MultiPosits: Universal Coding of $\mathbb{R}^n$

Peter Lindstrom(✉)

Lawrence Livermore National Laboratory, Livermore, CA 94550, USA
pl@llnl.gov

Abstract. Recently proposed real-number representations like Posits and Elias codes provide attractive alternatives to IEEE floating point for representing real numbers in science and engineering applications. Many of these applications represent fields on structured grids that exhibit smoothness, where adjacent scalar values are similar and often accessed together in stencil or vector computations. This similarity results in redundancy in representation, where several leading bits in the representation of adjacent values are shared.

We propose a generalization of scalar "universal codes" to small, multidimensional blocks of values that exploit their similarity and underlying dimensionality. Drawing upon ideas from multimedia and floating-point compression, our approach combines a decorrelating transform with adaptive, error-optimal interleaving of coefficient bits, which allows increasing accuracy per bit stored by orders of magnitude. Our solution accommodates both a fixed-length representation of blocks—facilitating random access—and variable-length storage to within a user-prescribed tolerance—e.g., for I/O, communication, and streaming computations. Our approach generalizes universal coding of the reals to vectors and tensors, and is straightforward to implement for several known number systems by extending a previously published framework for universal coding based on simple refinement rules.

Keywords: Number representations · Floating point · Universal coding · Data compression · Decorrelating transform · Vector quantization

1 Introduction

As data movement and storage have come to dominate the power and performance landscape in high-performance computing, there has been a recent push to investigate new real number representations that are more economical than the ubiquitous IEEE 754 floating-point format [1]. Example proposals include BrainFloats [11] and TensorFloats [2], which make a different tradeoff between the number of exponent and significand bits than IEEE 754. More significant departures from IEEE 754 include include Unums [8], Posits [9], URRs [10], and Elias codes [20], some of which generalize universal codes originally developed for positive integers [7] to the reals. Many of these representations can be synthesized using number system frameworks that allow experimenting with alternative number representations [18–20,23,25], and several examples have demonstrated the benefits of such representations in numerical applications in terms of improving the accuracy per bit stored [14,16,20]. Common to the universal number representations is the notion of *tapered accuracy* [21], where commonly occurring numbers near one are represented more accurately than rare numbers that are extremely small or large in magnitude. This is achieved by allocating fewer bits to represent the exponent in favor of retaining more bits for the significand (or fraction) for numbers near ± 1.

Many science and engineering applications model the physical world as mostly continuous scalar fields, such as temperature and pressure, that are sampled discretely onto uniform Cartesian grids and are represented as multidimensional arrays of reals. In these applications, values at adjacent grid points tend to exhibit significant correlation, which manifests as shared leading bit patterns in their number representation. The conventional approach of representing arrays as independent scalars wastes precious bits on such redundant information, resulting in a larger than necessary memory footprint and associated costs in moving data through the memory hierarchy. Recent efforts have attempted to remove the redundancy using variations on block-floating-point representation [12], by partitioning arrays into small blocks of correlated scalars and eliminating shared information [3,16,17]. Whether explicit or not, such methods substitute the *scalar quantization* of reals implied by the number representation with a *vector quantization* step, where each fixed-length codeword encodes a whole block of numbers (unraveled as a vector). Current block-floating-point representations are modeled on IEEE 754—they use a fixed-length exponent common to the block and a set of significands (or coefficients) that are scaled by the common exponent.

In this paper, we propose an alternative block-based representation that builds upon the ideas shared by universal number representations, which use a variable-length encoding of the exponent and that—given sufficient precision—can represent *any* real. This is unlike IEEE 754 and block-floating-point representations that utilize a fixed-length exponent, which places a fixed limit on the smallest and largest numbers representable regardless of precision. Our new representation further reduces redundancy by performing a decorrelating linear transformation, which in effect replaces leading bits shared among values in a block with strings of leading zeros (or ones) that can be efficiently encoded. We demonstrate the accuracy benefits of a tapered number system for blocks of reals combined with a decorrelation step that eliminates shared leading bits in order to represent more trailing bits of significands. Like most other floating-point-like representations, we may truncate the binary representation at any point—a step analogous to *rounding*—to achieve a fixed-length representation of each block that facilitates random access (at block granularity). We may alternatively truncate the representation when it satisfies an error tolerance, resulting in variable-length records. In applications where the data is accessed sequentially, e.g., in I/O and streaming computations, such variable-length codes ensure a uniform level of error and avoid an excess in precision when it is not needed or when the trailing bits are already contaminated with error, e.g., due to roundoff, discretization, approximate solvers, sensor noise, etc. [6,15,28].

Our framework generalizes universal codes for $\mathbb{R}$ to $\mathbb{R}^n$ without being tied to any particular number representation. Rather, we allow any universal code for the reals to be used and show how to optimally interleave the bits representing a collection of scalars from such a code to represent decorrelated real-valued vectors or tensors. In this paper, we present results of extending Posits and demonstrate their utility in multiple applications.

2 Preliminaries

One of our earlier insights [18] is that most real number representations are fully described by a cumulative distribution function (CDF), $F(x)$, with associated probability density, $f(x)$. $F(x)$ maps a real, $x \in \mathbb{R}$, to the interval $(0, 1)$, with $F(-\infty) = 0$ and $F(+\infty) = 1$.[1] The binary bit string $0.b_1 b_2 \ldots b_p$ thus represents $F(x) = \sum_{i=1}^{p} b_i 2^{-i} \in [0, 1)$ using p bits of precision.[2] In other words, for finite p, $F(x)$ is rounded to the nearest multiple of 2^{-p}.[3] This rounding may also be viewed as linear scalar quantization with step size $\varDelta = 2^{-p}$.

A *universal code* for the reals also satisfies the following properties:

1. $f(x) > 0 \ \forall x \in \mathbb{R}$, which ensures that every real can be represented uniquely. Because $f(x) = 0$ for $x >$ `FLT_MAX`, IEEE 754 is not a universal code.
2. $|x| \leq |y| \iff f(x) \geq f(y)$. In other words, $f(x)$ decreases monotonically away from zero, with larger $|x|$ requiring longer codewords. As a corollary, $f(x) = f(-x)$.
3. $\lim_{x\to\infty} \frac{-\log_2 f(x)}{\log_2 x}$ is finite. This ensures that $-\log_2 f(x)$, which governs code length, does not increase too rapidly. Like universal codes for integers, this property disqualifies representations like the unary code, whose length is arbitrarily longer than binary code.

