Efficient Hyperdimensional Computing with Spiking Phasors

Jeff Orchard[1]
P. Michael Furlong[2]
Kathryn Simone[1]
[1]Cheriton School of Computer Science, University of Waterloo, Waterloo, Canada
[2]Systems Design Engineering, University of Waterloo, Waterloo, Canada

Keywords: vector symbolic architecture, hyperdimensional computing, FHRR, spiking, phasor, binding, clean-up memory, neuromorphic

Accepted for publication in *Neural Computation*, 36:9, September 2024.

Abstract

Hyperdimensional (HD) computing (a.k.a. Vector Symbolic Architectures, VSAs) offer a method for encoding symbols into vectors, allowing for those symbols to be combined in different ways to form other vectors in the same vector space. The vectors and operators form a compositional algebra, such that composite vectors can be decomposed back to their constituent vectors. Many useful algorithms have implementations in HD computing, such as classification, spatial navigation, language modelling, and logic. In this paper, we propose a spiking implementation of Fourier Holographic Reduced Representation (FHRR), one of the most versatile VSAs. The phase of each complex number of an FHRR vector is encoded as a spike time within a cycle. Neuron models derived from these spiking phasors can perform the requisite vector operations to implement an FHRR. We demonstrate the power and versatility of our spiking networks in a number of foundational problem domains, including: symbol binding/unbinding, spatial representation, function representation, function integration, and memory (i.e. signal delay).

1 Introduction

Artificial neural networks are ubiquitous tools in modern computing, but bring with them substantial power demands. Spiking neural networks could have a big impact on the future of computing since they can take full advantage of the extreme parallelism

and low power requirements of neuromorphic hardware. While it is possible to translate gradient-based learning to spiking neural systems, implementing backpropagation methods in neuromorphic hardware, as in the brain, brings with it the weight transport problem, which imposes real, material costs in implementation. An alternative means to implementing powerful algorithms in spiking neural networks is to combine local learning rules in combination with symbolic structure imposed on neural circuits.

Vector symbolic architectures (VSAs) are one way of imposing structure on neural systems, but often the translation from their algebraic statements to their implementation in populations of spiking neurons comes at a considerable overhead, *i.e.*, representing each dimension of a hyperdimensional vector with multiple spiking neurons. While neuromorphic computers are coming closer to commercial realization, the memory technologies used to implement neuron dynamics and synaptic memories are still costly (Davies, 2021), hence there is a clear need for neuron-efficient implementations of VSAs.

In this paper, we demonstrate how a versatile and powerful neural paradigm can be implemented using a relatively small number of spiking neurons. We extend previous work (Renner et al., 2022b; Bent et al., 2022; Orchard and Jarvis, 2023) that shows how spike times within a cycle can be leveraged to represent hypervectors in a Vector Symbolic Architecture. We propose a specific neuron model, which we implement using the Brian2 simulator, that enables the translation of the chosen VSA into a population of neurons that need only one neuron per dimension of the VSA's hyperdimensional vectors. This particular VSA, the Fourier Holographic Reduced Representation (FHRR; Plate, 2003, Ch. 4), is capable of implementing an impressive array of algorithms, and here we demonstrate some of those on a substrate of spiking neurons.

Below we provide a brief review of VSAs, and in particular the Fourier Holographic Reduced Representation VSA (Section 1.1). Next we propose a specific neuron model that translates three key operations (binding, unbinding, and clean up memory) into efficient neural populations (Section 2). Finally, we demonstrate a number of applications of VSAs using our neuron model (Section 3), before discussing the implications of the proposed approach (Section 4), and finally, concluding.

1.1 Vector Symbolic Architectures

A VSA offers a way to represent data using high-dimensional vectors (Gayler, 2003). Also called Hyperdimensional (HD) computing, the "hypervectors" can be combined in different ways to produce other (hyper)vectors in the same vector space. Amazingly, those new vectors can still be decomposed into their constituent vectors. Hence, a VSA forms a type of compositional language over a set of symbols. The set of symbols in a VSA is called a *vocabulary*.

A VSA is comprised of a vector space paired with a small collection of operations: similarity $\odot$, binding $\otimes$, unbinding $\oslash$, bundling $\oplus$, permutation $\rho(\cdot)$, and clean-up.

