge bound.

(b) Iteration using the $V_{16,2}$ valid type behaves well in the first 30 steps.

Figure 6.11: Starting at $x_0 = 0$, we iterate by repeatedly adding 1 to x_i. Even this simple example causes problems as posit arithmetic eventually gets stuck.

half of all available values end up on the unit. This feature of posits really is not unique to the format. Tried and tested floating points are distributed in about the same way, especially when it comes to the higher resolution types actually in use today.

6.7 Some Experiments on Valids

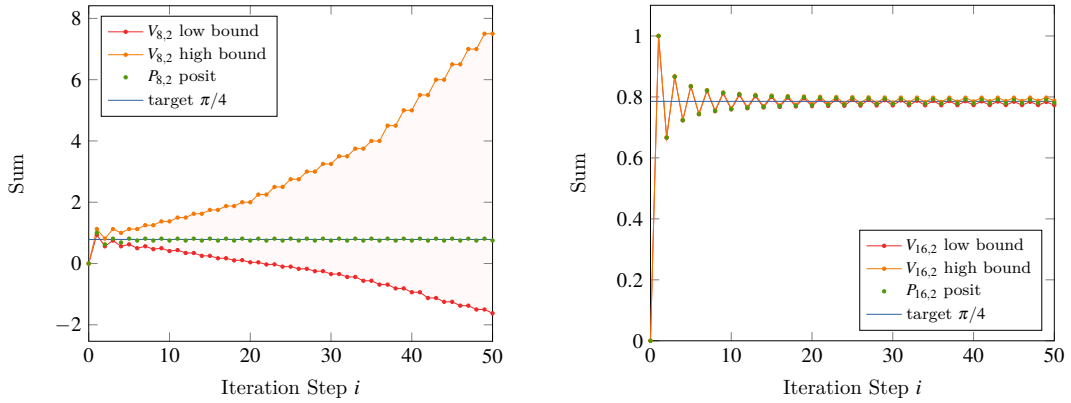
Our evaluation so far was focused on posit arithmetic. While be believe posits to be the center piece of Type III arithmetic, what about the novel valids? As our port of the `aarith` library contains rudimentary support for the format, we at the very least want to preset the following three examples dedicated to valid arithmetic.

Problem 8. *Continuously increment a counter by one. Even this simple example can cause problems in float or posit arithmetic. Check whether valids are any better.*

Starting at $x_0 = 0$ and repeatedly incrementing $x_{i+1} = x_i + 1$ can cause serious error when x_i is a floating point or posit type. Remember that either format rounds results to the nearest representable point on the number line. At some point, both floats and posits eventually get stuck. This happens when incrementing $x_i + 1$ is rounded down to just x_i, that is when

$$x_i + 1 = x_i. \tag{6.38}$$

In this experiment, we continuously incremented both 8 and 16 bit valids by one, the results are plotted in Figure 6.11. The 8 bit valid diverges to a huge interval starting right after $i = 16$. Meanwhile the 16 bit valid returns accurate results in the first 30 steps plotted in Figure 6.11. The reason for this curious behavior is that incrementing 8 bit



(a) Iteration in a low resolution 8 bit posit environment.

(b) Iteration in a higher resolution 16 bit posit environment.

Figure 6.12: Computing a series that should approach $\pi/4$. While 16 bit valids return a good bound on the expected result, the low resolution 8 bit valid diverges.

posits gets stuck at value 16. In $P_{8,2}$ arithmetic, we get

$$16 + 1 = 16 \tag{6.39}$$

as the next discrete step on the number circle after 16 is 20. For the higher resolution 16 bit type, this happens much later at 1024. While not pictured in our figures, the respective 16 bit valid type also starts diverging right after $i = 1024$, exactly when the associated posit type gets stuck itself.

This is a big win for valids. Valids as a mathematical rigorous debugging environment warn the user that the result of the requested operation does not fit in the underlying posit type. While valids do not automatically provide an accurate solution, they do not hide rounding error either.

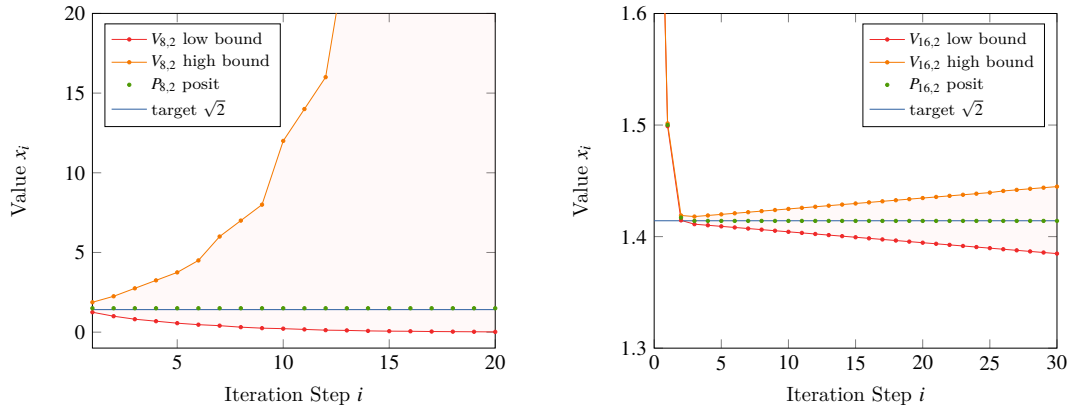
Problem 9. *Evaluate an infinite sum with both posits and valids. See how the result compares. Perhaps valids can be used to determine when iteration may halt.*

In this example, we look at how valids perform when summing up infinite series. One question in particular we are interested in whether valids can help us in finding the right number of summands to evaluate before halting computation. In concrete terms, we used series

$$\sum_{k=0}^{\infty} (-1)^k \frac{1}{2k+1} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} \pm \cdots = \frac{\pi}{4} \quad \text{[42, p. 77]} \tag{6.40}$$

to approximate $\pi/4$. The results for both 8 and 16 bit types are plotted in Figure 6.12.

