

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/323467921>

Universal Coding of the Reals: Alternatives to IEEE Floating Point

Conference Paper · February 2018

DOI: 10.1145/3190339.3190344

CITATIONS

32

READS

3,304

3 authors, including:



Peter Lindstrom

Lawrence Livermore National Laboratory

101 PUBLICATIONS 5,809 CITATIONS

[SEE PROFILE](#)



Jeffrey Hittinger

Lawrence Livermore National Laboratory

83 PUBLICATIONS 1,379 CITATIONS

[SEE PROFILE](#)

Universal Coding of the Reals: Alternatives to IEEE Floating Point

Peter Lindstrom

Lawrence Livermore National Lab
pl@llnl.gov

Scott Lloyd

Lawrence Livermore National Lab
scott@llnl.gov

Jeffrey Hittinger

Lawrence Livermore National Lab
hittinger@llnl.gov

ABSTRACT

We propose a modular framework for representing the real numbers that generalizes IEEE, POSITS, and related floating-point number systems, and which has its roots in universal codes for the positive integers such as the Elias codes. This framework unifies several known but seemingly unrelated representations within a single schema while also introducing new representations. We particularly focus on variable-length encoding of the binary exponent and on the manner in which fraction bits are mapped to values. Our framework builds upon and shares many of the attractive properties of POSITS but allows for independent experimentation with exponent codes, fraction mappings, reciprocal closure, rounding modes, handling of under- and overflow, and underlying precision.

CCS CONCEPTS

- Mathematics of computing → Coding theory; Arbitrary-precision arithmetic;

KEYWORDS

Number representations, floating point, posits, universal coding, tapered precision, roundoff error, numerical algorithms

ACM Reference Format:

Peter Lindstrom, Scott Lloyd, and Jeffrey Hittinger. 2018. Universal Coding of the Reals: Alternatives to IEEE Floating Point. In *CoNGA 2018: Conference for Next Generation Arithmetic, March 28, 2018, Singapore, Singapore*. ACM, New York, NY, USA, 14 pages. <https://doi.org/10.1145/3190339.3190344>

1 INTRODUCTION

As the high-performance computing community pushes toward exascale computing, it is becoming increasingly clear that *data movement* will be the dominant performance bottleneck for the foreseeable future, leading developers to re-evaluate the need for wide data types that consume precious memory bandwidth. However, given the choice between single and double precision storage, few application opt for the former, in part due to the limited precision and dynamic range supported in IEEE based single precision hardware.

One potential compromise is to use *mixed-precision arithmetic* [2], wherein a mixture of double, single, and even half precision are

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CoNGA 2018, March 28, 2018, Singapore, Singapore

© 2018 Association for Computing Machinery.

ACM ISBN 978-1-4503-6414-0/18/03...\$15.00

<https://doi.org/10.1145/3190339.3190344>

used depending on sensitivity of the overall calculation to the specific precision requirements of individual calculations. Another line of attack is to fundamentally rethink the representation of floating-point numbers, as pioneered by the recent work of Gustafson and others, including type-1 [7] and type-2 [8] *unums*, as well as their fixed-precision successor, *posits* [9]. These representations have shown to provide a better tradeoff between precision and dynamic range, and in effect increase the accuracy per bit stored.

This revived interest in floating-point representations builds on the work done in the 1970s and 1980s [3, 10, 12, 13, 21], before the IEEE standard was developed and codified in hardware. The choices made in IEEE leave much to be desired when accurate arithmetic and reproducibility of numerical computations are paramount, including support for an excessive number of not-a-number (NaN) representations, gradual underflow but not overflow, expensive exceptions such as subnormals, ambiguity due to two representations of zero, etc.

In order to assess how different choices of encoding real numbers impact accuracy in scientific applications, we have developed a modular framework that encompasses POSITS and IEEE like representations, and that also supports other schemes proposed in the literature. We in particular identify the key difference between POSITS and IEEE in terms of how the floating-point exponent is encoded and propose several alternatives that extend well-known *universal coding* schemes of positive integers. We further show how reciprocal closure can be implemented using nonlinear mappings of fraction bits to values that also allow for smooth number mappings, such as an exponential mapping. This framework allows for a modular design and for the evaluation of the interaction between independent concepts without the need to specialize the code for precision or combination of modules.

We evaluate the accuracy of several types expressible in our framework in terms of basic arithmetic operations, common linear algebra routines, and a more sophisticated mini-application that simulates a hydrodynamic shock propagating through a two-dimensional domain. These results confirm that IEEE consistently performs poorly relative to alternative representations, while POSITS tend to perform among the best of the evaluated types.

2 BACKGROUND

We begin with notation. Let x be the p -bit binary representation of the real number $y = \mathcal{D}(x) \in \mathbb{R}$, and let $\mathcal{E}(y) = x \in \mathbb{Z}$ map y to its corresponding p -bit string. In this paper, we consider the cases $p \in \{16, 32, 64\}$. The operators \mathcal{E} and \mathcal{D} encode reals and decode bit strings, respectively. Note that \mathcal{E} is non-injective, that is, many reals map to the same binary representation as dictated by rounding rules. Therefore, \mathcal{E}^{-1} may not exist. We will treat rounding separately,

and therefore we will, for convenience, assume that $y = \mathcal{D}(\mathcal{E}(y))$ such that rounding can be ignored.

Sometimes we will find it more convenient to express x independently of the precision p , and we define $\hat{x} = 2^{1-p}x$, such that $\hat{x} \in (-1, 1)$, with corresponding functions $\hat{\mathcal{E}}$ and $\hat{\mathcal{D}}$.

In this paper we assume that any non-zero finite real number y can be represented in terms of its sign, s , base-2 exponent, e , and fraction, f , as

$$y = (-1)^s 2^e \phi(f), \quad 0 \leq f < 1, \quad (1)$$

where $\phi : [0, 1] \rightarrow [1, 2)$ is a monotonic function. Aside from IEEE, which uses a sign-magnitude representation, all of our number formats use a two's complement representation. Hence the sign is trivially determined from the most significant bit.

Our focus is on two orthogonal concepts:

- the encoding of the exponent, e , and
- the mapping, ϕ , of bit strings representing fractions to real values.

The encoding, x , of y is given by the concatenation of the encoding of the sign, exponent, and fraction. Therefore, for convenience, we sometimes use $\mathcal{E}(e)$ to denote the substring of x that corresponds to the encoding of the exponent, e . For reasons that will become clear later, we distinguish between mappings, ϕ , of *subunitary* numbers, with $e < 0$, and *superunitary* numbers, with $e \geq 0$.

We use $\beta(y)$ to denote the standard binary representation of natural numbers, y . For example, $\beta(25) = 11001_2$. $\beta_p(y)$ denotes y as a p -bit string, possibly with leading zeros. $|x|$ denotes the length of bit string x , not counting any leading zeros. Thus, for a positive integer, y , we have $|\beta(y)| = 1 + \lfloor \log_2 y \rfloor$, e.g., $|\beta(25)| = 5$.

We find it instructive to study the *realmax sequence* of a number scheme, which is the sequence of largest finite reals representable as the precision, p , increases. This sequence is given by $\{\hat{\mathcal{D}}(1 - 2^{1-p})\}_p$.

2.1 IEEE-754

We begin by considering the IEEE-754 single-precision (32-bit) format [1]. IEEE encodes the exponent, e , in binary using $m = 8$ bits. To support subunitary numbers, the exponent is necessarily signed but uses a biased representation, where $0 \leq \mathcal{E}(e) = 2^{m-1} - 1 + e \leq 2^m - 1$.

IEEE uses a single mapping, $\phi(f) = 1 + f$, for sub- and superunitary numbers. One exception to this rule is when $\mathcal{E}(e) = 00 \dots 0_2$, in which case $\phi(f) = 0 + f$. These very small numbers are called *subnormal* and allow for gradual underflow.

2.2 Posits

Invented by Gustafson and Yonemoto [9], POSITS are one of the representations that can be expressed in our framework. The main idea behind POSITS is to represent the projective reals by partitioning them into four non-overlapping open intervals whose endpoints are given by $\{-\infty, -1, 0, +1, +\infty\}$. Each such interval encodes $2^{p-2} - 1$ values. The four remaining values are the interval endpoints, with $-\infty$ and $+\infty$ consolidated as the single “point at infinity,” $\pm\infty$, corresponding to 1/0 (see Fig. 1a).

Although POSITS, like IEEE, model real numbers based on Eq. (1), they differ from IEEE in the following important ways:

- POSITS use a variable-length encoding of the exponent, assigning fewer bits to small exponents, i.e., to the exponents of values near one. This feature gives rise to the concept of *tapered precision* [13], where more precision is allocated to the more commonly occurring numbers near unity.
- The posit exponent encoding is parameterized by a single environment variable, es , which denotes the *exponent size* in bits, similar to how IEEE varies the number of exponent bits with precision. Conceptually, the value of a posit exponent also depends on the *regime*; a variable-length bit string that precedes the exponent. Below we present an alternative interpretation of posit exponents more suited to our framework.
- POSITS do not support the notion of subnormal values. Except for $y = 0$, every finite value uses $\phi(f) = 1 + f$.
- Unlike IEEE, which supports an enormous number of ways in which to express NaN (not a number), POSITS do not have a NaN representation. Using an environment setting, $\pm\infty$ may be treated as NaN instead.

Our framework is built upon and can be thought of as an extension of POSITS to other exponent coding schemes and nonlinear fraction maps.

2.3 Reciprocal Closure

Whereas IEEE and POSITS are both closed under negation, neither is closed under reciprocation for reals other than powers of two. That is, for $y = \mathcal{D}(x)$, $1/y$ may not be exactly representable. The *type-2 unums* [8] that preceded POSITS were unique in that they were defined implicitly in terms of reciprocal closure, where for each representable real in $[1, \infty)$ its reciprocal in $(0, 1]$ was also represented, and vice versa. We show in Section 4 how this lack of reciprocal closure for both IEEE and POSITS can easily be recovered via judicious choices of fraction maps, i.e., we may close any number representation within our framework independent of other encoding decisions. Using our framework, we will later show how POSITS can be slightly modified to support full reciprocal closure.

3 EXPONENT CODING AND UNIVERSAL REPRESENTATIONS

The two examples above show how dyadic, signed fractions can be encoded. To proceed to other representations, we first consider known methods for encoding positive integers. IEEE encodes positive integers $y = 2^e + r$ with $0 \leq r < 2^e$ using a zero sign bit, a fixed-length representation of e , and an e -bit significand, r . (Remaining trailing zero-bits of the significand can be thought as padding to fit the code in exactly p bits.) The decision in IEEE-754 to fix the number of exponent bits fundamentally limits the range of numbers that can be encoded. That is, regardless of the precision, p , available, one cannot exceed the lower and upper bounds implied by the exponent size.