The **similarity** operation measures how "close" two vectors are. Importantly, pairs of randomly-chosen, high-dimensional vectors of a VSA tend to yield a similarity close to zero. For example, consider a vector space $\mathbb{V}$, and two random vectors, $u, v \in \mathbb{V}$. We would expect that $u \odot v \approx 0$, while $v \odot v = u \odot u = 1$.

Binding combines two vectors to get another vector that is not similar to either of

the two. Thus, if $w = u \otimes v$, then $w \odot u \approx 0$ and $w \odot v \approx 0$. Binding can be used to represent the conjunction of elementary symbols of the vocabulary. In some VSAs, binding can also be used as the basis for constructing continuous representations, but this is contingent on the definition of the binding operator. **Unbinding** does the opposite of binding to recover the constituent elements of the conjunction, so that $w \oslash u \approx v$.

Bundling creates a vector that is still highly similar to the two constituent vectors. If $w = u \oplus v$, then $w \odot u \gg 0$, though not necessarily equal to 1. This corruption comes from the fact that some information is lost in the bundling operation. How much is lost depends on the vector space, the specific bundling operation, and how many vectors were bundled together. Bundling can be used to construct vectors that represent sets of the constituent vector-symbols.

Permutation shuffles a vector – in a reversible way – so that the resulting vector is dissimilar from the original. Thus, if $w = \rho(v)$, then $w \odot v \approx 0$, but $\rho^{-1}(w)$ yields v back. Permutation is used to construct data structures that are represented as vectors. Permutation induces structure by removing the commutativity of operations like bundling and binding, while preserving similarity between vectors. For example, an ordered sequence can be constructed: $v \oplus \rho(u) \oplus \rho(\rho(w)) \oplus \cdots$. In this way, bundles can be constructed to reflect structures, including sequences, trees, and graphs (Kanerva, 2009; Plate, 2003).

Finally, **clean-up** takes a vector and restores it to the closest match in the VSA's vocabulary. The clean-up is helpful because the operations of binding, bundling, etc., tend to corrupt the vectors, and noise can accumulate. Using the noisy vectors in further operations could start to affect their proper function and produce unpredictable behaviours. The clean-up operation can undo that corruption and the clean vector can be used with greater confidence.

1.2 Fourier Holographic Reduced Representation

Fourier Holographic Reduced Representation (FHRR) is a VSA that uses complex-valued vectors to encode symbols (Plate, 1995). Let $v \in \mathbb{C}^N$ be a complex vector. The vectors of an FHRR are simply the Fourier transform of Holographic Reduced Representation (HRR) vectors; many HRR VSA operations can be done more efficiently using this Fourier representation (Plate, 1995). The complex number v_k can be written $|v_k|\mathrm{e}^{\mathrm{i}\phi_k}$, where $|v_k|$ is the modulus, and ϕ_k is its phase. If the vectors in the FHRR exhibit conjugate symmetry (so that $v_k = \overline{v}_{N-k}$ for $k = 0, \ldots, N-1$, where the bar notation indicates complex conjugation), then taking the inverse Fourier transform of v yields a real-valued vector, $\hat{v} \in \mathbb{R}^N$.

A complex-valued vector v is said to be *unitary* if $|v_k| = 1$ for each of its elements (Plate, 1995). If all the vectors in an FHRR are unitary, we will refer to it as a *normalized FHRR*. The use of unitary vectors in an FHRR simplifies binding and unbinding, since no arithmetic is required to compute the modulus.

Binding: In a normalized FHRR, the binding operation is done using element-wise multiplication, the *Hadamard* product. The vectors of a normalized FHRR are *unitary*, meaning that the vector elements are unit-modulus complex numbers, so that multiplying them is equivalent to adding their phases, $\mathrm{e}^{\mathrm{i}\phi_1}\,\mathrm{e}^{\mathrm{i}\phi_2} = \mathrm{e}^{\mathrm{i}(\phi_1+\phi_2)}$. The inverse operation – unbinding – is element-wise division (or multiplication by the conjugate),

as in $\mathrm{e}^{\mathrm{i}\phi_1}\,\mathrm{e}^{-\mathrm{i}\phi_2} = \mathrm{e}^{\mathrm{i}(\phi_1-\phi_2)}$.

Similarity: If u and v are vectors in an FHRR, then their similarity, $u \odot v$, is computed using the complex inner product, $u \cdot \bar{v}$, where $\bar{v}$ is the complex conjugate of of v. For a normalized FHRR, this similarity is the same as cosine similarity.