These properties essentially generalize similar properties required for universal integer codes; see [7].

Another key insight from [18] is that universal codes may be expressed as two functions: a generator function, g, that is used in unbounded search to bracket $x \geq 1$ or $x^{-1} \geq 1$, and a refinement function, $r(x_{\min}, x_{\max})$, that is used in binary search to increase the precision by narrowing the interval containing x. For Posits, $g(x) = \beta x$, where $\beta = 2^{2^m}$ is the *base* (also called *useed* in [9]); see [19]. In the 2022 Posit standard, $m = 2$ for Posits regardless of precision; thus, $g(x) = 16x$. The generalization of the Elias gamma code uses

[1] In Posits and [20], $-\infty$ and $+\infty$ map to the same point, called NaR, and are represented as $F(-\infty) \bmod 1 = F(+\infty) \bmod 1 = 0$.

[2] Using two's complement representation, it is common to translate $(0, 1)$ to $(-\frac{1}{2}, \frac{1}{2})$ by negating the $b_1 2^{-1}$ term such that the bit string $0.000\ldots$ represents $x = 0$.

[3] Special rounding modes may be used so that finite numbers are not rounded to the interval endpoints $\{0, 1\}$, which represent $\pm\infty$.

$g(x) = 2x$, and is thus a particular instance of Posits with $m = 0$, while Elias delta uses $g(x) = 2x^2$; see [18]. These number systems use as refinement function the arithmetic mean of the interval endpoints, $r(a, b) = (a + b)/2$, unless a and b differ in magnitude by more than 2, in which case the function returns the geometric mean, $r(a, b) = \sqrt{ab}$. Thus, a whole number system may be designed by specifying two usually very simple functions. For simplicity of notation, we will use the refinement function, r, to split intervals even when they are unbounded. We use the convention $r(-\infty, \infty) = 0$, $r(0, \infty) = 1$, $r(a, \infty) = g(a)$, $r(0, b) = r(b^{-1}, \infty)^{-1}$, $r(-b, -a) = -r(a, b)$; see [18]. Each codeword bit thus determines in which subinterval, $[a, s)$ or $[s, b)$, x lies, with $s = r(a, b)$.

A naïve way of extending universal codes from $\mathbb{R}$ to $\mathbb{R}^n$ would be to interleave the bits from the n independent codewords in round-robin fashion. Not only does this often result in significant redundancy of identical bits, but such round-robin interleaving is also suboptimal from an accuracy standpoint when absolute rather than relative error matters. As we shall see later, one can significantly improve accuracy by instead interleaving the bits in a data-dependent order.

3 Universal Coding of Vectors

Before describing our coding algorithm in detail, we first outline its key steps. Given a d-dimensional scalar array, we first partition it into blocks. As in the zfp [17] representation, we have chosen our blocks to be of length 4 in each dimension. This block size has proven large enough to expose sufficient correlation, yet small enough to provide sufficiently fine granularity and allow a fast and simple implementation of decorrelation. Furthermore, a block size that is a power of two simplifies indexing via bitwise shifting and masking instead of requiring integer division. Hence, a d-dimensional block consists of $n = 4^d$ scalars. If array dimensions are not multiples of four, we pad blocks as necessary; see [6].

Each block is then encoded separately, either using a fixed number of bits, np, where p is the per-scalar precision, or by emitting only as many bits as needed to satisfy an absolute error tolerance, ϵ. The encoding step begins by decorrelating the block (Sect. 3.1) using a linear transformation. The goal of this step is to eliminate any correlation between values and to *sparsify* the block such that common leading bits in a scalar representation of the values are (usually) replaced with leading zero-bits, with many transform coefficients having small magnitude. We use the same transformation as in zfp, which can be implemented very efficiently using addition, subtraction, and multiplication by $\frac{1}{2}$ or by 2. As in zfp, the resulting set of n transform coefficients is then reordered by *total sequency* using a fixed permutation $\pi(x)$ (Sect. 3.2).[4] Finally, the reordered coefficients are encoded by emitting one bit at a time from the universal scalar code of one of the n coefficients selected in each iteration (Sect. 3.3). We leave

[4] Sequency denotes the number of zero-crossings of a discrete 1D function. Total sequency denotes the sum of per-component zero-crossings of basis functions in a tensor product basis, and is analogous to total degree of a multivariate polynomial.

```
ENCODE($x = (x_1, \ldots, x_n), p, \epsilon$)
 1. $x \leftarrow Qx$                              // §3.1: decorrelate $x$
 2. $x \leftarrow \pi(x)$                           // §3.2: sort $x$ on sequency
 3. $R \leftarrow (-\infty, +\infty)^n$                 // initialize hyper-rectangle
 4. for $k = 1, \ldots, np$                        // §3.3: encode $x$ in up to $np$ bits
 5.     $i \leftarrow \mathrm{argmax}_j |R_j|$              // identify widest dimension $i$
 6.     $I \leftarrow R_i$
 7.     if $|I| \le \left(\frac{4}{15}\right)^d \epsilon$            // is error tolerance met?
 8.         terminate
 9.     $s \leftarrow r(I_{\min}, I_{\max})$                // compute interval split point
10.     if $x_i < s$                             // is $x_i$ below the split point $s$?
11.         output(0)
12.         $I_{\max} \leftarrow s$
13.     else
14.         output(1)
15.         $I_{\min} \leftarrow s$
16.     $R_i \leftarrow I$                            // update narrowed interval
```

Listing 1: Universal coding algorithm for n-dimensional ($n = 4^d$) vector x using precision, p, and error tolerance, ϵ. If unspecified, we assume $\epsilon = 0$. $|I|$ denotes the interval width $I_{\max} - I_{\min}$.

the choice of universal scalar code open; in our algorithm, this choice is determined solely by the interval refinement function, r. The coefficient chosen in each iteration of the coding algorithm is data-dependent and done in a manner so as to minimize the upper bound on the L_∞ error in the reconstructed block. The decoding algorithm performs the same sequence of steps but using their natural inverses and in reverse order.