The situation is different with posits, where in principle any number can be represented using a sufficiently large number of regime bits. Posits are *universal* in the sense that (1) any integer has a unique representation given by a *prefix code*,¹ (2) the number

¹A prefix code is a set of codewords for which no codeword is a prefix of any other codeword. The length of each codeword is encoded as part of the codeword itself.

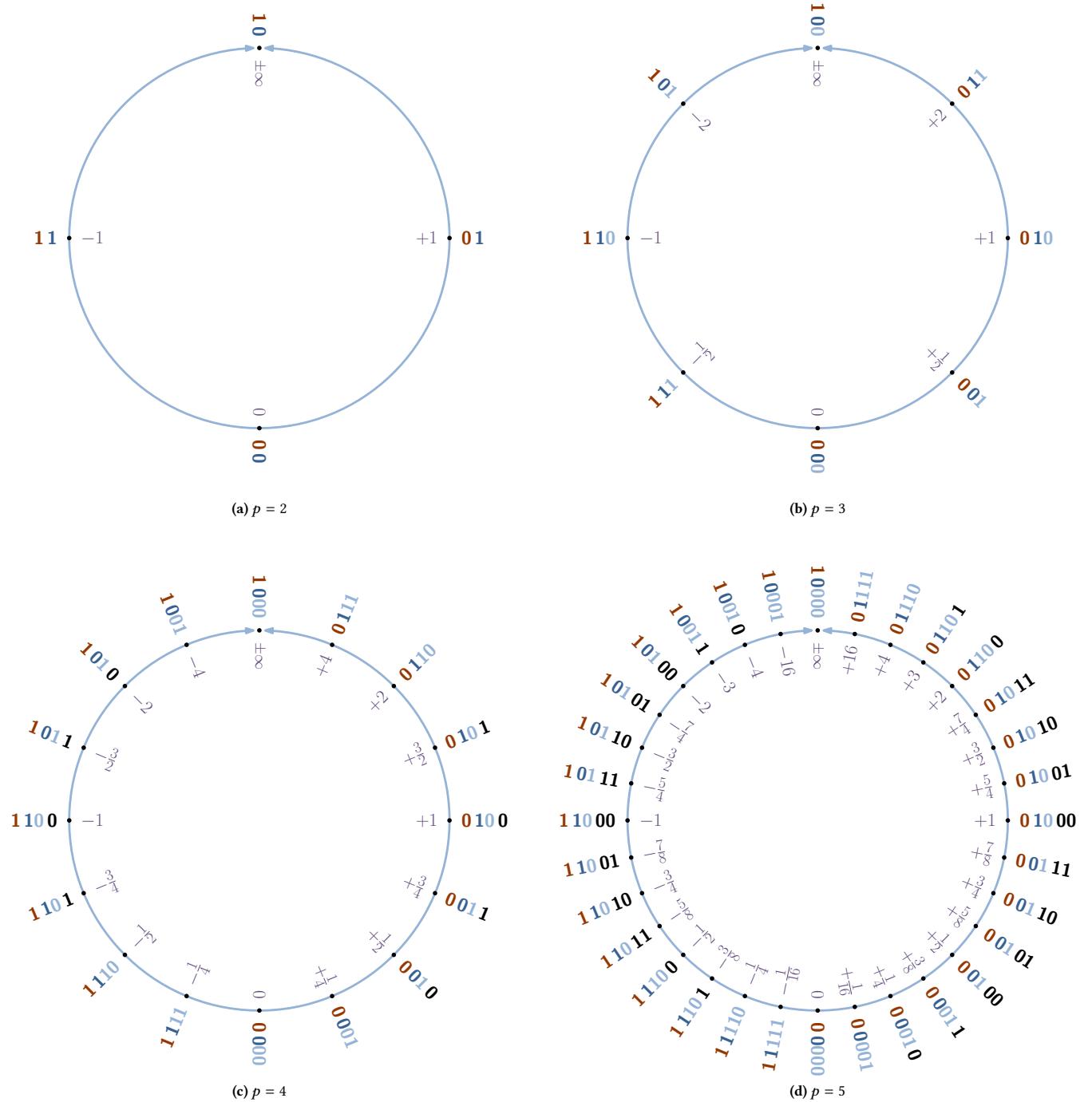


Figure 1: Ringplot [8, 9] depicting the projective reals and how p -bit strings (shown on the outside) map to the reals (shown on the inside) for the Elias δ representation (see Section 6.3). Notice how numbers ending in a zero-bit remain in place when reducing the precision by one bit. The sign bit is shown in red, exponent bits in blue, and fraction bits in black.

of bits required to represent positive integers is monotonically non-decreasing, and (3) the length of encoded integers is within

a constant factor of the length $|\beta(y)| = 1 + \lfloor \log_2 y \rfloor$ of the binary encoding of y . However, the posit representation is asymptotically

<i>decimal</i>	<i>binary</i>	<i>IEEE half</i>	<i>binary(5)</i>	<i>Elias γ</i>	<i>posit(1)</i>	<i>posit(2)</i>	<i>Elias $\delta(0)$</i>	<i>Elias $\omega(3)$</i>
1	1	0 01111	0 10000	0 10	0 100	0 1000	0 10	0 10
2	10	0 10000 0	0 10001 0	0 110 0	0 101 0	0 1001 0	0 110 0	0 110 0
3	11	0 10000 1	0 10001 1	0 110 1	0 101 1	0 1001 1	0 110 1	0 110 1
4	100	0 10001 00	0 10010 00	0 1110 00	0 110 000	0 10 10 00	0 11100 00	0 11100 00
5	101	0 10001 01	0 10010 01	0 1110 01	0 110 001	0 10 10 01	0 11100 01	0 11100 01
6	110	0 10001 10	0 10010 10	0 1110 10	0 110 010	0 10 10 10	0 11100 10	0 11100 10
7	111	0 10001 11	0 10010 11	0 1110 11	0 110 011	0 10 10 11	0 11100 11	0 11100 11
8	1000	0 10010 000	0 10011 000	0 11110 000	0 110 1000	0 10 11 000	0 11101 000	0 11101 000
9	1001	0 10010 001	0 10011 001	0 11110 001	0 110 1001	0 10 11 001	0 11101 001	0 11101 001
10	1010	0 10010 010	0 10011 010	0 11110 010	0 110 1010	0 10 11 010	0 11101 010	0 11101 010
11	1011	0 10010 011	0 10011 011	0 11110 011	0 110 1011	0 10 11 011	0 11101 011	0 11101 011
12	1100	0 10010 100	0 10011 100	0 11110 100	0 110 1100	0 10 11 100	0 11101 100	0 11101 100
13	1101	0 10010 101	0 10011 101	0 11110 101	0 110 1101	0 10 11 101	0 11101 101	0 11101 101
14	1110	0 10010 110	0 10011 110	0 11110 110	0 110 1110	0 10 11 110	0 11101 110	0 11101 110
15	1111	0 10010 111	0 10011 111	0 11110 111	0 110 1111	0 10 11 111	0 11101 111	0 11101 111
16	10000	0 10011 0000	0 10100 0000	0 11110 0000	0 1110 0000	0 110 0000	0 111000 0000	0 1110000 0000
17	10001	0 10011 0001	0 10100 0001	0 11110 0001	0 1110 0001	0 110 0001	0 111000 0001	0 1110000 0001
exponent	binary	binary	unary	Golomb-Rice	Golomb-Rice	Elias γ	Elias ω	
realmin ($p = 16$)	2^{-24}	$2^{-16} + 2^{-26}$	2^{-14}	2^{-28}	2^{-56}	2^{-8192}	2^{-65024}	
realmax ($p = 16$)	$2^{+16} - 2^{+5}$	$2^{+16} - 2^{+5}$	2^{+14}	2^{+28}	2^{+56}	2^{+8192}	2^{+65024}	
epsilon ($p = 16$)	2^{-10}	2^{-10}	2^{-13}	2^{-12}	2^{-11}	2^{-13}	2^{-13}	
flintmax ($p = 16$)	2^{+11}	2^{+11}	2^{+7}	2^{+9}	2^{+10}	2^{+8}	2^{+8}	

Table 1: Binary representations of small positive integers for some of the encoding schemes considered in this paper, with linear ϕ and 16-bit precision. The sign bit is shown in red, exponent bits in blue, and fraction bits in black. *exponent* refers to the encoding scheme used for the exponent. *realmmin* and *realmmax* are the smallest respectively largest representable positive real numbers. *epsilon* is the smallest positive real value, ϵ , such that $1 + \epsilon$ can be exactly represented. *flintmax* is the smallest positive integer, y , such that $y + 1$ cannot be exactly represented.

suboptimal in that the limit

$$\lim_{y \rightarrow \infty} \frac{|\mathcal{E}(y)|}{|\beta(y)|} = 1 \quad (2)$$

is not satisfied. Indeed, asymptotically optimal codes for the positive integers are well-known in the information theory community [16], where they often serve as the basis for encoding residuals in data compression. One implication of universality is that any finite positive integer has a finite-length representation, and unlike in IEEE it is only the precision, p , that limits the size of numbers that can be represented.

Several of the best-known universal integer codes were developed by Elias [4], including the γ , δ , and ω codes (see Table 1 and below). Several others have been proposed [16] and can easily be used within our framework. Due to space restrictions, we will limit our focus to the Elias codes.

3.1 Generalization to Signed Rationals

Equipped with these integer encoding schemes, we now explain how to encode superunitary rationals, subunitary rationals, and negative numbers. As with fixed-precision representations like IEEE and POSITS, any integer whose encoding requires fewer than p bits, where p is the precision, is represented in p bits by appending zero-bits.

Rational numbers are supported in a manner similar to IEEE and POSITS. That is, we may pad the encoding of short integers represented in less than p bits with arbitrary bit strings to represent

superunitary dyadic rationals, where these appended bits represent the fraction to the right of the binary point, with place value $2^{-1}, 2^{-2}, 2^{-3}, \dots$. Integers and rationals wider than p bits must be truncated during encoding. When decoding positive and negative numbers, any truncated bits to the right are replaced with zero-bits.

By appending fraction bits, we extend our representation from \mathbb{Z}^+ to $\mathbb{Q}_2^+ \subset \mathbb{Q}^+$, covering the first (northeast) quadrant of Fig. 1a. These superunitary numbers, for which $e \geq 0$, are all encoded with the binary prefix 01₂.