Bundling: In many VSAs, including the FHRR, the bundling operation is simply vector addition. Consider adding two unit-modulus complex numbers, $\mathrm{e}^{\mathrm{i}\phi_a} + \mathrm{e}^{\mathrm{i}\phi_b}$. The resulting complex number will have a phase of $\frac{1}{2}(\phi_a + \phi_b)$, but will probably not be unit-modulus. In a normalized FHRR, the modulus is ignored, and only the phase is kept. Discarding the modulus is a source of information loss, and is one of the reasons that bundling has limitations.

Permutation: Permutation is done by literally permuting the vector elements. If $\mathbf{P}$ is a permutation matrix, then $\rho(v) = \mathbf{P}v$, and $\rho^{-1}(u) = \mathbf{P}^{\mathrm{T}}u$.

Clean-up: In an HRR or FHRR, clean-up is often done either by an recurrent, Hopfield-like associative memory (in which the vocabulary vectors are fixed points) (Frady et al., 2018; Frady and Sommer, 2019; Frady et al., 2022), or by a feed-forward associative memory (Stewart et al., 2011).

Fractional Binding: The FHRR has an additional operation that is not present in all VSAs. Consider binding a vector v with itself, yielding $v \otimes v$. In the FHRR, $\otimes$ is the Hadamard product, so you can write that as v^2, where the exponent is applied to the vector elementwise. But now one can contemplate using non-integer exponents, such as $v^{1.5}$, or $v^{0.7}$ (Plate, 1995). This is called *fractional binding* (Lu et al., 2019), or *Fractional Power Encoding* (FPE) (Frady et al., 2018, 2021, 2022). If one of the elements of a unitary vector is $\mathrm{e}^{\mathrm{i}\phi}$, with $-\pi < \phi \leq \pi$, then raising that vector to the exponent α yields a vector with the element $\mathrm{e}^{\mathrm{i}\phi\alpha}$. In other words, the fractional power encoding of α is the same as multiplying all the phases in the vector by α.

1.3 Spiking Phasors

In most neural network simulations, a neuron's activity is represented by a real number. However, there is utility in modelling a neuron's activity state as a unit-modulus complex number, or *phasor* (Noest, 1988). This approach works well with the normalized FHRR, since its vector components are unit-modulus complex numbers. Moreover, since each complex number in a normalized FHRR vector really only encodes phase, that complex number can also be represented by the timing of a spike within a cycle – the spike's phase. Thus, each complex number in a normalized FHRR hypervector can be represented by a single neuron. All the neurons spike at the same frequency, but they spike at different times with respect to a global, baseline cycle. Each neuron's spike time in that cycle represents that neuron's phase (Frady et al., 2018). These neurons are called *spiking phasors*. An N-dimensional vector in a normalized FHRR can be represented by the spikes of a population of N spiking phasors. Each neuron emits one spike per cycle, so that the population, as a whole, renders the FHRR vector each cycle as N spike trains coming from N neurons.

Binding (unbinding) amounts to phase summation (or phase subtraction) in a normalized FHRR. Put in terms of spiking phasors, binding (unbinding) two incoming spike trains amounts to adding together (subtracting) the incoming spike times, and generating a spike at this new phase sum (difference). A discretized version of such

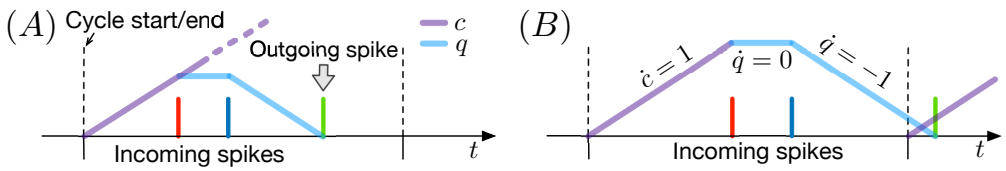

Figure 1: Phase sum neuron model. (A) illustrates the model for the case when the sum of the incoming phases is less than 1 cycle. (B) shows how the method still works even if their sum is longer than 1 cycle. Lines for c and q are only shown where they are relevant.

binding was demonstrated using binary one-hot vectors, where the location of the 1 indicated the binned spike time (Laiho et al., 2015). In that VSA, binding was done using circular bit shifts. Rather than binning a spike time, one can use the spike time itself to represent phase. Binding is done by adding these spike times (Renner et al., 2022b; Bent et al., 2022). Similarly, fractional binding can be implemented in a spiking-phasor neuron model by multiplying its phase offset by the desired multiplier (α). These neuron models will be described in more detail in the next section.