Listing 1 gives pseudocode for encoding a single d-dimensional block of $n = 4^d$ real scalars. Any arithmetic performed and state variables used must have sufficient precision, e.g., IEEE double precision. We proceed by describing each step of the encoding algorithm in more detail and conclude with some implementation details.

3.1 Decorrelation

When a (mostly) continuous function is sampled onto a sufficiently fine uniform grid, values at adjacent grid points tend to be significantly correlated. Such discrete data is said to be "smooth" or to exhibit autocorrelation. Autocorrelated data is undesirable because it introduces overhead in the representation, as exponents and leading significand bits of adjacent values tend to agree. The process of removing correlation is called *decorrelation*, which can be achieved using a linear transformation (i.e., by a matrix-vector product).

Consider a partitioning of a d-dimensional array into equal-sized blocks of $n = 4^d$ values each. Then each of the n "positions" within a block may be

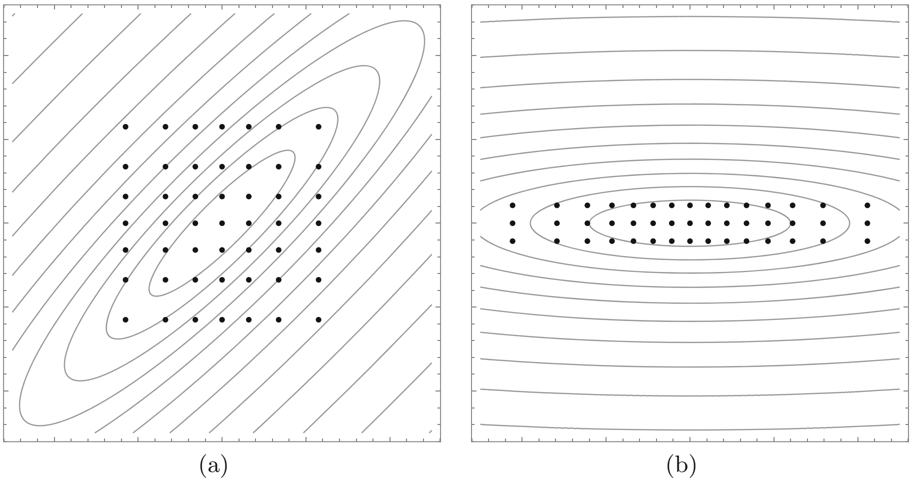


Fig. 1. Decorrelation of two correlated and identically distributed variables (X, Y) with variance $\sigma^2 = 1$ and covariance $\rho\sigma^2 = 0.9$. In this simple example, the decorrelating transform is merely a 45-degree rotation. These plots show contours of the joint probability density before and after decorrelation as well as the set of representable vectors (indicated by dots at regular quantiles) using 6 total bits of precision. In (a), vectors are represented using 3 bits per component. In (b), 4 bits are used for the X component (with variance $1 + \rho = 1.9$), while 2 bits are used for the Y component (with variance $1 - \rho = 0.1$). Notice the denser sampling and improved fit to the density in (b). Decorrelation here removes the covariance between the two variables.

associated with a random variable, X_i, with the values from the many blocks constituting random variates from the n random variables. The spatial correlation among the $\{X_i\}$ is determined by their variance and covariance. The covariance—and therefore correlation—is eliminated by performing a transformation (or change of basis) using a particular orthogonal $n \times n$ matrix, Q. For perfect decorrelation, this matrix Q is given by the eigenvectors of the covariance matrix, and the associated optimal transform is called the Karhunen-Loève Transform (KLT) [27]. The KLT is data-dependent and requires a complete analysis of the data, which is impractical in applications where the data evolves over time, as in PDE solvers. Instead, it is common to use a fixed transform such as the discrete cosine transform (DCT) employed in JPEG image compression [26], the Walsh-Hadamard transform [27], or the Gram orthogonal polynomial basis, which is the foundation for the transform used in the current version of ZFP [6] as well as in our encoding scheme. Such suboptimal transforms do not entirely eliminate correlation, though in practice they tend to be very effective.

To visualize the process of decorrelation, Fig. 1 shows a cartoon illustration using two correlated and identically distributed random variables (X, Y), representing the relationship between pairs of adjacent grid points. In Fig. 1(a),

$X, Y \sim \mathcal{N}(0,1)$ are unit Gaussians with covariance matrix

$$\Sigma = Q^T D Q = \begin{pmatrix} 1 & \rho \\ \rho & 1 \end{pmatrix}, \tag{1}$$

where

$$Q = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix} \qquad D = \begin{pmatrix} 1+\rho & 0 \\ 0 & 1-\rho \end{pmatrix} \tag{2}$$

represent the eigendecomposition of Σ. (X, Y) are decorrelated by the linear transformation $\begin{pmatrix} X' & Y' \end{pmatrix}^T = Q \begin{pmatrix} X & Y \end{pmatrix}^T = \frac{1}{\sqrt{2}} \begin{pmatrix} X+Y & Y-X \end{pmatrix}^T$, leaving D as the diagonal covariance matrix, i.e., covariance and therefore correlation have been eliminated. Furthermore, whereas X and Y have identical variance, $\sigma^2(X') = 1 + \rho$ is far greater than $\sigma^2(Y') = 1 - \rho$ when ρ is close to one, as is often the case. Consequently, we expect random variates from Y' to be small in magnitude relative to X', which allows representing Y' at reduced precision relative to X' without adverse impact on accuracy. In other words, in a Posit or other universal coding scheme of (X', Y') as independent components, with $|X'| \gg |Y'|$, the leading significand bits of X' carry more importance than the leading significand bits of Y', which have smaller place value.

Our approach to universal encoding of $\mathbb{R}^n$ is to make use of the independent scalar universal codes of the vector components (e.g., X' and Y' above), but to interleave bits from those codes by order of importance, i.e., by impact on error. This allows for fixed-precision representation of vectors from $\mathbb{R}^n$ (in our example, $n = 2$) as a fixed-length prefix of the full-precision bit string of concatenated bits. We may also use a variable-precision representation, where we keep all bits up to some minimum place value $\epsilon = 2^e$, where ϵ represents an absolute error tolerance.