Subunitary numbers are handled in a similar manner, where in essence we resort to using a two's complement encoding of the exponent, e . We note that any given $e \geq 0$ corresponds to a whole interval of numbers, $[2^e, 2^{e+1})$, and each superunitary interval is in one-to-one correspondence with a subunitary interval, $[2^{-(e+1)}, 2^{-e})$, where the lower bounds 2^e and $2^{-(e+1)}$ represent the intervals. That is, each $e \geq 0$ is in correspondence with $-(e+1) = \bar{e}$, where \bar{e} denotes bitwise complement. Hence negative exponents can be handled by flipping their bits and then making use of the decoder for nonnegative exponents. The subunitary positive numbers, with $e < 0$, have a binary prefix of 00₂.

Extremely small and large numbers require special attention, because using variable-length encoding it is possible that not even the exponent fits in p bits. For instance, due to exponent truncation e may correspond to the wider interval $[2^e, 2^{e+2})$ whose corresponding subunitary interval is $[2^{-(e+2)}, 2^e)$. In this case, the lower bound 2^e is in correspondence with the other lower bound $2^{-(e+2)}$ (i.e., not with $2^{-(e+1)}$).

Negative numbers are, as alluded to above, represented using a two's complement format, similar to **posit**s but unlike **IEEE**, which uses a sign-magnitude representation (which introduces the somewhat awkward concept of -0 distinct from but numerically equal to $+0$). Using two's complement, a binary representation, x , satisfies $\mathcal{D}(-x) = -\mathcal{D}(x)$, where $-x$ denotes two's complement negation, i.e., $-x = \bar{x} + 1 \bmod 2^p$. In this way, we establish a one-to-one mapping between $(-\infty, 0)$ and $(0, +\infty)$ (see Fig. 1). Note that the most significant bit of x is 0 iff $\mathcal{D}(x) \geq 0$.

Our framework guarantees the following properties:

- **Nesting:** $\mathcal{D}(x) = \mathcal{D}(x0)$ for any nonempty bit string, x , where $x0$ denotes x appended with a zero-bit. Thus, the p -bit reals are a proper subset of the $(p+1)$ -bit reals.
- **Lexicographic ordering:** $\mathcal{D}(x0) < \mathcal{D}(x1)$ for any nonempty bit string, x . As a corollary, when neither x nor x' corresponds to $\pm\infty$, $\mathcal{D}(x) < \mathcal{D}(x') \iff x < x'$, where $x < x'$ refers to two's complement comparison of bit strings.
- **Closure:** Each interval $(-\infty, -1), (-1, 0), (0, +1), (+1, +\infty)$ contains $2^{p-2} - 1$ values, ensuring closure under negation and, if desired, reciprocation.
- **Power-of-two reciprocal closure:** For $e \in \mathbb{Z}$, $\mathcal{E}(2^{-e}) = 100\dots0_2 - \mathcal{E}(2^e)$.

4 FRACTION MAPS

By far the most common choice of fraction map is the linear one, $\phi(f) = 1 + f$ ($0 \leq f < 1$), which does little but prepend the implicit leading one-bit well-known from **IEEE** and, more recently, **posit**s. Our decision to consider other maps originally stems from the notion of reciprocal closure supported by type-2 unums [8], but we shall see that these maps have a more general purpose and allow expanding our representations to important cases.

In type-2 unums, the sign bit, which denotes additive inversion, is followed by another bit that denotes multiplicative inversion. In this manner, we have $\mathcal{D}(100\dots0_2 - x) = \mathcal{D}(x)^{-1}$. That is, for any representable $y = \mathcal{D}(x)$, we have an exact representation of its reciprocal, y^{-1} , which is obtained by flipping the $p-1$ least significant bits of x and adding one. In Fig. 1, reciprocation corresponds to a vertical flip of the unit circle.

This explicit treatment of reciprocation can also be implicitly accomplished by using different fraction maps for sub- (ϕ^-) and superunitaries (ϕ^+). To make this more clear, let us first consider how numbers in the interval $[1, 2]$ map to their reciprocals in $[\frac{1}{2}, 1]$ when the superunitaries are mapped linearly via $\phi^+(f) = 1 + f$. Since $s = e = 0$, we have $\mathcal{D}(x) = \phi^+(f)$, where f is the fraction encoded in x . For each $0 \leq f < 1$, we need for the mirrored fraction $1-f$ to correspond to the reciprocal of $\phi^+(f)$, i.e., $2^{-1}\phi^-(1-f) = \phi^+(f)^{-1}$. Equivalently, ϕ^- and ϕ^+ must satisfy the following conjugacy relationship:

$$\phi^-(f) = \frac{2}{\phi^+(1-f)}. \quad (3)$$

Thus, when $\phi^+(f) = 1 + f$, we have as conjugate map the reciprocal linear map $\phi^-(f) = \frac{2}{2-f}$. Notice how ϕ^- is monotonic with $\phi^-(0) = 1$ and $\phi^-(1) = 2$, as required. To summarize, we may implement reciprocal closure simply by choosing the two fraction maps to satisfy conjugacy.

Another important fraction map is the exponential one: $\phi(f) = 2^f$, which is self-conjugate, i.e., $\frac{2}{2^{1-f}} = 2^f$. When combined with a binary encoding of the exponent, we have

$$y = (-1)^s 2^e 2^f = (-1)^s 2^{e+f}, \quad (4)$$

with e an integer and f a fraction $0 \leq f < 1$. This choice ensures a smooth, exponential representation over the entire domain, with no “wobbling accuracy” [8], also known as a *logarithmic number system* (LNS) [17]. A linear rational, self-conjugate map that very closely approximates 2^f is given by: $\phi(f) = \frac{1+pf}{1-qf}$ with $p = \sqrt{2}-1$, $q = \frac{1-p}{2}$.

The idea of fraction maps as one being orthogonal to exponent encoding and reciprocal closure is beneficial in terms of constructing new number systems. Our templated design allows for any combination of exponent encoding schemes and sub- and superunitary fraction maps, accommodating a very wide range of number representations under a single umbrella. Desired properties like reciprocal closure are obtained “for free” by simply choosing conjugate maps, and no special cases are needed in the code that would treat bits of the binary representation specially.

We note that the implementation of fraction maps (and their inverses) other than the linear one has an associated computational expense, and it is unlikely that nonlinear maps would be implemented in hardware. Nevertheless, for software implementations of number systems—especially for very low precision like 8 and 16 bits—nonlinear maps become an attractive alternative in non-performance critical applications; especially the linear reciprocal and rational maps, which involve division only.

5 ROUNDING, UNDERFLOW, AND OVERFLOW

Rounding is the process of mapping arbitrary reals, $y \in \mathbb{R}$, to one of the representable codes, x . Conceptually, rounding brackets y such that $y_l = \mathcal{D}(x) \leq y < \mathcal{D}(x+1) = y_u$ and then determines which of y_l and y_u is “closer” to y . Equivalently, one tests whether y is above or below the *tie point* given by the *arithmetic mean* $(y_l + y_u)/2$. For extremely small and large numbers, however, we may have $y_u > 2y_l$, in which case these two numbers are both powers of two with no room for significand bits. The posit solution is then to use as tie point the *geometric mean* $\sqrt{y_l y_u}$, which is equivalent to averaging their exponents.

We use a similar but more general strategy that extends to any number representation. Our tie points are given by the set of 2^p numbers introduced by extending the precision p by one bit, as these fall somewhere in between the p -bit numbers. This approach results in the standard rounding rules for **IEEE** and **posit**s, but also handles recursive codes like Elias ω in those cases where the “exponent of the exponent” does not fit. To avoid bias, we use *balanced rounding*, also known as “round ties to even,” by alternatingly rounding tie points up and down.

Rounding for values smaller than y_{min} and larger than y_{max} is done differently. We adopt the posit solution as a default, where such values never underflow to zero or overflow to infinity, but are snapped to y_{min} or y_{max} . We do allow this behavior to be overridden, however.

6 PROPERTIES

In this section, we establish several properties of and relationships between the number representations considered so far. We will also see how other published representations can be cast in our framework.

6.1 Posits

As described in [9], posits use a combination of “regime” and “exponent” bits. Taken together, however, these bits can be thought of as simply the Golomb-Rice [16] representation of the exponent, e . Golomb-Rice codes are parameterized by m —the number of bits that are encoded in binary—with the leading bits encoding $\lfloor e/2^m \rfloor \geq 0$ in unary. The trailing binary portion encodes $e \bmod 2^m$. Thus, the posit regime bits correspond to the unary part while the exponent bits correspond to the binary part, with $es = m$. This interpretation puts posits on equal footing with the other representations considered here, and makes for an easier comparison with representations like IEEE. For instance, IEEE can be thought of as using a length-limited Golomb-Rice code with m exponent bits and no room for the unary prefix.

6.2 Elias γ

The lexicographic version of Elias γ encodes the (positive) exponent, e , in unary, α , making it a special case of posits with $es = 0$. That is,

$$\gamma(2^e + r) = \begin{cases} 0, & \text{if } e = 0, \\ 1\alpha(e)\beta_e(r), & \text{otherwise,} \end{cases} \quad (5)$$

where $0 \leq r < 2^e$, $\alpha(1) = 0$, $\alpha(e) = 1\alpha(e-1)$, and $\beta_e(r)$ denotes the e -bit binary representation of r .

This number representation has other interesting properties. Let $\hat{x} = 2^{1-p}x$ such that $\hat{x} \in (-1, 1)$. Define $h(\hat{x}) = \frac{\hat{x}}{1-|\hat{x}|}$. $h(\hat{x})$ describes a smooth mapping that is closed under reciprocation, but it does not directly fit into our framework. $h(\hat{x})$ has a realmax sequence of $(1, 3, 7, 15, \dots, 2^{p-1} - 1)$. Using a small modification we show how it relates to Elias γ , whose realmax sequence is $(1, 2, 4, 8, \dots, 2^{p-2})$. Whenever $\hat{D}_\gamma(\hat{x})$ is non-negative and a power of two, we have

$$\hat{D}_\gamma(\hat{x}) = \begin{cases} \frac{2h(\hat{x})}{h(\hat{x})+1} = 2\hat{x}, & \text{if } 0 \leq \hat{x} < \frac{1}{2}, \\ \frac{h(\hat{x})+1}{2} = \frac{1}{2-2\hat{x}}, & \text{if } \hat{x} \geq \frac{1}{2}. \end{cases} \quad (6)$$

For these to hold on non-powers of two, it is easy to show that the subunitary map must be linear and that the superunitary map be the linear reciprocal map $\frac{2}{2-\hat{x}}$. The resulting representation is closed under reciprocation and is (not just piecewise) linear on the subunitaries, $-\frac{1}{2} \leq \hat{x} \leq \frac{1}{2}$.