In the low resolution 8 bit environment, posits very quickly reach the correct solution, jumping between a value just above and one just below the expected result $pi/4$. Unfortunately this is not mirrored by the respective valid type. While posits quickly arrive at the



(a) Iteration using the $V_{8,2}$ valid type

(b) Iteration using the $V_{16,2}$ valid type.

Figure 6.13: Computing $\sqrt{2}$ using the Newton-Raphson method [40]. While the returned posit results are good, 8 and 16 bit valids actually result in unacceptably big bounds.

right results, the valid bound only grows with each step. We could interpret this as cause for alarm when in fact the computation was successful. In the 16 bit environment, posits also quickly converge at a correct result. But unlike before, valids follow suit, closely capturing the result as part of their bound.

As far as Problem 9 is concerned, valids return mixed signals. While in the low resolution 8 bit environment valids raise alarm, they happily converge in the higher resolution 16 bit environment. Judging from this limited experiment, valids cannot serve as a guide for when to halt summation.

Problem 10. *Compute the square root of two using the Newton-Raphson Method. See how valids react when the result starts being correct.*

This is similar to Problem 9, but instead of evaluating a series we run an iterative algorithm. In particular, the Newton-Raphson method computes the square root of x by iterating

$$x_{i+1} = \frac{1}{2}\frac{x_i + x}{2} \tag{6.41}$$

starting at $x_0 = 2$ [40]. We actually use this algorithm in our implementation of `sqrt` and as previous examples show that it serves us well (Section 5.2).

Figure 6.13 plots our results. As we are computing $\sqrt{2} \approx 1.414213$ and start at $x_0 = 2$, posits very quickly arrive at a result that is quite good. This is true even for the low resolution 8 bit environment. Unfortunately, in this case valids quickly diverge in both 8 and 16 bit environments. Again users are alerted of potential problems when in fact posits arrived at a perfectly useful result.

Summary

It is hard to draw definite conclusion from these three examples. While Problem 8 is promising, valids fail to provide good answers for Problems 9 and 10. Certainly we did

learn that naively adapting standard algorithms as in use with floats or posits will not necessarily yield productive result when run with valids. Admittedly we ran way less valid examples than we would have liked. It is up to future research to figure out how valids react in different applications.

6.8 Full Applications

Posits are meant to be drop replacements for floating points, but posit arithmetic certainly is not a perfect one-to-one match. While floats differentiate between ∞, $-\infty$ and *NaN*, posits only have *NaR* as a special value. In addition, posits never round reals to *NaR* or zero, rather they round to *maxpos* or *minpos* instead. To gather some related hands-on experience, it makes sense to port real-world applications to use posits instead of floats.

6.8.1 Porting a Raytracer

Problem 11. *Port the `nsilvestri/cpp-raytracer` [53] project to use posits instead of floats. While there are many ray tracers out there, this one in particular is delightfully short.*

`cpp-raytracer` is a rudimentary implementation of ray tracing [54, pp. 1-31] written in C++, clocking in at about 11,000 lines of code [53]. All arithmetic is done with the standard `double` floating point type on the CPU. We ported `cpp-raytracer` to use arbitrary floating point and posit types provided by our version of the `aarith` library.

Two Step Porting

Directly porting even a small project such as `cpp-raytracer` to posits can be difficult as it involves many small changes. If even one change is incorrect, the entire result might turn out wrong. This certainly was the case for us; a simple find and replace of all occurrences of the float type with a respective posit type yielded nothing but a black screen. What proved less error prone was to first port the project to use `aarith` floating point types. Comparing different outputs at different steps allowed us to find careless copy-paste errors. Once the port to `aarith` floating points was complete, switching to `aarith::posit` was trivial.

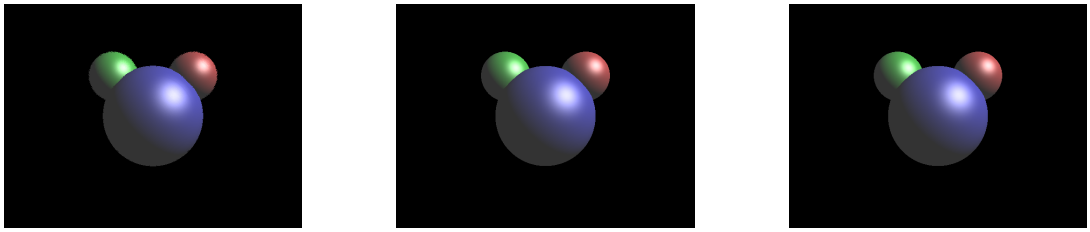
Easy Port

We find that that this particular implementation of ray tracing does not take advantage of special values such as *NaN* or infinity. As such, once porting to the posit type was complete, our implementation *just worked*, returning matching results to the naked eye.

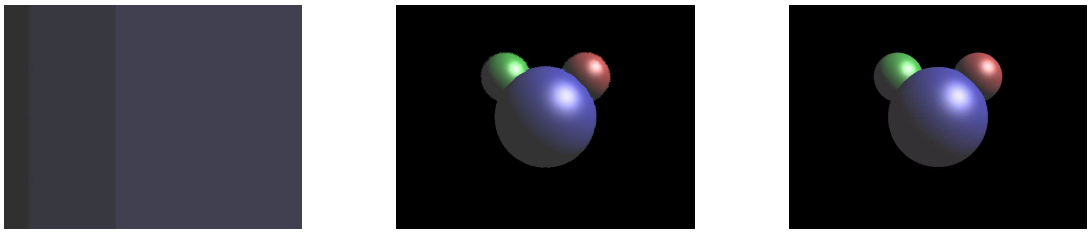
16 Bit Posit Equal To 32 Bit Float

With our port complete and functional, it was naturally tempting to try out types of different resolution and evaluate how they compare in this particular use-case. Figure 6.14 presents the output for different types. When we have at least 32 bits of precision, both standard floats and posits perform just fine. The original version of the project uses `double` for everything, but in our experiments `float` proved sufficient to produce satisfactory results. Curious things start happening once we scale down to only 16 bits of precision. While the standard 16 bit posit type performs well aside from slight aliasing, 16 bit floats fail to provide a proper result.