Decorrelating Transform. The example above shows a decorrelating transform for pairs of values. In practice, in numerical applications where physical fields are represented (e.g., temperature on a 3D grid), values vary slowly and smoothly, and correlations extend beyond just immediate neighbors. This observation is the basis for block compression schemes such as jpeg image compression [26] and zfp floating-point compression [17], where larger d-dimensional blocks of values are decorrelated together, e.g., 8×8 in jpeg and $4 \times 4 \times 4$ in 3D zfp. Due to its success in science applications, we chose to base our universal encoding scheme on the zfp framework, which relies on a fast transform that approximates the discrete cosine transform used in jpeg:

$$Q = \frac{1}{16} \begin{pmatrix} 4 & 4 & 4 & 4 \\ 5 & 1 & -1 & -5 \\ -4 & 4 & 4 & -4 \\ -2 & 6 & -6 & 2 \end{pmatrix} \qquad Q^{-1} = \frac{1}{4} \begin{pmatrix} 4 & 6 & -4 & 1 \\ 4 & 2 & 4 & 5 \\ 4 & -2 & 4 & -5 \\ 4 & -6 & -4 & 1 \end{pmatrix} \tag{3}$$

This transform, which is slightly non-orthogonal, can be implemented very efficiently in place using *lifting steps* [5] and involves only 5 additions, 5 subtractions, and 6 multiplications by $\frac{1}{2}$ or 2, compared to 12 additions and 16 multiplications for a standard matrix-vector multiplication; see [6] for details. Unlike in ZFP, we perform arithmetic in floating point, e.g., using IEEE double precision or, if desired, MPFR arbitrary-precision arithmetic. Note how $\|Q\|_\infty = 1$, which ensures that there is no range expansion during application of Q. Conversely, $\|Q^{-1}\|_\infty = \frac{15}{4}$. Thus, rounding errors in the transform coefficients may expand by as much as $\left(\frac{15}{4}\right)^d$ in d dimensions, which is accounted for in Listing 1, line 7.

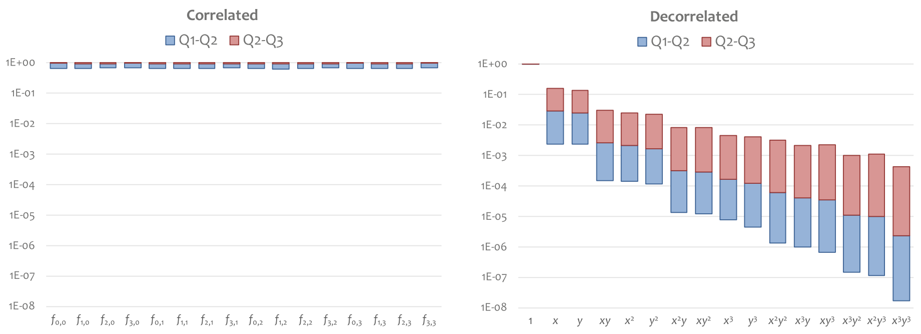


Fig. 2. Distributions as interquartile range of the magnitude relative to maximum for each of the 4×4 values $f_{i,j}$ (left) and transform coefficients for basis functions $O(x^i y^j)$ (right). The distributions represent four million randomly sampled 2D blocks from 32 fields from seven different data sources. Notice the effectiveness of decorrelation in compacting the signal energy into the low-order modes (vertical axis is logarithmic).

As in ZFP, we make use of a tensor product basis in $d > 1$ dimensions, where we apply the transform along each dimension of a block to decorrelate its 4^d values. Following decorrelation of a block, we proceed by encoding its transform coefficients, most of which tend to be very small in magnitude. Each block is thus transformed and encoded independent of other blocks, allowing access to arrays at block granularity.

Figure 2 illustrates the benefits of decorrelation by plotting the distributions of values from 2D blocks before and after decorrelation. These distributions represent the magnitude of values in each block relative to the block's largest value, i.e., $|f_{i,j}| / \max_{0 \le i,j \le 3} |f_{i,j}|$. The plots show how essentially identically distributed values $f_{i,j}$ are sparsified by decorrelation using the ZFP tensor product basis $Q \otimes Q$. Each basis vector approximates a regularly sampled orthogonal Gram polynomial, with coefficients for high-order polynomial terms being several orders of magnitude smaller than the constant and linear terms. This implies that the data within each block is well approximated using only a few low-order terms. The basis functions $O(x^i y^j)$ have been ordered by total degree $i + j$, then by $i^2 + j^2$, resulting in a nearly monotonic decrease in each of the quartiles ($Q1 = 25\%$, $Q2 = 50\% =$ median, $Q3 = 75\%$).

Using a Taylor expansion of the continuous scalar field being encoded, one can show that the magnitude of the ith transform coefficient, f'_i, in $d = 1$

dimension varies as $O(h^i)$, with h being the grid spacing. The extension to higher dimensions is straightforward, e.g., $|f'_{i,j,k}| = O\big((\Delta x)^i(\Delta y)^j(\Delta z)^k\big)$. Thus, $|f'_{i,j,k}| = O(h^{i+j+k})$ when $\Delta x = \Delta y = \Delta z = h$. This further explains why total sequency ordering by $i + j + k$ results in a monotonic decrease in magnitude as $h \to 0$. Moreover, this has implications on the variance of transform coefficients, which could be exploited if different scalar coding schemes were used for the transform coefficients.

3.2 Reordering

One observation from Fig. 2 is that the decorrelating transform results in transform coefficients whose distributions differ widely. In particular, coefficients corresponding to basis functions with high total sequency (shown toward the right in this figure) tend to be close to zero. Thus, the encoding of bits from those coefficients tend only to confirm that their intervals should be further narrowed toward zero. Conversely, low-sequency coefficients tend to carry most of the information, and hence their bits (within a given bit plane) tend to be more valuable. Thus, when tiebreaking decisions have to be made in terms of ordering coefficients within a single bit plane, we prefer to encode bits from low-sequency coefficients first. This is accomplished by reordering the coefficients by total sequency, as is also done in ZFP and JPEG. That is, a coefficient $f'_{i,j,k}$ in a 3D block has total sequency $i + j + k$. We use as secondary sort key $i^2 + j^2 + k^2$, e.g., a trilinear term ($i = j = k = 1$) precedes a cubic one ($i = 3, j = k = 0$), and break any remaining ties arbitrarily. Note that this ordering tends to list coefficients roughly by decreasing magnitude.