6.3 Elias δ

The Elias δ code [4] uses the γ code to encode the non-negative exponent, e . However, γ is defined only for positive integers, and therefore an exception is needed to support coding of $e = 0$. Two solutions are in common use: either encode $e + 1$, which is what the δ code originally proposed by Elias does, or prepend a bit that indicates whether $e = 0$, which is the approach used by our modified δ scheme:

$$\delta(2^e + r) = \begin{cases} 0, & \text{if } e = 0, \\ 1\gamma(e)\beta_e(r), & \text{otherwise.} \end{cases} \quad (7)$$

As in Golomb-Rice, we further generalize the δ code to allow the least m bits of the exponent to be encoded in binary, using γ only for the leading bits $\lfloor e/2^m \rfloor$. $\delta(m)$ denotes this generalized code.

We note that our Elias δ code coincides with the URR representation due to Hamada [10], which is perhaps not evident at first glance. Hamada begins with four disjoint intervals, $[-\infty, -1]$, $[-1, 0]$, $[0, +1]$, $[+1, \infty)$ and then recursively partitions them for each bit processed. The split point is given by the lower hyper-4 operator, $2^{2^{p-3}}$, for the unbounded intervals; by the geometric mean for intervals that span more than a factor of two (corresponding to averaging of exponents); and otherwise by the arithmetic mean. An initial sequence of one-bits builds up the exponent, and corresponds to the unary part of the γ code used for encoding the exponent.

6.4 Elias ω

The Elias ω code is a recursive code, where the exponent, e , in $y = 2^e + r$ is itself represented as $e = 2^{e'} + r'$, and so on until $e' = 0$. Decoding ω corresponds to performing repeated exponentiation, a concept also used by the level-index representation of Clenshaw and Olver [3]. They use as base the Euler number, $e = 2.718\dots$, instead of base 2, and unlike ω use the fraction as the final exponent. Their system involves exponentiation, logarithms, and a transcendental base that makes for a rather challenging representation with which to work.

We modify ω in two ways. First, we permute the bits of the original code to make it lexicographically ordered using the following definition:

$$\omega(2^e + r) = \begin{cases} 0, & \text{if } e = 0, \\ 1\omega(e)\beta_e(r), & \text{otherwise.} \end{cases} \quad (8)$$

Second, even four levels of recursion allow for exponents as large as 2^{65536} , so we use a parameter to limit the number of recursive levels, which saves a bit of precision for large numbers.

Our lexicographic $\omega(\infty)$ code is closely related to the Levenshtein code [16], which after removal of the leading bit that distinguishes zero from positive integers is identical to our code.

6.5 Asymptotic Optimality

We now examine the length $|\mathcal{E}(y)|$ needed to encode integers of the form $y = 2^i$, which in binary require $|\beta(2^i)| = i + 1$ bits. In Fig. 2a, we plot the ratio $|\mathcal{E}(y)|/|\beta(y)|$, which for an asymptotically optimal code approaches one as $y \rightarrow \infty$. The excess to one corresponds to the relative cost of encoding the exponent. For Elias γ , half of the bits (plus one terminating zero-bit) encode the exponent, e , in unary, while the remaining half encode the value bits, resulting in a ratio that quickly reaches two. It is easy to show that the ratio approaches $1 + 2^{-es}$ for POSITS. The other two Elias codes are known to be asymptotically optimal, i.e., they approach a ratio of one. It is evident from this plot that POSITS are efficient for small integers.

6.6 Dynamic Range and Hyper Operations

For our universal representations, the exponent length is not fixed, and therefore *realmax* (y_{max}) consists only of exponent bits and is thus a power of two. As a consequence, *realmin* ($y_{min} = y_{max}^{-1}$), is also a power of two due to reciprocal closure of powers of two, and the dynamic range, $\log_2(y_{max}/y_{min})$, is given in whole bits.

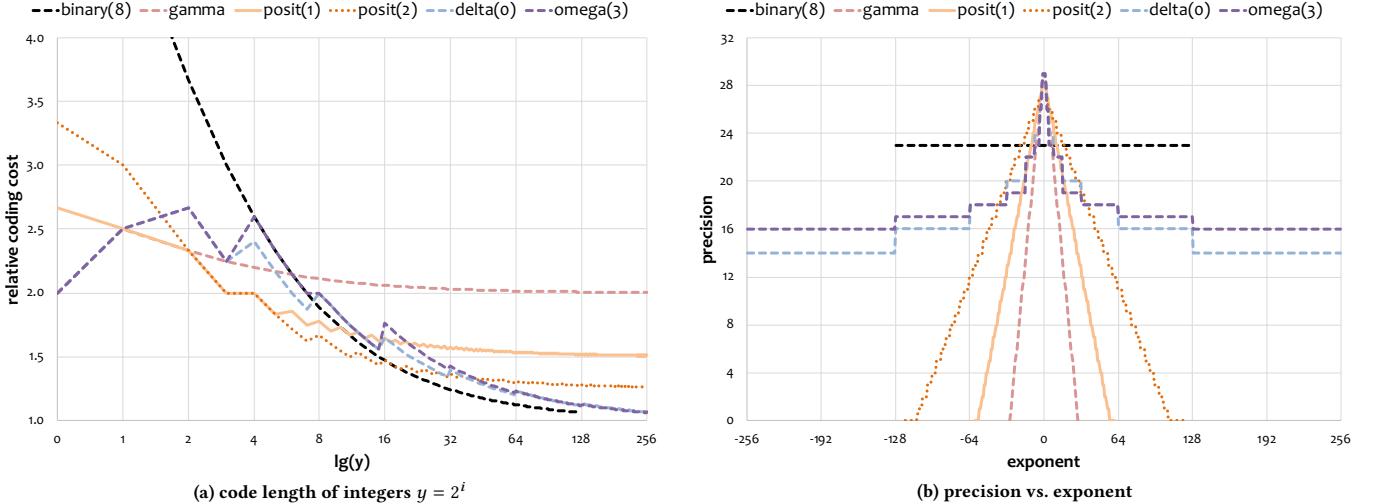


Figure 2: (a) The ratio of code length $|\mathcal{E}(y)|$ to binary length $|\beta(y)| = 1 + \lg y$. *binary(8)* uses a fixed 8-bit signed exponent and can therefore not code values larger than 2^{128} . The other representations are universal and can, given enough precision, code any size integer. (b) Precision (number of stored fraction bits) as a function of exponent, illustrating tapered precision for the variable-length exponent coding schemes.

<i>type</i>	$\lg(y_{\max}(p)/y_{\min}(p))$	<i>realmax sequence</i>	$y_{\max}(p+1)$	<i>operation</i>
unary	$2 \lg(p-1)$	1, 2, 3, 4, 5, 6	$1 + y_{\max}(p)$	hyper-0 (succession)
<i>posit(m)</i>	$2^{m+1}(p-2)$	1, b, b^2, b^3, b^4, b^5	$b \times y_{\max}(p)$	hyper-3 (exponentiation)
Elias γ	$2(p-2)$	1, 2, 4, 8, 16, 32	$2 \times y_{\max}(p)$	hyper-3 (exponentiation)
Elias δ	2^{p-2}	1, 2, 4, 16, 256, 65536	$y_{\max}(p)^2$	lower hyper-4
Elias ω	$2 \times (2 \uparrow\uparrow (p-3))$	1, 2, 4, 16, 65536, 2^{65536}	$2^{y_{\max}(p)}$	upper hyper-4 (tetration)

Table 2: Dynamic range (in bits), realmax sequences, and their recurrence relations for a few p -bit representations.