2 Methods

2.1 Binding (Phase Sum)

In the normalized FHRR, binding is done by the element-wise multiplication of unit-modulus complex numbers, which is equivalent to adding their phases as discussed in Section 1.2. We developed a spiking-phasor neuron model that receives spikes from 2 other spiking phasors, and is able to add the phases of those incoming spikes and generate its own spike at that phase sum.

The neuron model has 2 internal integrators. One is the cycle-tracking variable, or clock. Suppose the neuron cycles at frequency λ Hz. Then its period is $L = \frac{1}{\lambda}$ seconds. The variable c starts at 0 at the beginning of the cycle and follows the differential equation $\dot{c} = 1$ until it reaches its threshold value of L, indicating the end of the cycle, at which point c resets to 0 and the cycle repeats.

When the first spike arrives, a second integrator q is set equal to c, thus recording the spike's arrival time. The integrator q holds that value by following the differential equation $\dot{q} = 0$ until another spike arrives. After the second spike, q starts integrating down following $\dot{q} = -1$ until it reaches its threshold value of 0, at which point the neuron generates its own spike. This process is illustrated in Fig. 1A. Part B of that figure shows that the neuron model still works when the sum of the spike times is larger than the cycle period.

The spikes that arrive are not labelled "first" or "second". Rather, the neuron model performs the same internal operation as each spike arrives, regardless of whether it's the first or second spike. The differing effects of the two spikes are implemented using a $\max$ and $\min$ function. Here is how. We initialize $\dot{q} \leftarrow 1$ and decrement $\dot{q} \leftarrow \dot{q} - 1$ for each spike that arrives. Also, when a spike arrives, we increment $q \leftarrow q + \max(\dot{q}, 0) * c$,

which only affects q on the first spike since that is the only incoming spike for which $\dot{q} > 0$. Similarly, instead of incrementing the integrator q using $q \leftarrow q + \Delta t * \dot{q}$ each time step (where Δt is the step size), it is incremented by $q \leftarrow q + \Delta t * \min(\dot{q}, 0)$. Thus, q does not change until $\dot{q}$ is negative, which will happen after the 2nd spike. This strategy adds extra comparisons to each time step, but allows the spike-arrival behaviours to be generic. This model is summarized in Algorithm 1.

Data: $c \leftarrow 0$, and $\dot{q} \leftarrow 1$
for *each time step of* Δt **do**
  Update integrators:
  $c \leftarrow c + \Delta t$
  $q \leftarrow q + \Delta t * \min(\dot{q}, 0)$
  if *a spike arrives* **then**
    $q \leftarrow c * \max(\dot{q}, 0)$
    $\dot{q} \leftarrow \dot{q} - 1$
  end
  if $q < 0$ **then**
    SPIKE, and $q \leftarrow 0$, $\dot{q} \leftarrow 1$
  end
  if $c > L$ **then**
    Cycle Reset: $c \leftarrow 0$
  end
end

Algorithm 1: Phase-sum neuron model

Note that the outgoing spike will always occur before the first incoming spike in a cycle, as long as the the phases of the incoming spikes stay constant. The binding model will still work even if the phases are changing, since it recomputes the phase sum every cycle. However, depending on the specific spike times, it might take an extra cycle to settle to the correct phase sum.

Our binding model assumes that two spikes arrive each cycle. If 3 or more spikes arrive in a cycle, this neuron model will not work properly. However, a similar model can be constructed to bind together 3 spikes per cycle, or 4, or more. For a given phase-sum layer, the number of vectors being bound together would need to be known *a priori* so the appropriate neuron model could be used. If not, the model will not correctly sum the incoming phases.