3.3 Encoding

At this point, we have a set of decorrelated values roughly ordered by decreasing magnitude. Because they are no longer correlated, their joint probability density (in the idealized case) is given by the product of marginal densities:

$$f(X_1, X_2, \ldots, X_n) = f_1(X_1)f_2(X_2)\cdots f_n(X_n). \tag{4}$$

Due to this independence, vector quantization is reduced to independent scalar quantization, where the quantization results in an n-dimensional "grid" onto which the vector $X = (X_1, \ldots, X_n)$ is quantized. Note that such a grid need not have the same number of grid points (as implied by the per-variable precision) along each dimension.

Though the X_i are independent, note that they are not identically distributed, as evidenced by Fig. 2. Ideally, we would design a separate code optimal for each such distribution, however this brings several challenges:

– The actual distributions are data or application dependent. While some efforts have been made to optimize number systems for given data distributions [13], such approaches become impractical in computations like PDE solvers, where the distributions are not known a priori.

– Even if the data distributions were known, finding corresponding error minimizing codes is an open problem. Currently, L_2 optimal codes are known for only a few distributions, most notably the Laplace distribution [22].
– Assuming these two prior challenges can be addressed, the CDF for an error optimal code would likely not be expressible in closed form or would involve nontrivial math functions that would be prohibitively expensive to evaluate. For best performance, we prefer CDFs that are linear over binades.

Faced with these challenges, we take a different approach by making use of "general purpose" universal number representations like Posits and by optimizing the order in which bits from the X_i are interleaved to minimize the L_∞ error norm. While representations like Posits are parameterized (on "exponent size"), which would allow parameter selection tailored to each random variable X_i, we do not pursue such an approach here but believe it would be a fruitful avenue for future work.

Given a codeword c comprised of interleaved bits, c can be thought of as encoding the path taken when traversing a k-d tree that recursively partitions the n-dimensional space—a hyper-rectangle—in halves using a sequence of binary cuts, each along one of the n axes. To minimize the L_∞ error norm, we should always cut the hyper-rectangle containing x along the axis in which it is widest. Due to the expected monotonic and rapid decrease in magnitude of the x_i, this suggests that the hyper-rectangle is usually wider for small i than for large i, and that a few leading bits for x_i with large i are sufficient to determine that such coefficients are small and contribute little to the overall accuracy. Hence, many leading bits of the codeword will be allocated to $x_{0,\ldots,0}$—the mean value within a block—while the bits for small, high-frequency components are deferred until later since they have only small impact on accuracy.

Our encoding algorithm tracks the interval endpoints for each x_i. In each iteration, corresponding to the output of a single bit, it conceptually sorts the intervals by width. For each codeword bit, we split the widest interval; when there is a tie, we prefer x_i with low index (i.e., total sequency), i. The resulting scheme effectively reduces to *bit plane coding* (cf. [17,24]), where the n bits of a bit plane are encoded together before moving on to the next significant bit. The bracketing sequence associated with universal coding, however, quickly prunes many bits of a bit plane by marking whole groups of bits of a coefficient as zero.[5] Consequently, by simple bookkeeping (through tracking intervals), many bits of a bit plane are known to be zero and need not be coded explicitly.

The decoding step proceeds in reverse order and progressively narrows the n intervals based on the outcomes of single bit tests. The result of this process is a set of intervals that x is contained in. Our current approach is to simply use the lower interval bound along each dimension as representative. Other strategies, such as using the next split point or by rounding the input vector during encoding could also be used, though the latter is complicated by not knowing a priori the precision of each vector component, which is data-dependent.

[5] This marking is done in variable-radix coding [19] by testing whole digits of radix $\beta > 2$, e.g., four bits at a time are tested in Posits with $\beta = 2^4 = 16$.

3.4 Implementation

Although the algorithm in Listing 1 is straightforward, it involves an expensive step to repeatedly find the widest of $n = 4^d$ intervals (line 5). A linear search requires $O(n)$ time, which can be accelerated (especially for $d \geq 3$) to $O(\log n)$ time using a heap data structure. In each iteration, we operate on one of the n intervals and keep the remaining $n - 1 = 2^{2d} - 1$ intervals sorted in a heap, which conveniently is a perfect binary tree with $2d$ levels. Following the narrowing of an interval (line 16), we compare its width to the heap root's, and if still larger, we continue operating on the same interval in the next iteration. Otherwise, we swap the current interval with the heap root and sift it down (using $O(\log n)$ operations) until the heap property has been restored. For 3D data, we found the use of a heap to accelerate encoding by roughly 4×.

We note that the implementation of universal vector codes presented here has not been optimized for speed. The need to perform arithmetic on intervals and to process a single bit at a time clearly comes at a substantial expense. We see potential speedups by tracking interval widths in terms of integer exponents instead. We may also exploit faster scalar universal coding schemes developed, for example, for Posits, which process multiple bits at a time. Furthermore, it may be possible to avoid data-dependent coding by exploiting expected relationships between coefficient magnitudes such that the order in which bits are interleaved may be fixed. Such performance optimizations are left as future work.

4 Results

We begin our evaluation by examining the rate-distortion tradeoff when encoding static floating-point outputs from scientific simulations. Although it is common to compare representations by plotting the signal-to-noise ratio (SNR) as a function of rate—the number of bits of storage per scalar value—we have chosen to represent the same information in terms of what we call the *accuracy gain* vs. rate. We define the accuracy gain, α, as

$$\alpha = \log_2 \frac{\sigma}{E} - R = \frac{1}{2} \log_2 \frac{\sum_i (x_i - \mu)^2}{\sum_i (x_i - \tilde{x}_i)^2} - R, \tag{5}$$

where σ and μ are the standard deviation and mean of the original data, x_i is one of the original data values and $\tilde{x}_i$ is its approximation in a given finite-precision number system, E is the L_2 error (distortion), and R is the rate. Here the term $\log_2 \frac{\sigma}{E}$ provides a lower bound on the rate required to encode an (uncorrelated) i.i.d. Gaussian source within error E [4, §10.3.2], and effectively serves as a baseline against which R is measured. For correlated data, we expect $R \leq \log_2 \frac{\sigma}{E}$ for a number representation that exploits correlation, resulting in $\alpha \geq 0$. Conversely, because scalar representations like IEEE 754 and Posits ignore such correlations, they yield $\alpha \leq 0$. We note that α is high when the error, E, and the rate, R, are low. For effective coding schemes, $\alpha(R)$ tends to increase from zero at low rates—indicating that "compression" is achieved—until a stable plateau is reached, when each additional bit encoded results in a halving of the error—indicating that random, incompressible significand bits have been reached. Ultimately, E either converges to zero ($\alpha \to \infty$) or to some small nonzero value, e.g., due to roundoff errors, where additional precision is not helpful ($\alpha \to -\infty$). The maximum α indicates the amount of redundant information that a representation is able to eliminate. In addition, α allows comparing the efficiency of representations when both R and E differ by a nonnegligible amount, which using R, E, or $\text{SNR} = 20 \log_{10} \frac{\sigma}{E}$ alone would be difficult.