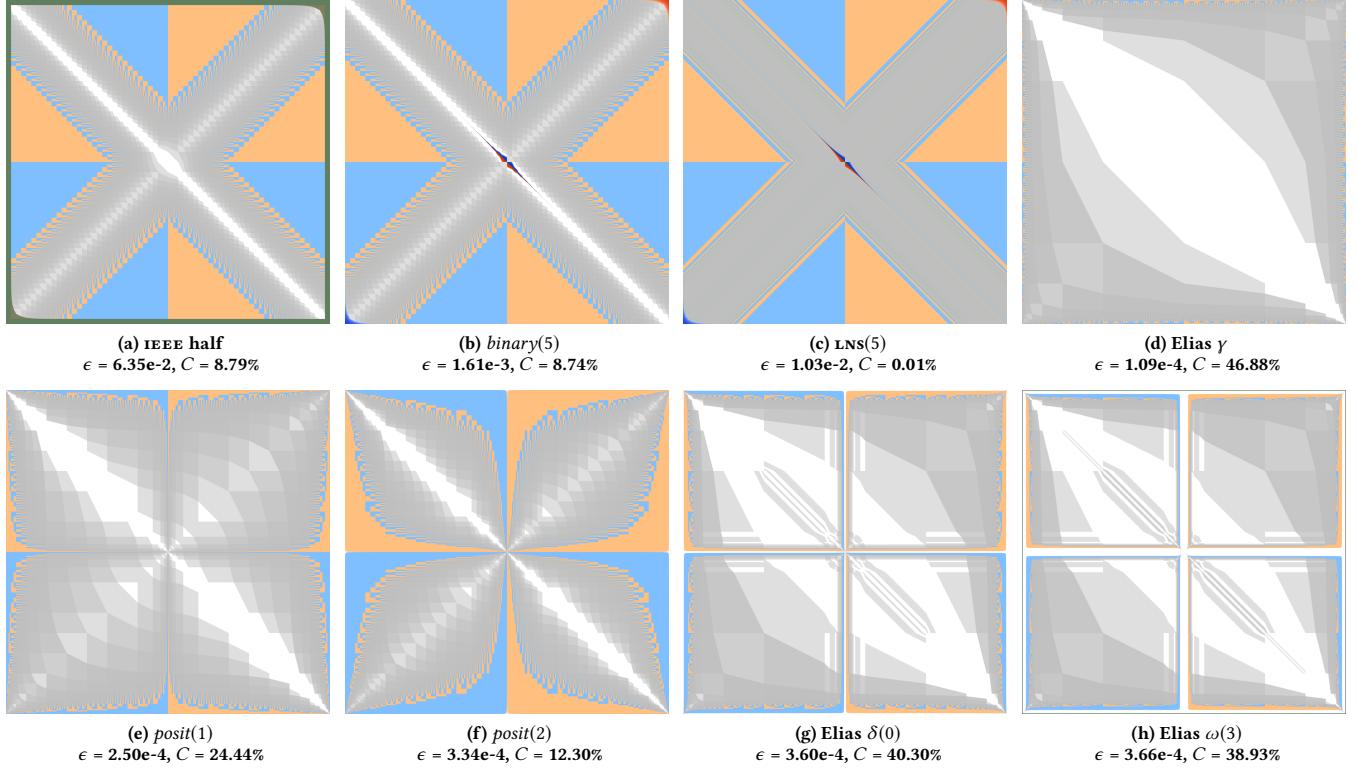
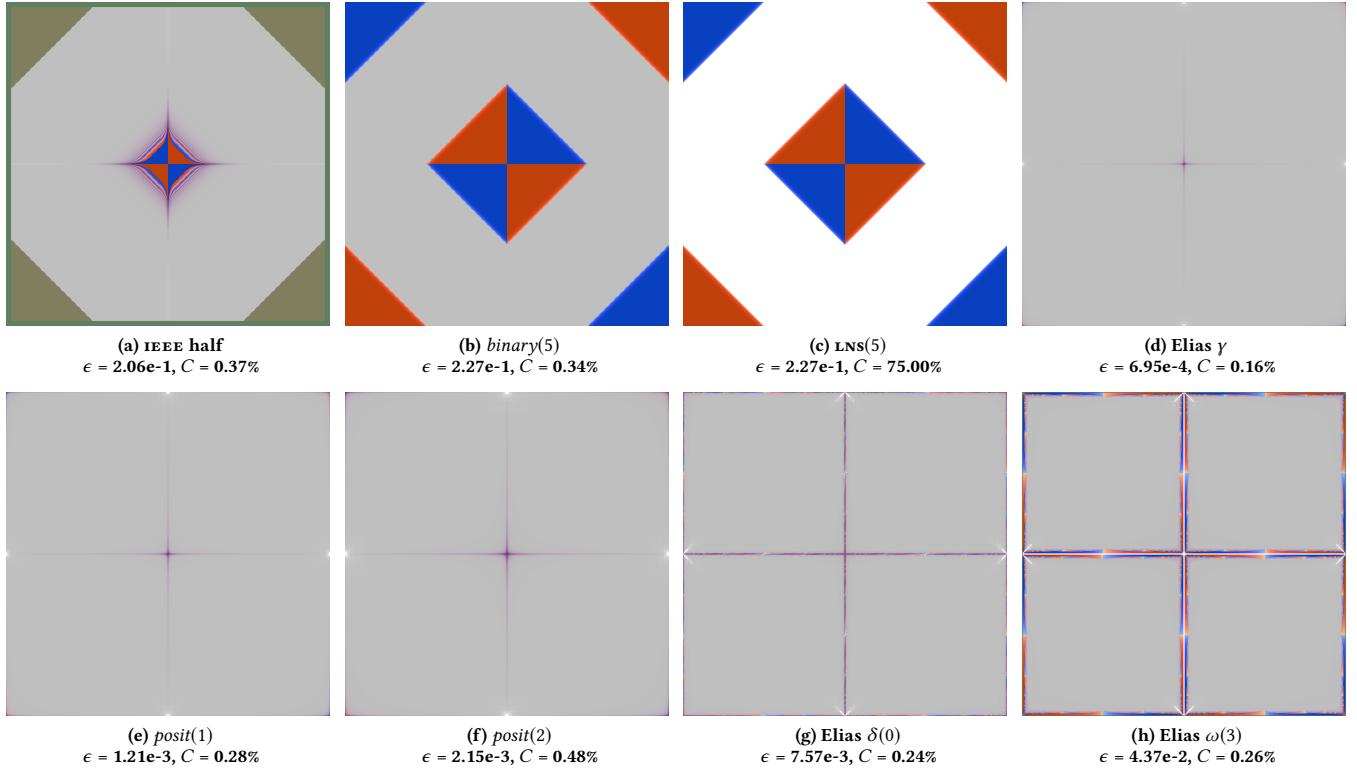
We see in Table 2 the dynamic range and realmax sequences for a few schemes, including signed unary (which does not fit into our framework). For Elias γ , realmax is given by base-2 exponentiation (hyper-3): $y_{\max}(p) = 2^{p-2}$. By extension, $y_{\max}(p) = b^{p-2}$ for *posit(m)*, where $b = 2^{2^m}$ is the base. For Elias ω , y_{\max} increases incredibly fast via *tetration*: $y_{\max}(p) = 2 \uparrow\uparrow (p-2)$, where the tetration operator is given by $b \uparrow\uparrow 0 = 1$ and $b \uparrow\uparrow k = b^{b \uparrow\uparrow (k-1)}$. Elias δ falls in the middle, whose sequence is given by the lower hyper-4 operator, $y_{\max}(p) = 2^{2^{p-3}}$. We note that addition (hyper-1) and multiplication (hyper-2) give schemes that are essentially equivalent to succession (hyper-0).

Compared to representations like δ and ω , the smaller dynamic range of IEEE and POSITS makes it possible to implement exact sums and dot products using very wide (hundreds to thousands of bits) but still manageable hardware accumulators [9, 11] that represent values as integer multiples of y_{\min}^2 . Although the high dynamic range of δ and ω makes such exact arithmetic impractical for extreme values, the same hardware could be used to support exact dot products for any type over the same range supported by IEEE or POSITS. Of course, software solutions exist for computing accurate dot products without the assistance of hardware accumulators [14].

7 EVALUATION

In this section, we evaluate the effectiveness of the number representations discussed so far in a variety of applications ranging from simple arithmetic to a full-fledged physics simulation.

Our framework is implemented in C++ using templates and operator overloading to simplify integration with applications. Because we cannot directly perform arithmetic on our number representations, we distinguish between the *storage type* and *arithmetic type*, the latter being one of the available hardware supported types. In all instances, including when IEEE is the storage type, we use 80-bit extended precision as arithmetic type, which is often implemented as `long double` in C++. Arithmetic and mathematical functions are evaluated using this auxiliary type, with results either directly converted back to the storage type—which we refer to as “eager rounding”—or converted only upon stores—which we refer to as “lazy rounding”. Lazy rounding allows a sequence of intermediate expressions to be evaluated in the usually wider precision supported by the arithmetic type, and is explicitly allowed for by the C and C++ language standards, as given by the setting of `FLOAT_EVAL_METHOD`. To stress the accuracy intrinsic to the number representations, all of our computations use eager rounding.

Figure 3: Additive closure rate (C) and mean relative error (ϵ).Figure 4: Multiplicative closure rate (C) and mean relative error (ϵ).

We note that types like δ and ω have a dynamic range larger than that supported even by IEEE quadruple precision. Very small numbers in these representations must therefore be approximated when converted to the arithmetic type. However, we expect such approximations to be rare and to have minimal impact on accuracy.

In addition to the representations discussed so far, we include *binary*(m), which like IEEE uses a fixed-length m -bit exponent. The primary differences wrt. IEEE are the lack of subnormals and NaNs, the under- and overflow behavior, and an exponent bias that ensures an equal number of sub- and superunitaries. The LNS type is identical but uses the exponential fraction map. Unless otherwise noted, our types use linear sub- and superunitary fraction maps.

7.1 Arithmetic Closure

At a most fundamental level, we wish for arithmetic expressions such as sums and products to be accurately representable in a number system. Gustafson and Yonemoto [9] investigated the arithmetic closure properties of 8-bit types by measuring the ratio, C , of binary operations whose results are exactly representable. We here extend those results to addition and multiplication for 16-bit operands. Our study reveals that operator closure does not paint a complete picture, as types that do not admit perfect closure may still give very small errors in practice. Hence, we also report the relative error, $\epsilon(\tilde{z}, z) = 2(\tilde{z} - z)/(|\tilde{z}| + |z|)$, in the result of each binary operation.

In Fig. 3, we plot the relative errors between the true sum, $z = \mathcal{D}(x) + \mathcal{D}(y)$, and rounded sum, $\tilde{z} = \mathcal{D}(\mathcal{E}(\mathcal{D}(x) + \mathcal{D}(y)))$, as a two-dimensional map whose x and y coordinates correspond to the binary representation of the two operands, with zero in the middle, negative operands on the left and bottom, and positive operands on the right and top.² Large negative errors correspond to saturated blue; large positive errors appear as saturated red; infinities and NaNs are shown in green; zero error is indicated by white. Note that several representations have no error along $x = -y$, which means that these encodings possess exact additive reciprocals. A large number of combinations result in infinities and NaNs for IEEE half precision, while the other encodings do not have this problem. The various Elias and POSITS formats produce remarkably little relative error even near the limits of representation.

In addition to closure rate (C), Fig. 3 also lists mean relative error (ϵ). Evidently Elias γ (aka. *posit*(0)) does quite well with the highest closure and smallest relative error. This is a result of being linear on the subunitaries, which comes at the expense of a limited dynamic range (see Table 1). POSITS with increasing “exponent size” (es) give lower closure rates and higher errors. Elias δ and ω support extremely small values, which when involved in addition with larger operands lead to small relative errors. This property is the source of the white bands through the origin.

Fig. 4 illustrates multiplicative closure and error. Because the product of two p -bit linear fractions in general has $(2p - 1)$ -bit precision, the closure rate for types that use linear fraction maps is very low. Using exponential maps, $\phi(f) = 2^f$, instead (Fig. 4c), the precision required for the product is only $p + 1$, with the extra bit often absorbed in the exponent. In practice, this improved closure rate has virtually no impact on relative error, however.

²Note that the computed error maps are $2^{16} \times 2^{16}$ pixels, but have been filtered and downsampled, resulting in averaging of colors, not errors.

7.2 Matrix Inversion

Among the most common numerical computations are those involving linear algebra, such as solving linear systems of equations. Known challenging numerical problems include inversion of ill-conditioned matrices, such as Hilbert and Vandermonde matrices, for which closed form solutions are known. Both of these types of matrices can be scaled to have rational elements such that their exact inverses have all-integer coefficients given by factorials, binomials, and Stirling numbers. To compute the matrix inverses numerically, we used the Eigen C++ library [6] whose templated design allows us to represent and compute with arbitrary scalar types. We used the dense matrix *inverse()* member function, which uses LU decomposition with complete pivoting to compute the inverse.

Fig. 5 shows the RMS error in the matrix inverse with respect to the exact solution for Hilbert and Vandermonde matrices of various size. Although there is no clear winner, IEEE generally performs poorly and tends to coincide with the *binary* representation in our framework, which like IEEE uses a fixed-length binary encoding (of the same size) of the exponent. The figure also shows that Elias γ (aka. *posit*(0)) does poorly for the larger matrices, which we attribute to its low dynamic range. *posit*(1), on the other hand, does quite well, as do the high dynamic range representations based on Elias δ and ω , which suggests an accuracy advantage to using variable-length exponent representations.