2.2 Unbinding (Phase Subtraction)

A similar neuron model, illustrated in Fig. 2, is used to implement unbinding by subtracting spike times. In this model, the variable q starts integrating from 0 when the first spike arrives, according to $\dot{q} = 1$. When the second spike arrives, an internal threshold variable, θ, is set equal to the instantaneous value of q. The neuron spikes when the cycle-tracking variable, c, reaches the threshold ($c = \theta$), after which a refractory toggle is set to 1 until the start of the next cycle (preventing the neuron from spiking again until the next cycle). Unlike the other neuron models described in this paper, the unbinding

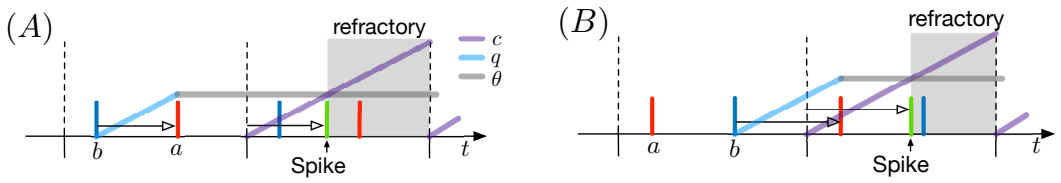

Figure 2: Phase subtraction neuron model. Both plots illustrate spike time b subtracted from spike time a. In (A), the spike b occurs before a in the cycle. In (B), the spike a occurs before b. In each plot, the two horizontal arrows are the same width. Lines for c, q, and θ are only shown where they are relevant for the cycle's computation.

neuron model is the only one that distinguishes between its two input synapses, since $a - b$ is different from $b - a$. In contrast to the binding neuron model, where the spike could be generated in the same cycle or the next cycle in which the spikes arrived, the unbinding neuron model always generates spikes in the following cycle.

2.3 Fractional Binding (Phase Multiplication)

The operation of fractional binding of the value α turns into multiplying phase by α. In terms of spike times, the displacement of the spike from the nearest cycle boundary is multiplied by α.

In this model, the clock variable (which we will denote $\hat{c}$) follows the differential equation $\dot{\hat{c}} = \lambda$ and wraps half-way through the cycle; it starts at 0 at the beginning of the cycle, then jumps from $\frac{1}{2}$ to $-\frac{1}{2}$ mid-cycle. This allows the clock variable $\hat{c}$ to increase smoothly from negative to positive as one cycle ends and the next begins. This continuous transition is important because $\hat{c}$ is multiplied by α when the spike arrives, setting a threshold variable, $\theta = \alpha\hat{c}$. When $\hat{c} > \theta$, the neuron sends an outgoing spike and engages an internal refractory toggle that stops the neuron from firing another spike until the next mid-cycle reset. Half-way through the next cycle, the toggle is reset and the neuron is ready to fire again.

The inner workings of the phase multiplication neuron model are depicted in Fig. 3. Notice that the red spike's displacement (ϕ) from the cycle start/end is multiplied by α to yield the outgoing spike's time ($\alpha\phi$, in green). The figure also shows how the clock variable $\hat{c}$ sets the threshold θ when the spike arrives, and a spike is generated as soon as $\hat{c}$ reaches that threshold. Note that the resulting (green) spike could be in the current cycle, or in the next cycle, depending on when the incoming spike arrives, and the value of α being encoded.

If α is large, the resulting threshold could be outside of the range $\left[-\frac{1}{2}, \frac{1}{2}\right]$. However, this simply means that the desired spike time has to be phase-wrapped. For example, a threshold of $\theta = 0.7$ corresponds to the same spike time as a threshold of $\theta = -0.3$. If θ is outside $\left[-\frac{1}{2}, \frac{1}{2}\right]$, the neuron model increments (decrements) θ by 1 (-1, respectively) each time step until $-\frac{1}{2} < \theta \leq \frac{1}{2}$.

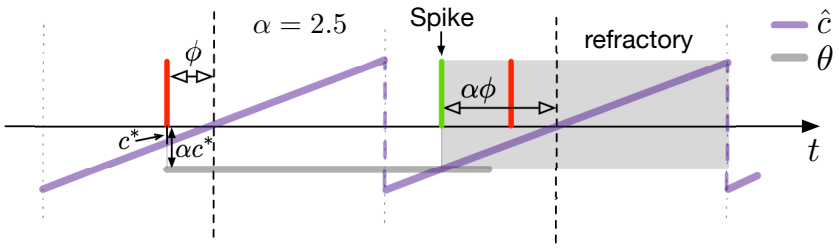

Figure 3: Phase multiplication neuron model. This neuron is reset halfway through the baseline cycle, as tracked by the integrator variable $\hat{c}$. The relative displacement of the incoming spike (ϕ, red), is multiplied by α to get the desired time for the outgoing spike ($\alpha\phi$, green). When the red spike arrives, θ is set to $\alpha\hat{c}$. The neuron generates a spike when $\hat{c} > \theta$, after which the neuron is unable to spike again until $\hat{c}$ resets at the next mid-cycle. Lines for $\hat{c}$ and θ are shown only where they are relevant to the cycle's computation.