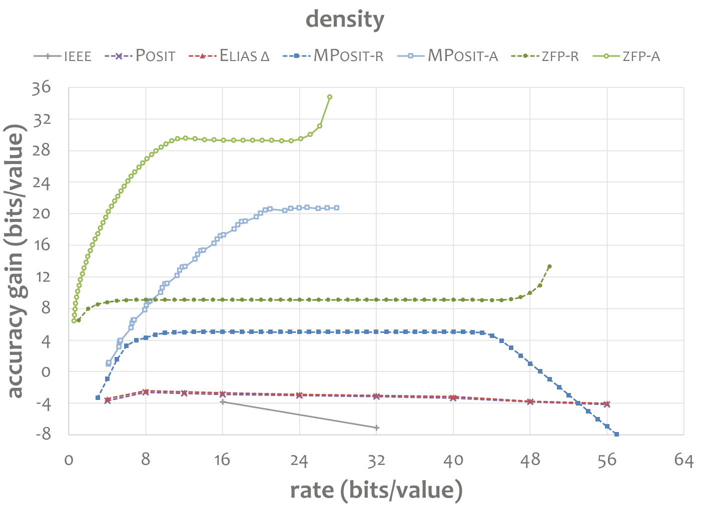
4.1 Static Data

Figure 3 plots the accuracy gain (higher is better) for various representations of two fields (density and viscosity) from a hydrodynamics simulation.[6] The density field varies in the range $[1, 3]$ while the viscosity field spans many orders of magnitude and also includes negative values.

The representations compared include IEEE 754 (half and float); Posits and Elias δ; two versions of MultiPosits based on our universal vector coding scheme; and two corresponding versions of zfp. Here the -R suffix indicates fixed-rate representations, where each block is assigned the same number of bits; the -A suffix indicates fixed-accuracy representations, where a given error tolerance dictates the storage size of each block. Fixed-accuracy mode is generally preferable when emphasis is on error rather than storage size, as then errors are roughly uniform over the entire domain, which allows for a smaller storage budget when the tolerance is met. In fixed-rate mode, additional bits are typically spent on each block, but the total L_2 error is usually dominated by the highest-error blocks. Hence, reducing the error nonuniformly across blocks does not appreciably reduce the total error but does increase storage. Of course, the variable-rate storage associated with a fixed-accuracy representation complicates memory management and random access, but we include such results here as they serve an important use case: offline storage and sequential access.

The two plots in Fig. 3 suggest several trends. First, fixed-rate MultiPosits generally improve on Posits by about 3.5–8 bits of accuracy across a wide range of rates; Posits in turn perform better than IEEE. The negative accuracy gain for the scalar representations essentially corresponds to the overhead of encoding exponents, and we see that IEEE does worse when using 8 (float) rather than 5 (half) exponent bits. In all cases, zfp outperforms MultiPosits, for reasons that will be discussed below. We also see that fixing the accuracy (-A) rather than rate (-R) is a substantial improvement. We note that this may be of importance for I/O and communication applications, where the data is serialized and transferred sequentially. While we have implemented fixed-accuracy mode for MultiPosits, the same idea could be generalized to scalar representations like Posits and IEEE, i.e., by truncating any significand bits whose place value fall

[6] The double-precision fields are from the Miranda code and are available from SDRBench at https://sdrbench.github.io.



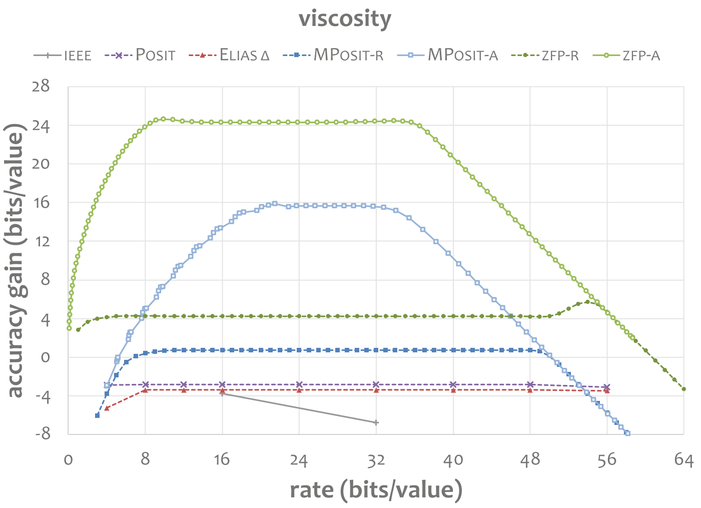


Fig. 3. Accuracy gain as a function of rate for two $384 \times 384 \times 256$ fields from a hydrodynamics simulation. The small dynamic range of the density field allows it be represented without loss using most representations, resulting in an eventual uptick in accuracy gain as the error approaches and even reaches zero. The viscosity field cannot be represented without loss, and the error eventually converges to a small value, resulting in an eventual decline in accuracy gain as additional bits do not reduce the error.

below some given power of two. Another observation is that MultiPosits-A gives a somewhat irregular curve both in terms of rate and accuracy gain, in contrast to zfp-A; there is a noticeable jump in both R and α every four data points at low rates. These jumps correlate with the increase in number of Posit regime bits [9], which occur every time the binary exponent increases or decreases by four. Such increments introduce a flurry of additional bits for the next bit plane that increase both rate and accuracy. We finally note that some representations, like MultiPosits, incur additional roundoff error from the use of double-precision arithmetic, as evidenced by the gap between MultiPosits and zfp at high rates.