7.3 Eigenvalue Decomposition

We next consider the accuracy of determining the real eigenvalues and eigenvectors of a positive definite symmetric matrix. The $n \times n$ matrix, A_k , is parameterized in size by k , where $n = 2^k$. The eigenbasis of A_k is given by the (sequency ordered) symmetric *Walsh matrix* [20], W_k , whose entries are all ± 1 , with column j having $j - 1$ sign changes. Uniform scaling gives an orthogonal matrix $\hat{W}_k = 2^{-k/2} W_k$. The spectrum of A_k is given by $\lambda_i = 2(n - i) + 1$, where $1 \leq i \leq n$. This prescription gives rise to the recurrence

$$A_0 = (1), \quad A_k = \begin{pmatrix} 2A_{k-1} & J \\ J & 2A_{k-1} \end{pmatrix} = \hat{W}_k \Lambda_k \hat{W}_k, \quad (9)$$

where J is the exchange matrix (the identity matrix with its rows in reverse order). Note that A_k is sparse, and its nonzero entries are all powers of two, which makes it possible to represent A_k exactly using all types in our framework. Moreover, for even k , the eigenvectors \hat{W}_k have components that are integer powers of two, and the eigenvalues Λ_k are small integers. Hence, it is in principle possible to arrive at the exact eigendecomposition, with no error, from an exact representation of A_k . To compute the eigenvalues, we used Eigen’s *SelfAdjointEigenSolver*, which is based on QR iteration using a dense matrix representation.

The root mean square error in eigenvalues is plotted in Fig. 6 for matrices up to $k = 10$ ($n = 1024$). Evidently, IEEE consistently performs poorly compared to representations that use tapered precision. Elias γ does well for small matrices, but quickly suffers from limited dynamic range. *posit*(1) generally does the best of all posits, while the types with largest dynamic range, Elias δ and $\omega(3)$, generally perform the best. For 64-bit precision, these outperform IEEE by around three orders of magnitude.

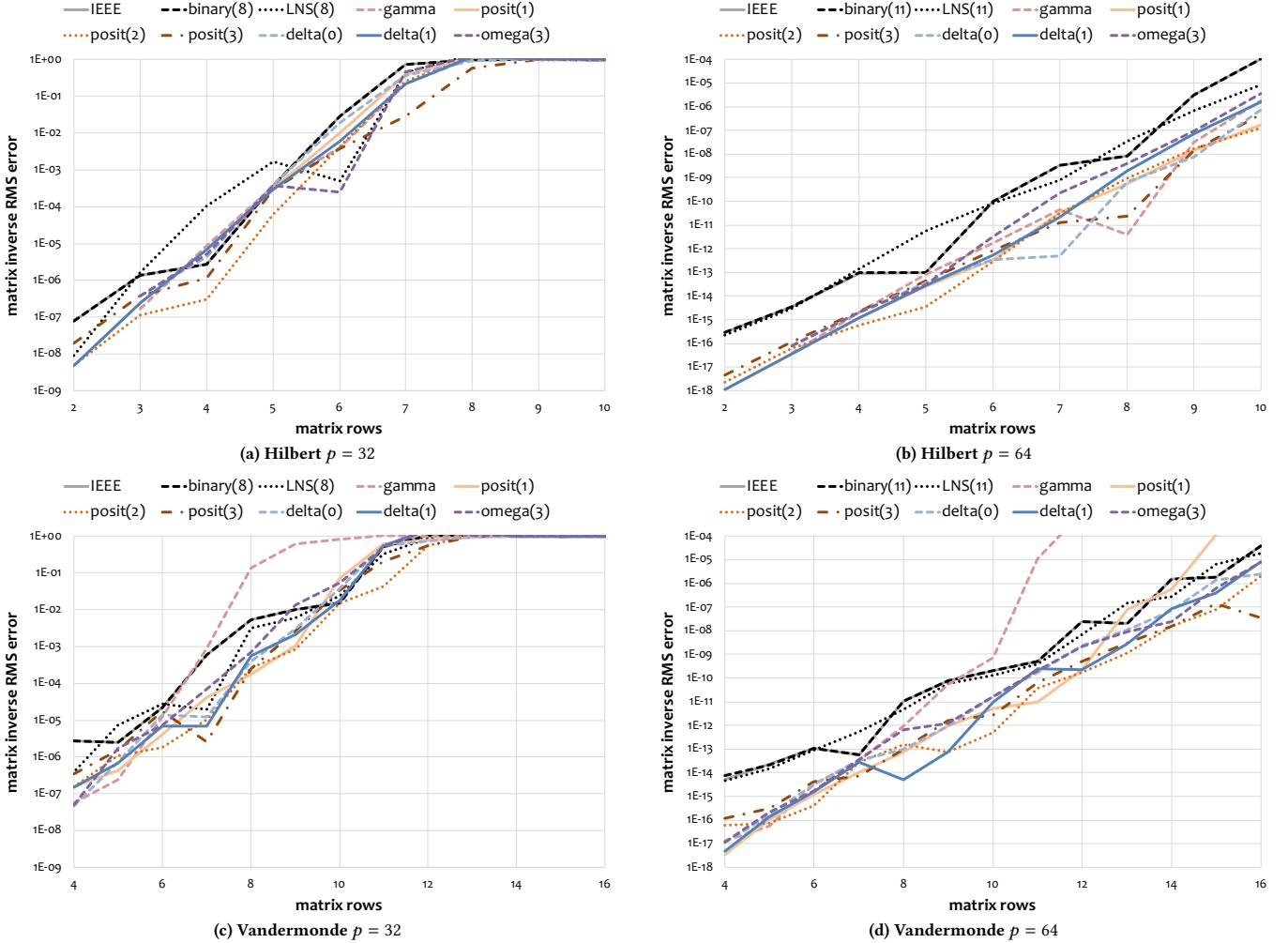


Figure 5: RMS errors in inverses of Hilbert and Vandermonde matrices for different number encodings and precision, $p \in \{32, 64\}$. More often than not, IEEE (which is closely matched by binary) is outperformed by the POSITS and Elias encodings.

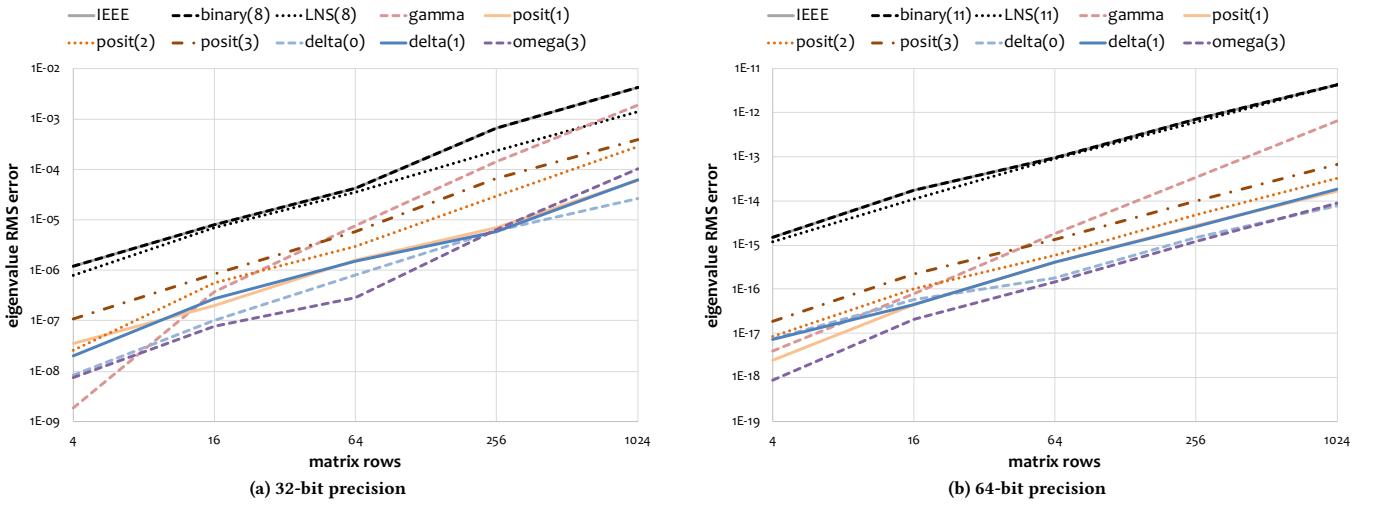


Figure 6: Eigenvalue error as a function of matrix size and number representation. All types but LNS use linear fraction maps.

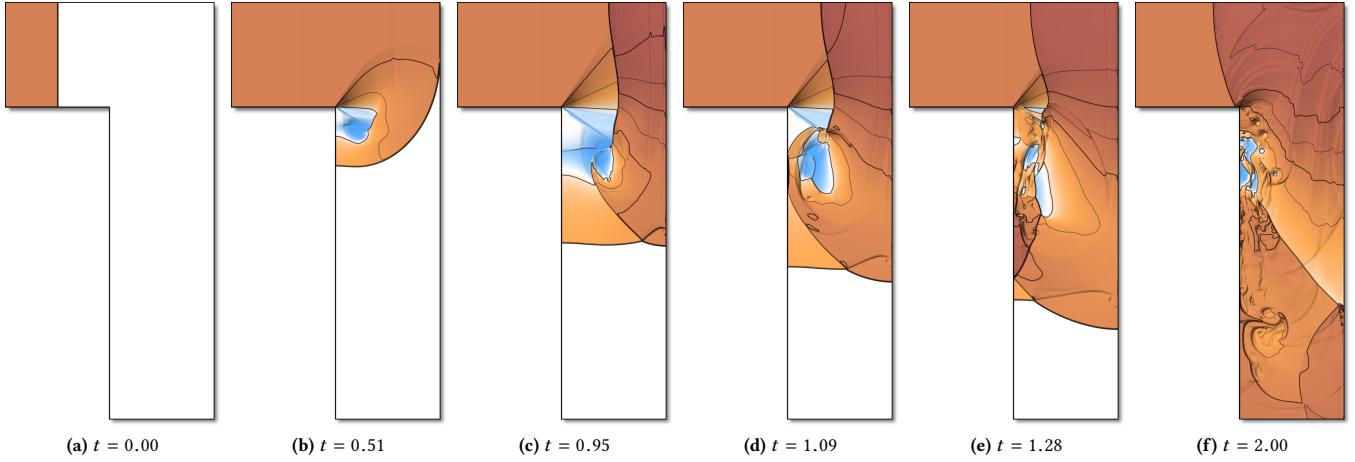


Figure 7: Snapshots in time, t , from the Euler2D mini-application showing the evolution of the density field in an L-shaped chamber. Blue color indicates density lower than the initial density (red) of the shock wave. (a) Initial state. (b) Shock reflects off of far wall. (c) Reflected shock hits vortex. (d) Shock reflects off of near wall. (e) Second reflection hits vortex. (f) Final state.

7.4 Euler2D Mini-Application

As an example application, we use a relatively “simple” code called Euler2D, which implements an explicit, high-resolution Godunov algorithm [5] to solve the Euler system of equations for compressible gas dynamics on an L-shaped domain. Such a solver is simple enough to instrument and understand while providing sufficient complexities, e.g., a nonlinear hyperbolic system with shock formation and minimal dissipation. For context, we briefly describe the system of equations and the numerical algorithm in Appendix A.