2.4 Clean-Up Memory

The clean-up memory is based on the two-part Hopfield architecture outlined in (Krotov and Hopfield, 2021). It consists of two populations, G and H, each containing resonate-and-fire (RF) neurons. When G receives an input pattern (hypervector), the two populations (G and H) continuously exchange activity until they settle into an equilibrium state, where G (hopefully) stores the nearest vocabulary vector.

Suppose the vectors in our VSA are d-dimensional, so that the population G has d RF neurons. That population projects to H, which is a population of m RF neurons, where m is the number of patterns we have in our vocabulary. Let $W \in \mathbb{C}^{m \times d}$ be the connection-weight matrix from G to H. Each row of W contains one of the vocabulary vectors. If we suppose that G is encoding a unitary vector $g \in \mathbb{C}^d$, then the input to H can be written as the matrix-vector product $\overline{W}g$. This input is like taking the inner product (similarity) between g and each vocabulary vector. If g is most similar to the kth row of W, then we would expect the input to h_k to be larger than the input to the other neurons in H. In this sense, each neuron in H represents one of the vocabulary vectors. Importantly, even though this matrix-vector product involves complex numbers, it is implemented using scalar weights and spike delays, as described in (Frady et al., 2018; Frady and Sommer, 2019).

The neurons in H all inhibit each other; each time a neuron spikes, it sends strong, inhibitory input to all the other neurons in H. Because of this mutual inhibition, the population tends toward a winner-takes-all (one-hot) state. In our experience, this convergence usually takes about 6 cycles.

Finally, H projects back to G using connection weights W^{T} (again, using scalar weights and synaptic delays). In other words, each time a neuron in H spikes, it sends to G a weighted copy of the pattern it represents. Figure 4 shows the G and H populations of the clean-up network.

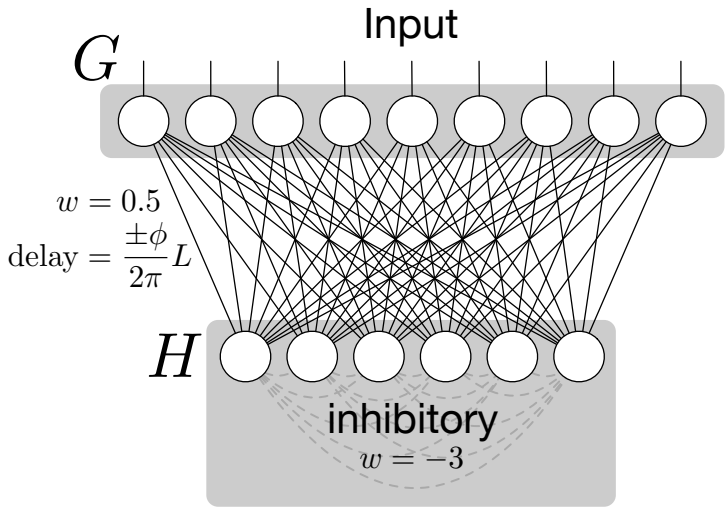

Figure 4: Clean-up network. Each neuron in H represents one of the vocabulary vectors. The connections from a neuron in H to all the neurons in G have a weight of 0.5, but delays that are determined by the phase of the vocabulary vector. The connections going the other way have the same weight, but with opposite (negative) delay.

The differential equations governing the neurons in G and H are,

$$\text{for each neuron in } G \begin{cases} \dot{x}_G(t) = -2\pi\lambda\, y_G - 0.4\, x_G \\ \dot{y}_G(t) = \quad 2\pi\lambda\, x_G - 0.4\, y_G \end{cases} \tag{1}$$

$$\text{for each neuron in } H \begin{cases} \dot{x}_H(t) = -2\pi\lambda\, y_H - (0.02+\eta)\, x_H \\ \dot{y}_H(t) = \quad 2\pi\lambda\, x_H - (0.02+\eta)\, y_H \\ \dot{\eta}(t) = -\dfrac{\eta}{\tau_\eta} \end{cases} \tag{2}$$

where λ is the cycle frequency (in Hz). Each time a neuron in G or H receives a spike, its x-value is incremented by the amount specified by the corresponding connection weight. The only exception is the inhibitory interactions between neurons in H. The neuron's variable η integrates the inhibitory spikes from all other neurons in H. Each time a neuron in H spikes, it increments the η-value of all other neurons in H by 3. These RF neurons send a spike when both $x > 0.9$ and $y > 0$, after which their x-value is reset to 0.7.