4.2 Dynamic Data

We have implemented our universal vector code within the context of the zfp framework, which accommodates user-defined codecs for its compressed-array C++ classes. These classes handle encoding, decoding, and caching of blocks (in ieee double-precision format) for the user and expose a conventional multidimensional array API, thus hiding all the details of how the arrays are represented in memory. We additionally implemented a codec that uses a traditional scalar representation of blocks to allow for an apples-to-apples comparison using a single array implementation.

Based on these arrays, we implemented a 3D Poisson partial differential equation (PDE) solver using finite differences with Gauss-Seidel updates, i.e., array elements are updated in place as soon as possible. The equation solved is

$$\Delta u(x, y, z) = \sqrt{x^2 + y^2 + z^2} = r \tag{6}$$

on $\Omega = [-1, 1]^3$ with boundary condition $u = \frac{1}{12}r^3$ and initial condition $u = 0$ on the interior of the domain. Given this setup, the closed form solution equals $u = \frac{1}{12}r^3$ on the entire domain. We use a standard second-order 7-point stencil for the Laplacian finite difference operator and a grid of dimensions 64^3. Higher-order stencils did not appreciably change the results.

Figure 4 plots the L_2 error in Δu as a function of solver iteration. As is evident, the low-precision scalar types quickly converge to a fixed error level as they run out of precision to accurately resolve differences. The MultiPosit and zfp vector types perform significantly better, both at 16- and 32-bit precision. Compared to ieee 32-bit float, the 32-bit MultiPosit representation improves the solution accuracy by five orders of magnitude.

5 Discussion

Our universal vector codes generalize the corresponding scalar codes for correlated multidimensional fields that often arise in scientific computing. Using a decorrelating step, we decouple the vector quantization step into independent scalar quantization steps and later interleave the bits from their binary representation so as to minimize error. Our framework relies on the simple and general framework from [18] to produce a codeword one bit at a time, which ensures a straightforward if inefficient implementation.

Our framework shares several steps with the zfp number representation for multidimensional blocks:

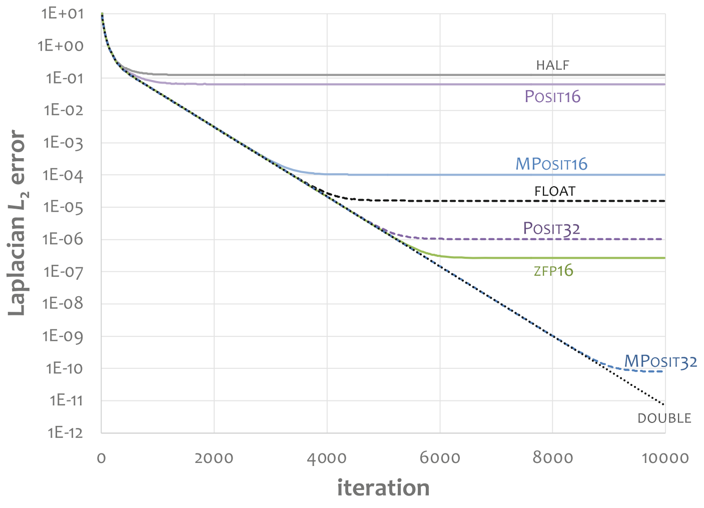


Fig. 4. Poisson equation solution error for various representations of the evolving state variable. Not shown is ZFP32, which coincides with the curve for DOUBLE.

– A decomposition of d-dimensional arrays into blocks of 4^d values.
– The same fast, linear, decorrelating transform. The key difference is that we implement our transform in floating point rather than integer arithmetic.
– The same reordering and prioritization of transform coefficients.

The two frameworks also differ in several ways:

– Whereas ZFP uses a fixed-length encoding of a single per-block exponent, we use per-coefficient tapered exponent coding.
– We inherit the same two's complement representation used for scalar POSITS, whereas ZFP encodes integer values in negabinary.
– ZFP makes use of additional control bits to encode the outcome of group tests, which apply to multiple bits within the same bit plane. Our framework does not use group testing across vector components but rather within each scalar to form regime bits. These bits directly refine the representation, like all significand bits, whereas ZFP's group tests instead govern the control flow.

In head-to-head competition, ZFP is a clear winner in terms of accuracy, storage, and speed, in part due to a more sophisticated coding scheme, though the speed advantage comes from its ability to process multiple bits simultaneously. Our framework as designed is data-dependent and operates at the single-bit level. ZFP also has the advantage of exploiting the sparsity of transform coefficients, which allows concise encoding of up to 4^d zeros in d dimensions using a single bit. In contrast, our scheme achieves only up to 4:1 "compression" of POSIT zero-bits and must encode at least 2×4^d bits to finitely bound each of the 4^d transform coefficients. By comparison, ZFP routinely allows a visually fair representation of 3D blocks using one bit per value or less. In fixed-rate mode, our framework like zfp suffers from lack of proper rounding, as the per-coefficient precision is data-dependent and not known until encoding completes. Hence, coefficients are always rounded toward $-\infty$. We suggest possible strategies to combat the effects of improper rounding above. Finally, the tapered nature of Posits and related number systems implies that blocks whose transform coefficients differ significantly from one may require many bits to even bracket the coefficients. In fact, the range preserving nature of the decorrelating transform on average causes already small coefficients to be reduced even further. zfp performs some level of bracketing by aligning all values to a single common block exponent, which requires only a fraction of a bit per value to encode.

From these observations, we conclude that MultiPosits offer a significant advantage over Posits in applications that involve smooth fields while not rivaling the zfp number system. Nevertheless, we believe that the ideas explored here may seed follow-on work to improve upon our framework, both with respect to accuracy per bit stored and speed. For instance, our coding scheme ignores the potential for intra bit plane compression and the potential to avoid data dependencies by adapting codes better suited to each of the transform coefficients.

6 Conclusion

We have presented a universal encoding scheme that generalizes the Posit and other universal scalar number systems to vectors or blocks of numbers for numerical applications that involve spatially correlated fields. Our approach is to partition the data arrays into blocks, decorrelate the blocks using a fast transform, and then interleave bits from a universal coding of vector components in an error-optimal order. Using numerical experiments with real data and partial differential equation solvers, we demonstrated that MultiPosits may yield as much as a six orders-of-magnitude increase in accuracy over conventional Posits for the same storage, and even larger increases compared to ieee 754 floating point. While our approach, as currently presented, is primarily of theoretical interest due to its high computational cost, we envision that our results will inspire follow-on work to address the performance issues associated with bitwise coding of vectors. In particular, we hope to develop data-independent universal vector codes that reap similar per-bit accuracy benefits with near-zero computational cost.