16 Bit Storage



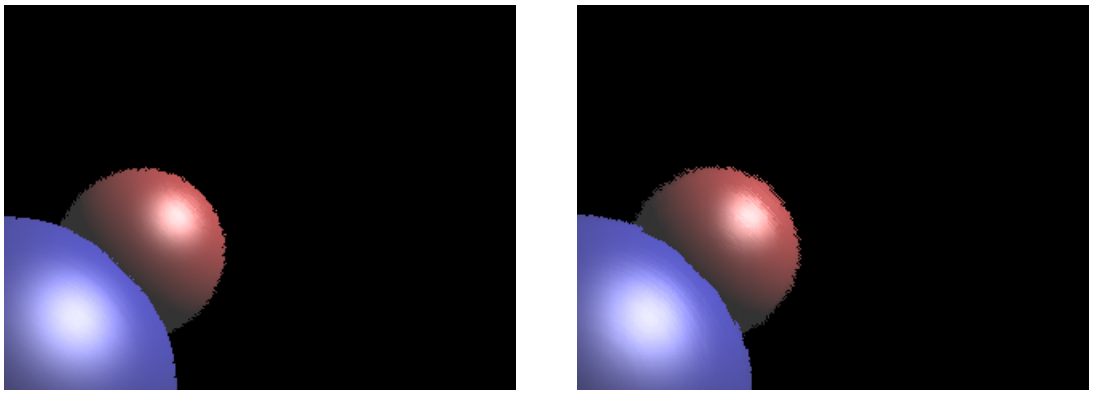
(a) Standard `__fp16` type. (b) Standard `float` type. (c) Standard `double` type.



(d) 8 bit `aarith` posit. (e) 16 bit `aarith` posit. (f) 32 bit `aarith` posit.

(g) 8 bit `aarith` float. (h) 16 bit `aarith` float. (i) 32 bit `aarith` float.

Figure 6.14: Running our port of `cpp-raytracer` with different floating point and posit types.



(a) Standard `__fp16` type. (b) 16 bit `aarith` posit.

Figure 6.15: Running our port of `cpp-raytracer` with 16 bit types. Using 16 bit types introduces aliasing not present in higher resolutions.

```
float max_of(vector<float> &elements)
{
        float maxval = -INFINITY;

        for (float elem : elements)
        {
                if (elem > maxval)
                {
                    maxval = elem;
                }
        }

        return maxval;
}
```

Listing 2: Computing the maximum value of a vector of floating point values. Function `max_of` uses the special `INFINITY` value not available in posit arithmetic.

For comparison, we also ran `cpp-raytracer` with the `__fp16` data type which represents an IEEE half precision float, available on 64 bit ARM platforms [55]. Rendering the scene with this `__fp16` type results in a mostly satisfactory result, exhibiting similar aliasing as when run with 16 bit posits. Figure 6.15 shows a close-up of the aliasing introduced by the respective 16 bit types. Why the difference between `__fp16` and `aarith` floating point types? While ARM platforms and compilers do support 16 bit IEEE floating points, the 16 bit format is used only for storage. When arithmetic is done on `__fp16` values, they are first converted to a standard 32 bit `float`. Arithmetic is done in the higher resolution, the result converted back to `__fp16` [55]. This allows for compact storage while arithmetic is still reasonably accurate.

Summary

In summary, porting `cpp-raytracer` to use posits was easy, especially as we were able to first port the project to use `aarith` floats in an intermediate step. In this particular case, posits are a genuine drop-in replacement for floating point arithmetic. Additional experiments with the `__fp16` type showed that using low resolution types for storage and high resolution types for arithmetic can yield very productive results, a topic worth further investigation for both floats and posits.

6.8.2 Porting a Genetic Algorithms Library

Problem 12. *`Arash-codedev/openGA` [56] provides configurable implementations of genetic algorithms. It might be interesting to port it to use posits instead of floating points.*

`openGA` is a framework that can be used to solve optimization problems with genetic algorithms written in C++ [56]. In its original version, all computations use the standard `double` type. For our experiments, we ported the project to support `aarith` floating point and posit types as well. Our experiments show that both types perform about the same,

Type	Generations	Cost	Type	Generations	Cost
`float64`	28.6	2.9	`posit64`	29.3	3.1
`float32`	29.1	2.9	`posit32`	29.4	2.9
`float16`	140.1	2.5	`float16`	25.8	1.3 (wrong!)

Table 6.4: Running the `so-1` example from the `Arash-codev/openGA` package with various `aarith` data types. Each experiment was run a total of 1,000 times, values listed in this table are averages. Less is better.

though at low resolutions, floats take longer while posits return incorrect results.

Difficult Port

While our port of `cpp-raytracer` did not require sophisticated modifications, this was not the case with `openGA` as the project uses floating point value ∞ in its logic. In particular, code in the vain of Listing 2 had to be modified. As the code base of this particular project comprises of less than 4,000 lines, it was not particularly difficult to find all relevant cases and replace them with appropriate workarounds. In bigger projects however, we are looking at a more delicate task. Calling posit arithmetic a drop-in replacement is problematic because porting applications that use floats to posit arithmetic can introduce tricky bugs.

Generations

Genetic algorithms often work in generations. In each generation, randomly chosen genotypes are combined in a way that hopefully improves the overall result [57]. `openGA` contains various examples, including example `so-1` which runs a single objective optimization problem until the result is reasonably optimized. Naturally it is desirable for this to occur as quickly as possible, that is we want a good result in a small number of generations.