3 Experiments

In this section, we demonstrate the utility of these spiking-phasor neuron models, and show that they effectively implement state-of-the-art VSA algorithms. All these experiments were implemented in Python, using the Brian2 neural simulation language (Stimberg et al., 2019). The code can be downloaded from `https://github.com/jorchard/SpikingPhasorVSA`.

3.1 Impact of Clean-Up Memory

We wanted to investigate the effect of clean-up memory on the performance of our spiking-phasor networks. Because the vectors are not perfectly orthogonal, there is a slight loss of information when unbinding a bound vector. For example, suppose you have the unitary FHRR vectors v and a, and they are bound together to form a new vector $w = v \otimes a$. The unbound vector, $\hat{a} = w \oslash v$, will be close to a but contain some noise. That is, their similarity, $\hat{a} \odot a$, will be slightly less than the ideal value of 1.

A more significant source of error is bundling. A population of spiking-phasor neurons can only encode a unitary vector, but bundles (from adding vectors together) are not usually unitary. Renormalization back to a unitary vector results in a loss of information. To see this, consider the set of m unitary vectors $\mathcal{V} = \{v_1, \ldots, v_m\}$. Let B be the bundle of all the vectors in $\mathcal{V}$, so that $B = \sum_k v_k$. Since $\|v_k\|^2 = 1$ for each k, we know that $\|B\|^2$ can be as large as m. If we define $\hat{B}$ as $\frac{B}{\|B\|}$, then the similarity $\hat{B} \odot v_k$ will likely be smaller than $\frac{1}{\sqrt{m}}$, much lower than the ideal value of 1.

Let us put these two factors together. In addition to the set of vectors, $\mathcal{V}$, we introduce two more sets of m unitary vectors, $\mathcal{A} = \{a_1, \ldots, a_m\}$ and $\mathcal{U} = \{u_1, \ldots, u_m\}$. Using these sets of vectors, we construct two bundles by pairwise binding, so that

$$V = \sum_{k=1}^{m} v_k \otimes a_k \ , \text{ and}$$
$$U = \sum_{k=1}^{m} u_k \otimes a_k \ .$$

Finally, let $\hat{V}$ and $\hat{U}$ be unitary versions of those two bundles.

Given those bundle vectors, $\hat{U}$ and $\hat{V}$, as well as a particular v_j, we would like

$$\hat{U} \oslash \left(\hat{V} \oslash v_j \right) \tag{3}$$

to yield a vector that is closest to u_j. Why? Because $\hat{V} \oslash v_j$ should give a vector closest to a_j, and $\hat{U} \oslash a_j$ should give a vector closest to u_j. The problem is that $\hat{V} \oslash v_j$ is unlikely to be close enough to a_j to work in the second unbinding operation.

The clean-up memory can help by taking the result from the first unbinding operation, the vector from $\hat{V} \oslash v_j$, and cleaning it up by converging it toward the closest vector in $\mathcal{A}$. That cleaned-up vector has a much better chance at working in the second unbinding operation.

The following experiment compares the performance of two different networks at evaluating the sequential unbinding expression in (3). Denote the *clean-up* operation as a function $C_{\mathcal{S}}$ such that, given a set of vectors $\mathcal{S}$,

$$C_{\mathcal{S}}(w) \approx \underset{s \in \mathcal{S}}{\operatorname{argmin}} \left(w \odot s \right) \ .$$