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.

References

1. IEEE std 754-2019: IEEE standard for floating-point arithmetic (2019). https://doi.org/10.1109/IEEESTD.2019.8766229

2. Choquette, J., Gandhi, W., Giroux, O., Stam, N., Krashinsky, R.: NVIDIA A100 tensor core GPU: performance and innovation. IEEE Micro **41**(2), 29–35 (2021). https://doi.org/10.1109/MM.2021.3061394
3. Clark, M.A., Babich, R., Barros, K., Brower, R., Rebbi, C.: Solving lattice QCD systems of equations using mixed precision solvers on GPUs. Comput. Phys. Commun. **181**(9), 1517–1528 (2010). https://doi.org/10.1016/j.cpc.2010.05.002
4. Cover, T.M., Thomas, J.A.: Elements of Information Theory, 2nd edn. Wiley-Interscience (2005). https://doi.org/10.1002/047174882X
5. Daubechies, I., Sweldens, W.: Factoring wavelet transforms into lifting steps. J. Fourier Anal. Appl. **4**(3), 247–269 (1998). https://doi.org/10.1007/bf02476026
6. Diffenderfer, J., Fox, A., Hittinger, J., Sanders, G., Lindstrom, P.: Error analysis of ZFP compression for floating-point data. SIAM J. Sci. Comput. **41**(3), A1867–A1898 (2019). https://doi.org/10.1137/18M1168832
7. Elias, P.: Universal codeword sets and representations of the integers. IEEE Trans. Inf. Theory **21**(2), 194–203 (1975). https://doi.org/10.1109/tit.1975.1055349
8. Gustafson, J.L.: The End of Error: Unum Computing. Chapman and Hall (2015). https://doi.org/10.1201/9781315161532
9. Gustafson, J.L., Yonemoto, I.T.: Beating floating point at its own game: posit arithmetic. Supercomput. Frontiers Innov. **4**(2), 71–86 (2017). https://doi.org/10.14529/jsfi170206
10. Hamada, H.: URR: universal representation of real numbers. N. Gener. Comput. **1**, 205–209 (1983). https://doi.org/10.1007/bf03037427
11. Kalamkar, D., et al.: A study of BFLOAT16 for deep learning training (2019). https://doi.org/10.48550/arXiv.1905.12322
12. Kalliojarvi, K., Astola, J.: Roundoff errors in block-floating-point systems. IEEE Trans. Signal Process. **44**(4), 783–790 (1996). https://doi.org/10.1109/78.492531
13. Klöwer, M.: Sonum8 and Sonum16 with maximum-entropy training (2019). https://doi.org/10.5281/zenodo.3531887
14. Klöwer, M., Düben, P.D., Palmer, T.N.: Posits as an alternative to floats for weather and climate models. In: Conference for Next Generation Arithmetic, pp. 2.1–2.8 (2019). https://doi.org/10.1145/3316279.3316281
15. Klöwer, M., Razinger, M., Dominguez, J.J., Düben, P.D., Palmer, T.N.: Compressing atmospheric data into its real information content. Nat. Comput. Sci. **1**, 713–724 (2021). https://doi.org/10.1038/s43588-021-00156-2
16. Köster, U., et al.: Flexpoint: an adaptive numerical format for efficient training of deep neural networks. In: Conference on Neural Information Processing Systems, pp. 1740–1750 (2017). https://doi.org/10.48550/arXiv.1711.02213
17. Lindstrom, P.: Fixed-rate compressed floating-point arrays. IEEE Trans. Visual Comput. Graphics **20**(12), 2674–2683 (2014). https://doi.org/10.1109/TVCG.2014.2346458
18. Lindstrom, P.: Universal coding of the reals using bisection. In: Conference for Next Generation Arithmetic, pp. 7:1–7:10 (2019). https://doi.org/10.1145/3316279.3316286
19. Lindstrom, P.: Variable-radix coding of the reals. In: IEEE 27th Symposium on Computer Arithmetic (ARITH), pp. 111–116 (2020). https://doi.org/10.1109/ARITH48897.2020.00024
20. Lindstrom, P., Lloyd, S., Hittinger, J.: Universal coding of the reals: alternatives to IEEE floating point. In: Conference for Next Generation Arithmetic, pp. 5:1–5:14 (2018). https://doi.org/10.1145/3190339.3190344

21. Morris, R.: Tapered floating point: a new floating-point representation. IEEE Trans. Comput. **C-20**(12), 1578–1579 (1971). https://doi.org/10.1109/T-C.1971.223174
22. Noll, P., Zelinski, R.: Comments on "Quantizing characteristics for signals having Laplacian amplitude probability density function." IEEE Trans. Commun. **27**(8), 1259–1260 (1979). https://doi.org/10.1109/TCOM.1979.1094523
23. Omtzigt, E.T.L., Gottschling, P., Seligman, M., Zorn, W.: Universal numbers library: design and implementation of a high-performance reproducible number systems library (2020). https://doi.org/10.48550/arXiv.2012.11011
24. Said, A., Pearlman, W.A.: A new, fast, and efficient image codec based on set partitioning in hierarchical trees. IEEE Trans. Circuits Syst. Video Technol. **6**(3), 243–250 (1996). https://doi.org/10.1109/76.499834
25. Thien, D., Zorn, B., Panchekha, P., Tatlock, Z.: Toward multi-precision, multi-format numerics. In: IEEE/ACM 3rd International Workshop on Software Correctness for HPC Applications (Correctness), pp. 19–26 (2019). https://doi.org/10.1109/Correctness49594.2019.00008
26. Wallace, G.K.: The JPEG still picture compression standard. IEEE Trans. Consum. Electron. **38**(1), xviii–xxxiv (1992). https://doi.org/10.1109/30.125072
27. Wang, R.: Introduction to Orthogonal Transforms. Cambridge University Press (2012). https://doi.org/10.1017/cbo9781139015158
28. Wilkinson, J.: Error analysis of floating-point computation. Numer. Math. **2**, 319–340 (1960). https://doi.org/10.1007/BF01386233