The problem solved in the Euler2D code is the propagation of a shock wave in air through an L-shaped conduit. The domain is the union of two rectangles: $[(0, 3), (2, 4)] \cup [(1, 0), (2, 3)]$. At the initial time, a shock, moving with dimensionless speed $M_s = 2.5$ relative to the quiescent state of $(\rho, v_x, v_y, p) = (1, 0, 0, 1)$, is positioned at $x = 0.5$ (Fig. 7a). The inlet flow at $x = 0$ is constant. The code is run with a uniform mesh of size $h = 1/n = 1/256$ using a fixed time step of $\Delta t \approx 2.8e-4$, resulting in roughly 1.3 trillion floating-point operations over the entire run.

As shown in Fig. 7, the shock propagates into the chamber and diffracts around the corner, initiating the shedding of a vortex from the corner. At time $t \approx 0.51$, the initial shock reflects off of the far wall, and the reflected shock propagates back upstream, encountering the vortex around time $t \approx 0.95$. The reflected shock breaks up the vortices shedding off of the corner and reflects again off of the near wall at several times. Eventually, the flow moves down the channel with a propagating sequence of oblique shock waves and a great deal of wave-wave interactions.

A pointwise closed form solution to the Euler2D hyperbolic PDE does not exist. In order to establish ground truth, we used the gcc quadruple precision type `__float128` to compute a high-precision solution. We then computed the root mean square pointwise error in the density field to establish solution accuracy. We expect the RMS error to be dominated by round-off error associated with each numerical type because we compute with fixed discretization parameters, i.e., fixed truncation error.

We plot the pointwise error in the density field over time with respect to the quad precision solution in Figs. 8 to 10. Once again, IEEE and related types do quite poorly in relation to posits and other tapered precision types, which is most evident in the 64-bit precision plot, where *posit(2)* outperforms IEEE by nearly three orders of magnitude. The errors in these plots are surprisingly not monotonic. We see spikes in error that correlate with events such as shock-wall and shock-vortex impact. These spikes are more pronounced in the 64-bit plot.

Fig. 9 highlights for the *posit(1)* exponent encoding scheme how different choices of sub- and superunitary fraction maps can have a significant impact on errors. Enforcing reciprocal closure via a reciprocal subunitary map and linear superunitary map (*posreclin*) has little impact on error, whereas swapping the two maps (*poslinrec*) greatly reduces error. Using the self-conjugate exponent map (*posexp*) also reduces error in general. More work is needed to better understand the underlying causes of these results.

Although no pointwise ground truth exists for Euler2D, the conservative scheme used to solve the Euler equations ensures certain invariants such as conservation of mass. With a constant mass inflow, the total mass increases linearly with time until the shock wave exits the domain around time $t \approx 1.66$. Any deviation from this closed form mass integral is due to accumulated round-off error. In Fig. 11, we plot the absolute error in total mass over time, once again demonstrating the superior accuracy of tapered precision representations over IEEE.

8 CONCLUSION

The increasing relative cost of data movement relative to floating point operations makes this an opportune time to re-evaluate the IEEE floating point representations of the real numbers. In this paper, we have introduced an overarching framework for encoding schemes of the real numbers that contains the IEEE formats as well as several other proposed representations including POSITS. We specifically discuss the choices of exponent encoding, fraction

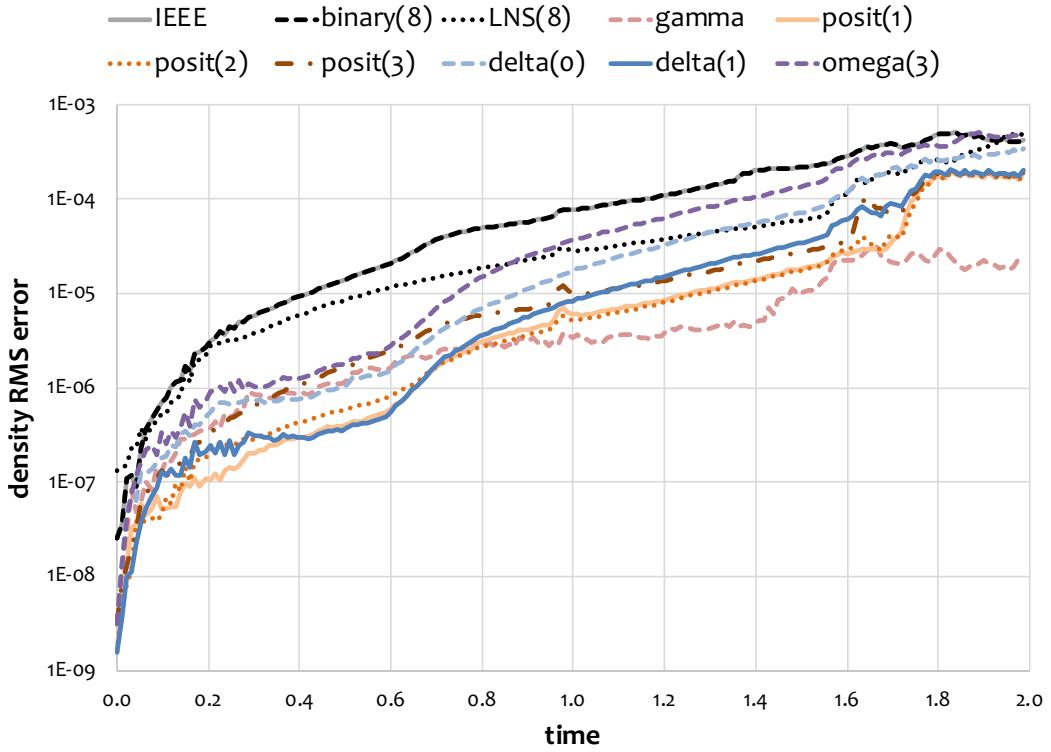


Figure 8: RMS error in the Euler2D density field as a function of simulation time and 32-bit number representation.

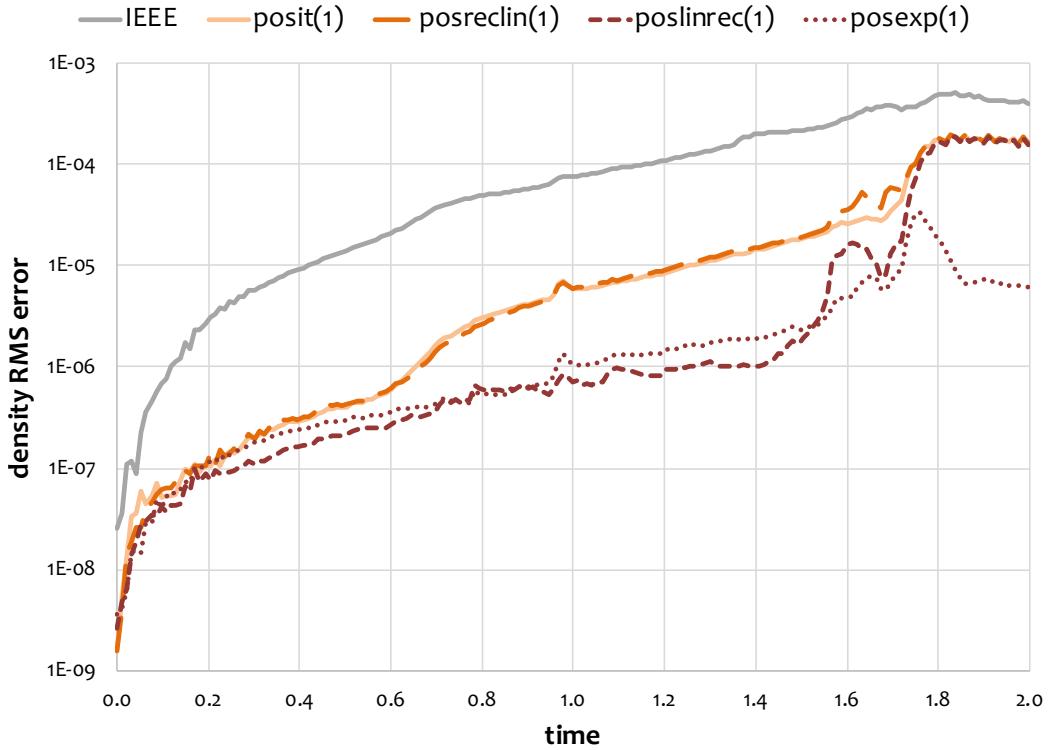


Figure 9: 32-bit *posit(1)* results for combinations of *linear*, *reciprocal*, and *exponential* sub- and superunitary fraction maps.

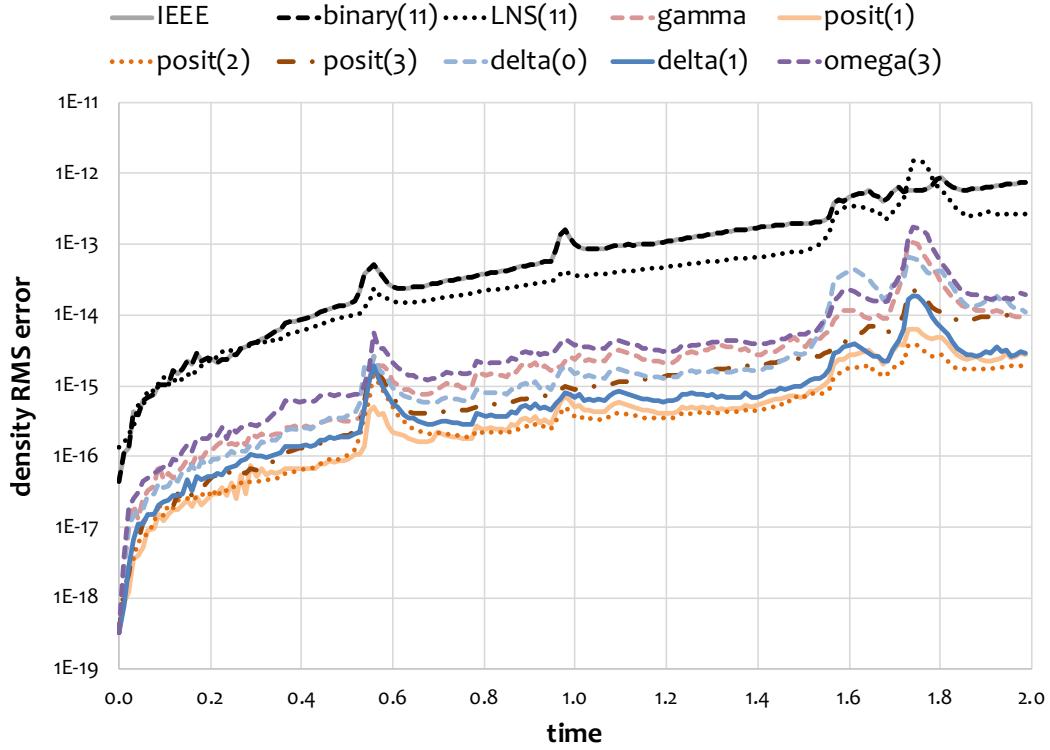


Figure 10: RMS error in the Euler2D density field as a function of simulation time and 64-bit number representation.