Deceiving Results

With our port in place, we compared the performance of floating point and posit types when running the `so-1` example, Table 6.4 lists our results. Standard 32 and 64 bit types perform about the same, arriving at a reasonable result in about the same number of steps. When we reduce the bit width to 16, things get interesting. Standard half precision floating point requires way more generations, but eventually arrives at a valid result. The 16 bit posit type on the other hand is problematic because the results are deceiving. On first glance, `posit16` requires the least number of steps while returning the best result. The problem is that result are inaccurate. The resulting genotypes, if computed with higher resolution, result in a cost much higher than when computed with `posit16`.

Summary

Porting `openGA` to use our version of the `aarith` library yields two interesting conclusions. First of all, posits really cannot be called a true drop-in-replacement for floats. Posits have no way of representing infinity and as such any logic that relies on it will require extra tweaking. We worry that porting big projects could be quite the endeavor. Second, we saw that in this particular application, floats and posits yet again perform about equally well. The one exception are are the low resolution 16 bit types where floats are slow and posits are wrong, meaning neither type is particularly advantageous compared to the higher resolution alternatives.

6.8.3 Porting a Neural Network Application

Problem 13. *`Rekpet/TypeCNN` "is a convolutional neural network library that provides reasonable amount of functionality and reasonable speed on CPU" [58]. As all arithmetic is done on the CPU, it can be adapated to support different arithmetic types.*

Easy Port

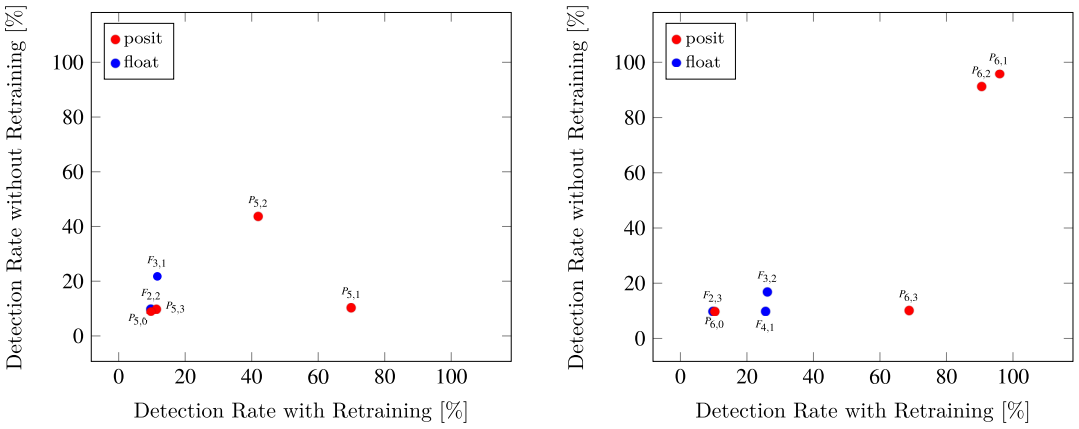
Previous work on the `aarith` library already added support for `aarith` floating types to the `TypeCNN` project [31]. It comes natural to extend the existing fork with support for posit arithmetic. As `TypeCNNN` does not use any float-specific logic like we encountered in the `openGA` project, porting `TypeCNN` to posits was as simple as replacing the respective type alias with an `aarith::posit` type.

Posits Outperform Floats

To compare the accuracy of different floating point and posit types in this application, we ran a benchmark that trains a neural network to recognize handwritten letters [59]. Previous work limited to floating points shows that low resolution floating point types just 10 bits in length perform well in this particular test [31]. However our tests show that posits are even better, the results are plotted in Figure 6.16. With posit arithmetic, we start to see reasonable results for sizes as small as 6 bits where floats need at least 8 bits to come close to an acceptable detection rate. Here we have an example were indeed posits beat floating point at their own game.

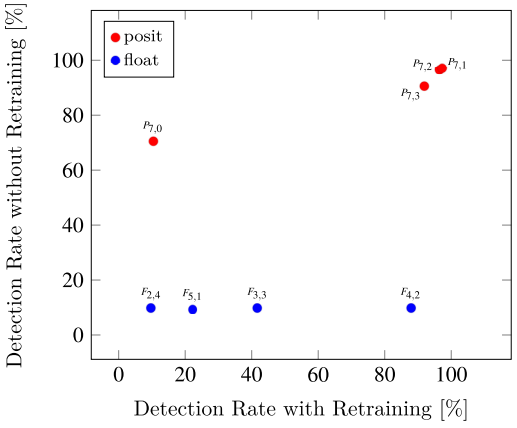
6.9 Summary

This concludes our evaluation of Type III unum arithmetic. In our first set of experiments, we saw that posits are more like floats than they are like the original unums. Though at the very least, valids can warn developers of something being amiss. As far as posit experiments and benchmarks are concerned, we generally were able to reproduce previous results. Where possible we extended previous tests to include more realistic parameters for the given unum types. Disappointingly, we also saw that posits and valids do not provide associativity and distributivity. Similarly, experiments with valids proved indecisive, returning mixed signals depending on resolution and application. In our final set of experiments we ported three real world applications to use Type III arithmetic. In general, porting was easy, though we remain worried about float-specific code that might turn out to be hard to track down and fix.

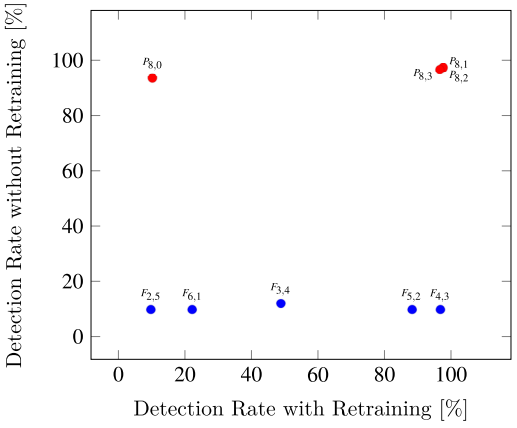


(a) $N = 5$.

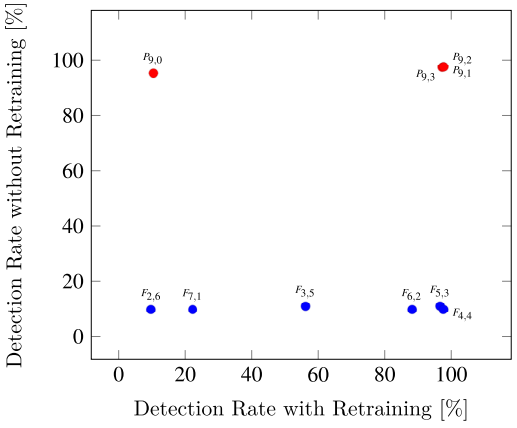
(b) $N = 6$.



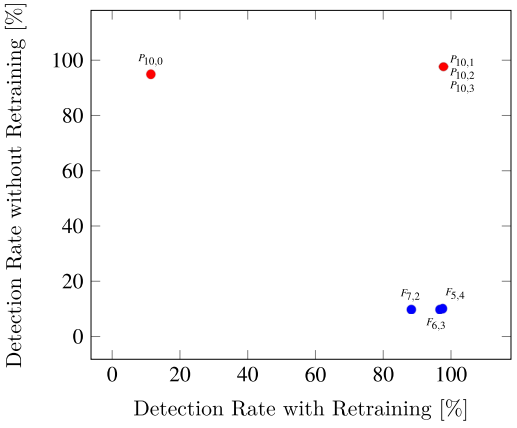
(c) $N = 7$.



(d) $N = 8$.



(e) $N = 9$.



(f) $N = 10$.

Figure 6.16: Detection rates when using a neural network to recognize handwritten letters. While posits perform well with as little as $N = 6$ bits, floats require at least $N = 8$ bits in size to achieve good detection rates.

7 Conclusion

Are posits and valids a replacement for IEEE 754 floating point arithmetic? We started this thesis with a grim look at problems introduced by computer arithmetic. Computer arithmetic is limited by a fixed number of bits and contains confusion duplicate patterns. Jagged accuracy results in curious results, even repeatedly incrementing a number does not necessary behave as expected. All of which are problems we have learned to live with, so perhaps it is true that floating points are in dire need of replacement?

Unum arithmetic promises to be an alternative to floating points that does not suffer from any of such problems. In particular, it wants to provide an automated solution to complicated numerical analysis. Type I and Type II unum arithmetic boast novel ideas like the u-bit but ultimately proved unreasonably hard to implement. Type III arithmetic is meant to solve these problems as it takes real hardware into account.

Type III arithmetic, split into posits, quires and valids can provide the programmer with the right tool for the right job. Posits are meant for quick and efficient computations while valids are more of a debugging tool used during development. Our evaluation of the posit format certainly showed that posit arithmetic can hold its water compared to traditional floating points. But in our experiments, posits never really were notably better than floating points. Posits usually are just as good as floats.

As for valids, we are not yet ready for a definite conclusion. While occasionally useful to indicate the presence of error, as a whole it appears that valids suffer from similar problems as traditional interval arithmetic. This should not be surprising as our definition of valids is based on the rules of traditional interval arithmetic. In particular, we showed that valids do not maintain associativity and distributivity on addition and multiplication. With this in mind, we really cannot call valids mathematically rigorous.

Future Work

Perhaps any good thesis asks more questions than it answers. Certainly there is lots of potential for future work that can build on our contributions. First of all, our definition of valids is of need of improvement. In particular, irregular valids are a topic that escaped us. Valids themselves also require more experimentation as we only presented some first steps. We believe that our `aarith`-based implementation can be a good breeding ground for developments in this direction. Aside from these big points, we also see smaller topics of interests. As we showed, finding accurate mathematical algorithms for functions such as `sqrt`, `log` and `sin` is a topic of its own.

Numeric Diversity

Surely the calls for the death of floating points are greatly exaggerated. Nevertheless, we found cases were indeed posits perform better. Maybe the future is one of *numeric diversity.* Why restrict ourselves to just one numeric formats when we can have multiple ones, tuned for different applications.

A Value of Floating Point

This example illustrates how to compute the value of some IEEE 754 floating point value as defined in Section 1.5

Here we will use the `binary16` type as defined by IEEE 754. It represents a $F_{5,10}$ float with bias $B = 15$ [12, p. 13]. For our example, now consider floating point

$$x = \texttt{0 10000 1001001000} \tag{A.1}$$

already written in a way that separates sign bit, exponent and mantissa fields. Bit string x does not represent a special value, so Equation 1.31 gives us

$$x = (-1)^{\texttt{0}} \cdot \texttt{1.1001001000} \cdot 2^{B-\texttt{10000}}. \tag{A.2}$$

This also explains the notation $1.m$ which asks us to interpret bit string m as the fractional part of an unsigned fixed point number with a single `1` at the front. All we have to do now is to plug in the numbers. Because sign bit s is zero, the sign is

$$(-1)^{\texttt{0}} = 1, \tag{A.3}$$

that is value x is non-negative. The fractional part of floating point x is a fixed point number, its value is

$$\texttt{1.1001001000} = 1 + \frac{1}{2} + \frac{1}{16} + \frac{1}{128} = 1.5703125. \tag{A.4}$$