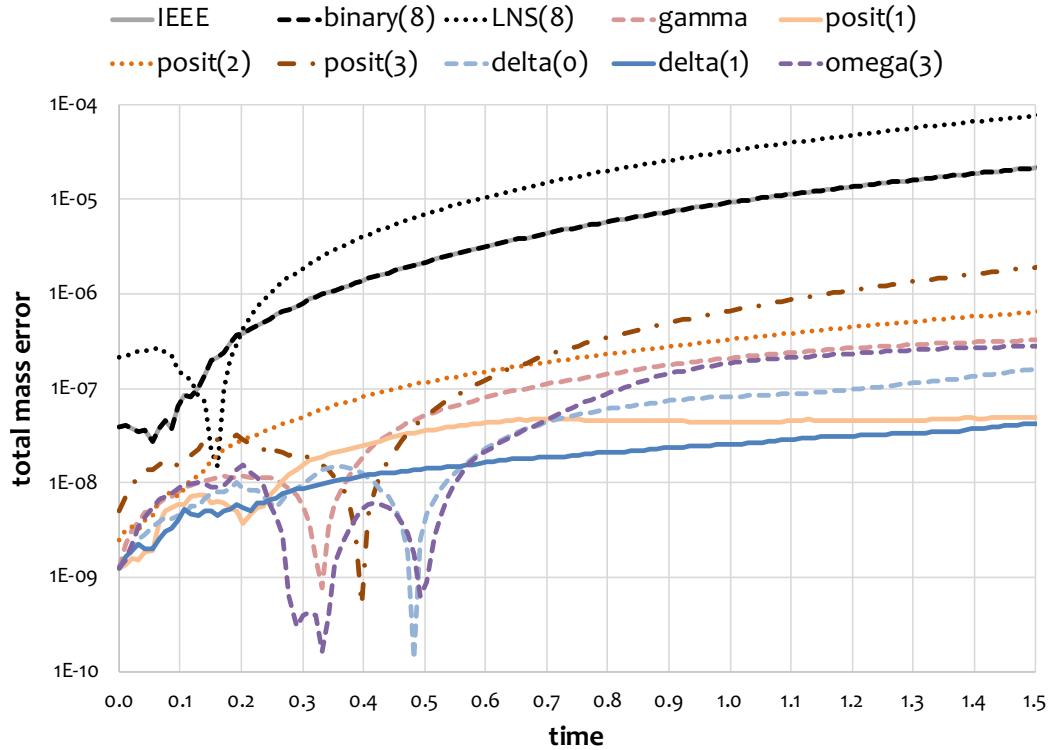


Figure 11: Euler2D error in total mass with respect to the analytical solution for 32-bit types. The sharp downward spikes correspond to zero crossings, where the mass error changes sign.

maps, and the treatment of rounding, overflow, and underflow, and we demonstrate how the encoding framework can be used to ensure desirable properties, such as reciprocal closure, lexicographical ordering, nesting, and tapered precision. The additive and multiplicative closure and approximation errors of eight representations are demonstrated experimentally, and we evaluate their performance in several numerical computations, including matrix inversion, eigenvalue decomposition, and the solution of a nonlinear hyperbolic system of conservation laws. While performance varies depending on the details of the computation, the data suggest that representations that use variable-length exponents fairly consistently outperform the fixed-exponent-length representations, including the IEEE types. Representations such as the POSITS and Elias codes generally have greater arithmetic closure, smaller mean representation error, more efficient representation of infinity and NaNs, and produce lower roundoff error (often by at least an order of magnitude) than the standard IEEE types in use today. While some of these more exotic types are impractical for implementation in hardware, our results provide some evidence that the more practical types should be given more consideration for *in silico* support.

ACKNOWLEDGMENTS

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 and was supported by the LLNL-LDRD Program under Project No. 17-SI-004.

A EULER SYSTEM AND NUMERICAL METHOD DESCRIPTION

The Euler equations of compressible gas dynamics are a nonlinear, hyperbolic system of partial differential equations that express conservation of mass, momentum, and energy. In flux divergence (conservative) form, all hyperbolic systems of conservation laws take the form of

$$\partial_t u + \nabla \cdot f(u) = 0 \quad (10)$$

on domain $\Omega \times t \in (0, T]$, with $u \in \mathbb{R}^{D+2}$ and $f(u) : \mathbb{R}^{D+2} \rightarrow \mathbb{R}^{D \times (D+2)}$ in D dimensions. The vector-valued function, $f(u)$, is called the flux function and represents the flux of conserved quantities into and out of any control volume in the domain Ω .

For the Euler equations in 2D for an ideal gas, the conserved state vector of mass, momentum, and energy and the corresponding fluxes are

$$u = \begin{pmatrix} \rho \\ \rho v_x \\ \rho v_y \\ \rho E \end{pmatrix}, \quad f_x = \begin{pmatrix} \rho v_x \\ \rho v_x^2 + p \\ \rho v_x v_y \\ \rho v_x H \end{pmatrix} \quad \text{and} \quad f_y = \begin{pmatrix} \rho v_y \\ \rho v_x v_y \\ \rho v_y^2 + p \\ \rho v_y H \end{pmatrix}, \quad (11)$$

where ρ is the mass density, (v_x, v_y) is the velocity vector, p is the pressure, $\rho E = (\gamma - 1)^{-1} p + \rho|v|^2/2$ is the total mass density, and $\rho H = \rho E + p$ is the total enthalpy density. The constant γ is the ratio of specific heats, which is 7/5 for a diatomic gas like air.

To approximate solutions to the system (10) numerically, we take a finite volume formulation to map the continuous problem to a discrete domain, i.e., a Cartesian mesh of uniform volumes of size $h = \Lambda/n$. In a finite volume formulation, integrating over the cell

i , where i is a multi-index $i = (i_x, i_y)$, and the time interval from $t = n\Delta t$ to $t = (n + 1)\Delta t$:

$$\bar{u}_i^{n+1} = \bar{u}_i^n - \frac{\Delta t}{h} \sum_{d=1}^D \left[\langle \hat{f}_d \rangle_{i+1/2e^d} - \langle f_d \rangle_{i-1/2e^d} \right], \quad (12)$$

where $\bar{\cdot}$ represents a cell average, $\langle \cdot \rangle$ represents a temporal and face average, and where e^d is the unit vector in the d -th direction. The conservative update equation (12) is exact; numerical approximation enters in the evaluation of the face- and time-averaged fluxes $\langle f_d \rangle_{i+1/2e^d}$.

In the Godunov approach, each interface flux is approximated as the solution to a Riemann problem: a self-similar, nonlinear solution to a piecewise-constant, two-state initial condition. To obtain higher-order (nominally 2nd in smooth regions) in space and time, a predictor step (Hancock's method [19]) in the form of a MUSCL algorithm [18] using nonlinearly-limited slope reconstructions is used. The slope limiter used in the calculations is the double minmod limiter [18], and the Riemann problem at each interface is solved using Roe's approximate Riemann solver [15], which is based on the eigenstructure decomposition of a particular linearization that preserves shock jump properties exactly. Note that the solution method requires the evaluation of multiple square roots and non-integer powers.

REFERENCES

- [1] IEEE standard for floating-point arithmetic, 2008.
- [2] M. Baboulin, A. Buttari, J. Dongarra, J. Kurzak, J. Langou, P. Luszczek, and S. Tomov. Accelerating scientific computations with mixed precision algorithms. *Computer Physics Communications*, 180(12):2526–2533, 2009.
- [3] C. W. Clenshaw and F. W. J. Olver. Beyond floating point. *Journal of the ACM*, 31(2):319–328, 1984.
- [4] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975.
- [5] S. K. Godunov. A difference method for numerical calculation of discontinuous solution of hydrodynamic equations. *Mathematicheskii Sbornik*, 47(3):271–306, 1959.
- [6] G. Guennebaud and B. Jacob. Eigen version 3.2.5. <http://eigen.tuxfamily.org>, 2015.
- [7] J. L. Gustafson. *The End of Error: Unum Computing*. Chapman and Hall, 2015.
- [8] J. L. Gustafson. A radical approach to computation with real numbers. *Supercomputing Frontiers and Innovations*, 3(2):38–53, 2016.
- [9] J. L. Gustafson and I. T. Yonemoto. Beating floating point at its own game: Posit arithmetic. *Supercomputing Frontiers and Innovations*, 4(2):71–86, 2017.
- [10] H. Hamada. URR : Universal representation of real numbers. *New Generation Computing*, 1:205–209, 1983.
- [11] U. Kulisch and V. Snyder. The exact dot product as basic tool for long interval arithmetic. *Computing*, 91:307–313, 2011.
- [12] S. Matsui and M. Iri. An overflow/underflow-free floating-point representation of numbers. *Journal of Information Processing*, 4(3):123–133, 1981.
- [13] R. Morris. Tapered floating point: A new floating-point representation. *IEEE Transactions on Computers*, C-20(12):1578–1579, 1971.
- [14] T. Ogita, S. M. Rump, and S. Oishi. Accurate sum and dot product. *SIAM Journal on Scientific Computing*, 26(6):1955–1988, 2005.
- [15] P. L. Roe. Approximate Riemann solvers, parameter vectors, and difference schemes. *Journal of Computational Physics*, 43(2):357–372, 1981.
- [16] D. Salomon. *Variable-Length Codes for Data Compression*. Springer-Verlag, 2007.
- [17] E. E. Swartzlander Jr. and A. G. Alexopoulos. The sign/logarithm number system. *IEEE Transactions on Computers*, C-24(12):1238–1242, 1975.
- [18] B. van Leer. Towards the ultimate conservative difference scheme. V - A second-order sequel to Godunov's method. *Journal of Computational Physics*, 32:101–136, 1979.
- [19] B. van Leer. Upwind and high-resolution methods for compressible flow: From donor cell to residual-distribution schemes. *Communications in Computational Physics*, 1(2):192–206, 2006.
- [20] R. Wang. *Orthogonal Transforms*. Cambridge University Press, 2012.
- [21] H. Yokoo. Overflow/underflow-free floating-point number representations with self-delimiting variable-length exponent field. *IEEE Transactions on Computers*, 41(8):1033–1039, 1992.