Finally, we have to determine the scale factor which is based on exponent

$$e = \texttt{10000} = 16. \tag{A.5}$$

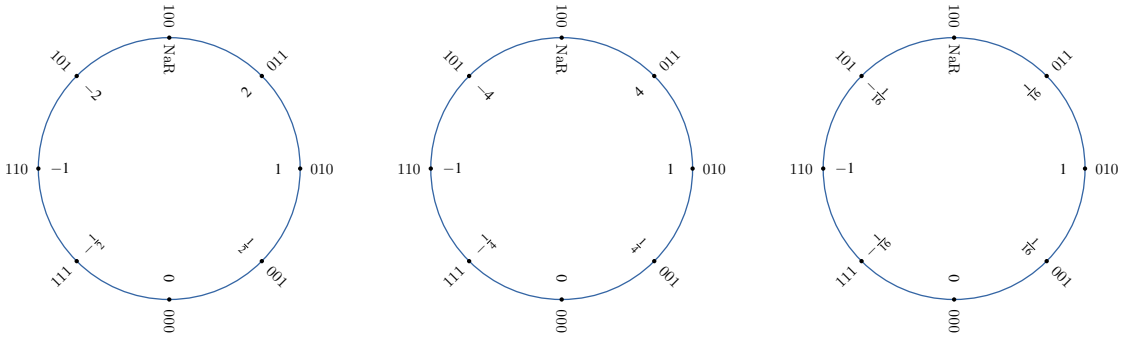
Pulling it all together, we get

$$x = (-1)^{\texttt{0}} \cdot 1.5703125 \cdot 2^{16-15} = 3.140625 \tag{A.6}$$

which is an, admittedly rough, approximation of π.

B Posits Visualized

Throughout this thesis, we reference posit types $P_{N,ES}$. It can be hard to understand certain examples without the specific type in mind. Here we include the number circles for some small resolution types available for review when necessary.

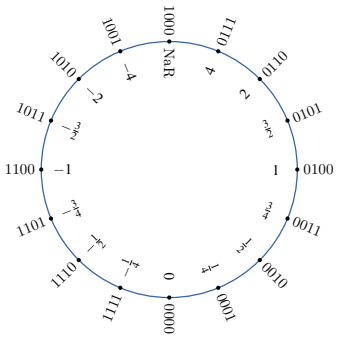


(a) $P_{3,0}$ posit.

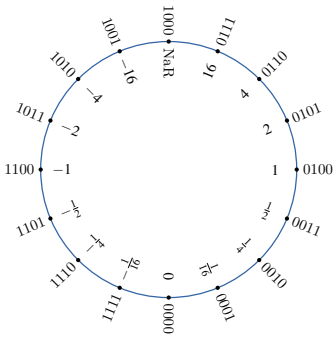
(b) $P_{3,1}$ posit.

(c) $P_{3,2}$ posit.

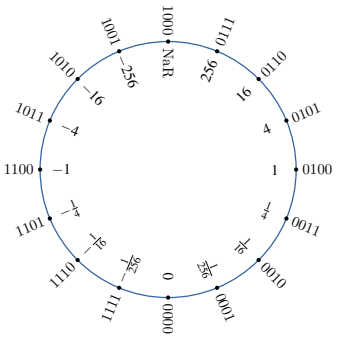
Figure B.1: Posits with $N = 3$ bits.



(a) $P_{4,0}$ posit.



(b) $P_{4,1}$ posit.



(c) $P_{4,2}$ posit.

Figure B.2: Posits with $N = 4$ bits.

$V_{N,ES}$ valids use $P_{N,ES}$ posit tiles as endpoints. As such the number circle for type $P_{N,ES}$ can also be used to illustrate valid type $V_{N,ES}$. Difference being that the respective valid type can also address the ranges between individual posit points on the circle.

C Standard Floating Point and Posit Types

Floating point and posit types can be configured with various parameters. In practice, only a small subset of parameters is actually in use. Here we list the standard types used throughout this thesis.

Name	IEEE 754	C Type	Size N	Exponent E	Mantissa M
`float8`, quarter precision	N/A	N/A	8	4	3
`float16`, half precision	N/A	N/A	16	5	10
`float32`, single precision	`binary32`	`float`	32	8	23
`float64`, double precision	`binary64`	`double`	64	11	52
`float128`, quad precision	`binary128`	N/A	128	15	112

Table C.1: Standard floating point types $F_{E,M}$, including standard types as defined by IEEE 754 [12, p. 8] and the C programming language [60, p. 36].

Name	Size N	Exponent Size ES
`posit8`	8	2
`posit16`	16	2
`posit32`	32	2
`posit64`	64	2
`posit128`	128	2

Table C.2: Standard posit types $P_{N,ES}$. Based on the latest draft of the posit standard [25, p. 6]. $V_{N,ES}$ valids use $P_{N,ES}$ posits as endpoints.

D Decimal Loss

Decimal loss is a metric for rating the accuracy of computer arithmetic systems introduced by Gustafson. This section gives a quick review of its definition.

Defintion 22. *Given some expected result x and an actual result y by some concrete implementation, then*

$$\left| \log_{10} \frac{x}{y} \right| \tag{D.1}$$

is called the "order-of-magnitude distance" or "decimal loss" of expected value x compared to actually returned value y [24, p. 78].

Example

Decimal loss is a tool for comparing the accuracy of computed values y to perfect mathematical solutions x. It provides us with a metric that correlates with the number of decimal digits that are incorrect or lost. For example, say that $x = 1$ is the expected accurate result. If for some reason our computer arithmetic system returns actual value $y = 10$, the decimal loss is

$$\left| \log_{10} \frac{1}{10} \right| = 1, \tag{D.2}$$

indicating that one decimal digit is incorrect or rather that we lost one digit of accuracy. In practice the differences between perfect solution x and computer arithmetic answer y will be more minute. Nevertheless, decimal loss can be a useful tool for evaluating accuracy, in particular because it is easy to automatically evaluate for many values.

E Properties of Binary Relations

Here we quickly review three properties of binary relations.

Defintion 23. *For some binary relation $\circ$, commutativity [61] means that arguments a and b can be switched without changing the result, that is*

$$a \circ b = b \circ a. \tag{E.1}$$

Defintion 24. *For some binary relation $\circ$, associativity [62] effectively means that parentheses can be added or omitted without changing the result, that is*

$$(a \circ b) \circ c = a \circ (b \circ d). \tag{E.2}$$

Defintion 25. *Looking at binary relations $\circ$ and $\star$, distributivity [63] means that $\star$ can be distributed over $\circ$ such that*

$$a \star (b \circ c) = (a \star b) \circ (a \star c) \quad \textit{and} \quad (b \circ c) \star a = (b \star a) \circ (c \star a). \tag{E.3}$$

Bibliography

[1] Jennifer S Light. “When computers were women”. In: *Technology and culture* 40.3 (1999), pp. 455–483.

[2] Donald E Knuth. *The Art of Computer Programming 2: Seminumerical Algorithms*. Addison-Wesley Longman, 1998.

[3] Ronald T Kneusel. *Numbers and Computers*. 2nd ed. Springer, 2017.

[4] *stdint.h - integer types*. The Open Group Base Specifications Issue 7, 2018 edition. 2018. URL: https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/stdint.h.html (visited on 2021-04-20).

[5] *Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*. Arm Limited or its affiliates. 2020. URL: https://developer.arm.com/documentation/ddi0487/fc/ (visited on 2021-06-09).

[6] Paul Zimmermann. “Comparison of three public-domain multiprecision libraries: BigNum, Gmp and Pari”. In: *Paul. Zimmermann@ inria. fr* (1998).

[7] Will Dietz et al. “Understanding integer overflow in C/C++”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 25.1 (2015), pp. 1–29.

[8] Cong Wang et al. “Go-Sanitizer: Bug-Oriented Assertion Generation for Golang”. In: *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE. 2019, pp. 36–41.

[9] Christopher Inacio and Denise Ombres. “The DSP decision: Fixed point or floating?” In: *IEEE Spectrum* 33.9 (1996), pp. 72–74.

[10] *Meet the Constants*. National Institute of Standards and Technology. 2019-12-19. URL: https://www.nist.gov/si-redefinition/meet-constants (visited on 2021-04-06).

[11] William Kahan. “IEEE standard 754 for binary floating-point arithmetic”. In: *Lecture Notes on the Status of IEEE* 754.94720-1776 (1996).

[12] “IEEE Standard for Floating-Point Arithmetic”. In: *IEEE Std 754-2008* (2008). DOI: 10.1109/IEEESTD.2008.4610935.

[13] John L Gustafson. *The End of Error: Unum Computing*. CRC Press, 2015.

[14] Laurens van Dam. *Enabling high performance posit arithmetic applications using hardware acceleration*. TU Delft Electrical Engineering, Mathematics and Computer Science, 2018.

[15] William Kahan. “A critique of John L. Gustafson’s The End of Error–Unum computation and his radical approach to computation with real numbers”. In: *23rd IEEE Symposium on Computer Arithmetic*. 2016.

[16] William Kahan. *Commentary on “THE END of ERROR” - Unum Computing*. 2016. URL: https://people.eecs.berkeley.edu/~wkahan/EndErErs.pdf (visited on 2021-04-28).

[17] Robert H Dargel, Frank R Loscalzo, and Thomas H Witt. “Automatic error bounds on real zeros of rational functions”. In: *Communications of the ACM* 9.11 (1966), pp. 806–809.

[18] “IEEE Standard for Interval Arithmetic”. In: *IEEE Std 1788-2015* (2015), pp. 1–97. DOI: 10.1109/IEEESTD.2015.7140721.

[19] Eldon R Hansen. “A generalized interval arithmetic”. In: *International Symposium on Interval Mathematics*. Springer. 1975, pp. 7–18.

[20] John L Gustafson. “A radical approach to computation with real numbers”. In: *Supercomputing frontiers and innovations* 3.2 (2016), pp. 38–53. DOI: 10.14529/jsfi160203.

[21] David W. Cantrell. *”Projectively Extended Real Numbers.” From MathWorld–A Wolfram Web Resource, created by Eric W. Weisstein.* URL: https://mathworld.wolfram.com/ProjectivelyExtendedRealNumbers.html (visited on 2021-02-05).

[22] Walter Tichy. “Unums 2.0: An interview with john l. gustafson”. In: *Ubiquity* 2016.September (2016), pp. 1–16.

[23] John L Gustafson. *Posit arithmetic*. 2017. URL: https://www.posithub.org/docs/Posits4.pdf (visited on 2021-02-12).

[24] John L Gustafson and Isaac T Yonemoto. “Beating floating point at its own game: Posit arithmetic”. In: *Supercomputing Frontiers and Innovations* 4.2 (2017), pp. 71–86.

[25] *Posit Standard Documetation Release 4.9-draft*. Posit Working Group. 2020-11-10. URL: https://groups.google.com/g/unum-computing/c/55Y-YIwD0bs/m/SwZBuX2WBwAJ (visited on 2020-12-03).

[26] *Survey of Posit Hardware and Software Development Efforts (July 2019)*. 2019. URL: https://posithub.org/docs/PDS/PositEffortsSurvey.html (visited on 2021-02-20).

[27] Isaac Yonemoto. *interplanetary-robot/SigmoidNumbers*. GitHub. 2018. URL: https://github.com/interplanetary-robot/SigmoidNumbers (visited on 2021-03-25).

[28] John L Gustafson. Hacker News. 2017-04-02. URL: https://news.ycombinator.com/item?id=14015194 (visited on 2021-01-25).

[29] Timothy Hickey, Qun Ju, and Maarten H Van Emden. “Interval arithmetic: From principles to implementation”. In: *Journal of the ACM (JACM)* 48.5 (2001), pp. 1038–1068.

[30] Eric W Weisstein. *”Infinitesimal.” From MathWorld–A Wolfram Web Resource.* URL: `https://mathworld.wolfram.com/Infinitesimal.html` (visited on 2021-06-16).

[31] Oliver Keszöcze et al. “Aarith: An Arbitrary Precision Number Library”. In: *ACM/SIGAPP Symposium On Applied Computing* (virtual conference). 2021-03-22/2020-11-26. DOI: 10.1145/3412841.3442085.

[32] Cerlane Leong. *cerlane/SoftPosit*. Gitlab. 2021. URL: `https://gitlab.com/cerlane/SoftPosit` (visited on 2021-06-17).

[33] David Thien. *DavidThien/softposit-rkt*. GitHub. 2019. URL: `https://github.com/DavidThien/softposit-rkt` (visited on 2021-06-17).

[34] Bill. *billzorn/sfpy*. GitHub. 2019. URL: `https://github.com/billzorn/sfpy` (visited on 2021-06-17).

[35] Milan K. *milankl/SoftPosit.jl*. GitHub. 2020. URL: `https://github.com/milankl/SoftPosit.jl` (visited on 2021-06-17).

[36] Clément Guérin. *libcg/bfp*. GitHub. 2019. URL: `https://github.com/libcg/bfp` (visited on 2021-07-24).

[37] Emanuele Ruffaldi. *eruffaldi/cppPosit*. GitHub. 2019. URL: `https://github.com/eruffaldi/cppPosit` (visited on 2021-07-24).

[38] E. Theodore L. Omtzigt et al. “Universal Numbers Library: design and implementation of a high-performance reproducible number systems library”. In: *arXiv:2012.11011* (2020).

[39] Vandevoorde David and Nicolai M Josuttis. “C++ Templates: The Complete Guide”. In: *Addison-Wesley Professional* (2017).

[40] Saba Akram and Quarrat Ul Ann. “Newton raphson method”. In: *International Journal of Scientific & Engineering Research* 6.7 (2015), pp. 1748–1752.

[41] BC Carlson. “An algorithm for computing logarithms and arctangents”. In: *Mathematics of Computation* 26.118 (1972), pp. 543–549.

[42] Gerhard Merziger et al. *Formeln + Hilfen: Höhere Mathematik*. 7th ed. Binomi Verlag, 2014-01.

[43] David Janzen and Hossein Saiedian. “Test-driven development concepts, taxonomy, and future direction”. In: *Computer* 38.9 (2005), pp. 43–50.

[44] Jeffrey Werner Bezanson. *Julia: an efficient dynamic language for technical computing*. Massachusetts Institute of Technology, 2012.

[45] Mikko Tommila. *apfloat*. 2005-02-28. URL: `http://www.apfloat.org/apfloat/2.41/apfloat.pdf` (visited on 2021-07-08).

[46] Jack J Dongarra, Piotr Luszczek, and Antoine Petitet. "The LINPACK benchmark: past, present and future". In: *Concurrency and Computation: practice and experience* 15.9 (2003), pp. 803–820.

[47] Nicholas J Higham and Srikara Pranesh. "Simulating low precision floating-point arithmetic". In: *SIAM Journal on Scientific Computing* 41.5 (2019), pp. C585–C602.

[48] Bill Young. *CS429: Computer Organization and Architecture: Integers*. Department of Computer Science, University of Texas at Austin. 2019-06-10. URL: https://www.cs.utexas.edu/~byoung/cs429/slides3-integers-4up.pdf (visited on 2021-07-22).

[49] David Goldberg. "What every computer scientist should know about floating-point arithmetic". In: *ACM computing surveys (CSUR)* 23.1 (1991), pp. 5–48.

[50] Scott B. Baden. *Lecture 17: Floating point*. Carnegie Mellon University. 2016. URL: https://www.andrew.cmu.edu/user/gkesden/ucsd/classes/wi17/cse160-a/applications/ln/lecture17.pdf (visited on 2021-04-14).

[51] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 3rd ed. Prentice Hall, 2010.

[52] Seungwon. *bowbowbow/DecisionTree*. GitHub. 2020. URL: https://github.com/bowbowbow/DecisionTree (visited on 2021-05-14).

[53] Nick Silvestri. *nsilvestri/cpp-raytracer*. GitHub. 2018. URL: https://github.com/nsilvestri/cpp-raytracer (visited on 2021-05-19).

[54] Andrew S Glassner. *An Introduction to Ray Tracing*. 1st ed. Morgan Kaufmann, 1989.

[55] *Half-Precision Floating Point*. GNU. 2021. URL: https://gcc.gnu.org/onlinedocs/gcc/Half-Precision.html (visited on 2021-05-19).

[56] Arash Mohammadi et al. "OpenGA, a C++ Genetic Algorithm Library". In: *Systems, Man, and Cybernetics (SMC), 2017 IEEE International Conference on*. IEEE. 2017, pp. 2051–2056.

[57] John H Holland. "Genetic algorithms". In: *Scientific american* 267.1 (1992), pp. 66–73.

[58] Petr Rek and Lukas Sekanina. "TypeCNN: CNN Development Framework With Flexible Data Types". In: *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2019, pp. 292–295. DOI: 10.23919/DATE.2019.8714855.

[59] Yann LeCun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[60] Dennis M Ritchie, Brian W Kernighan, and Michael E Lesk. *The C programming language*. 2nd ed. Prentice Hall Englewood Cliffs, 1988.

[61] Eric W. Weisstein. *Commutative. From MathWorld–A Wolfram Web Resource*. URL: `https://mathworld.wolfram.com/Commutative.html` (visited on 2021-05-28).

[62] Eric W. Weisstein. *Associative. From MathWorld–A Wolfram Web Resource*. URL: `https://mathworld.wolfram.com/Associative.html` (visited on 2021-05-28).

[63] Eric W. Weisstein. *Distributive. From MathWorld–A Wolfram Web Resource*. URL: `https://mathworld.wolfram.com/Distributive.html` (visited on 2021-05-28).