One network, denoted Net-1, performs the operation without clean-up,

$$r_1 = \hat{U} \oslash \left(\hat{V} \oslash v_j \right) \ , \tag{4}$$

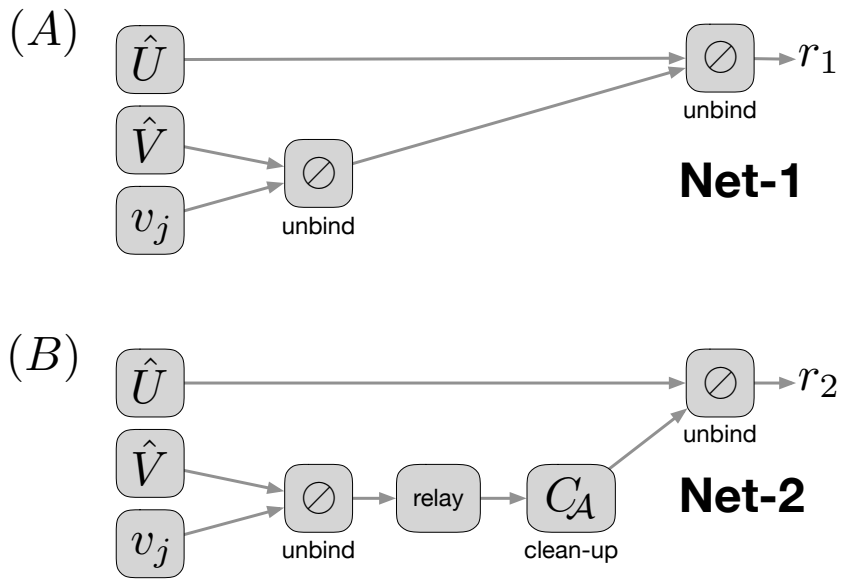

Figure 5: Network diagrams for Net-1 and Net-2. (A) Net-1 does not use a clean-up memory. (B) Net-2 uses a clean-up memory to restore $\hat{V} \oslash v_j$ to a_j. The "relay" node is a population of neurons that simply spike when they receive a spike, but go dormant after 250ms. For the remaining 250ms, the clean-up memory operates without input.

while the other network, Net-2, performs the operation with clean-up,

$$r_2 = \hat{U} \oslash C_{\mathcal{A}}\Big(\hat{V} \oslash v_j\Big) \ . \tag{5}$$

If Net-1 (or Net-2) is successful, then r_1 (or r_2) will be more similar to u_j than any of the other vectors in $\mathcal{U}$. The two networks are shown in Fig. 5. The "relay" node in Net-2 allows the spikes to reach $C_{\mathcal{A}}$ for the first 250ms, but then blocks the spikes for the remaining 250ms. This allows the clean-up memory to converge. The spiking phasors were cycling at 40Hz.

In our experiment, we used hypervectors of dimension of 512, and bundles containing $m = 30$ vectors. We used Net-1 and Net-2 to evaluate r_1 and r_2 from equations (4) and (5), for all 30 vectors. We counted how many times r_1 or r_2 had the highest similarity to the correct vector in $\mathcal{U}$. We ran this experiment 10 times, each time generating new vector sets ($\mathcal{V}$, $\mathcal{A}$, $\mathcal{U}$), giving a total of 300 trials. The non-clean-up Net-1 was successful in 48 of the 300 trials (16% accuracy), while the clean-up Net-2 was successful in 289 of the 300 cases (96% accuracy).

3.2 Spatial Memory

Several objects, and their locations, can be stored in a single vector. For example, given different vectors for each of {Red, Green, Blue, Square, Triangle, Circle, X, Y}, we can represent a red square at location $(-1.3, -1.1)$ as the vector,

$$\mathrm{Red} \otimes \mathrm{Square} \otimes \mathrm{X}^{-1.3} \otimes \mathrm{Y}^{-1.1} \ .$$

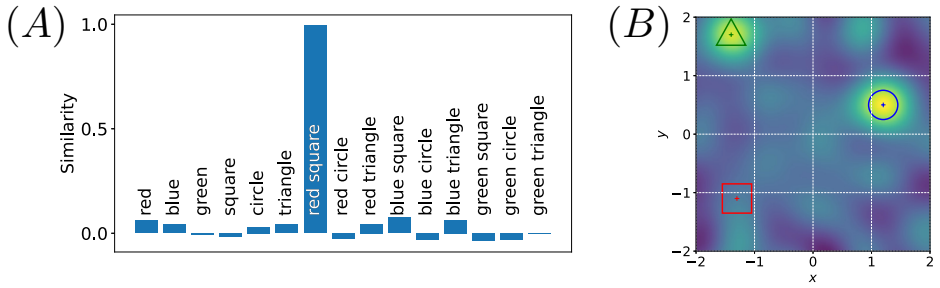

Figure 6: Queries of spatial memory vector, M. (A) shows the similarity between possible object vectors and the query $M \oslash \mathrm{X}^{-1.3} \oslash \mathrm{Y}^{-1.1}$. (B) shows the similarity between $M \oslash \big[\,(\mathrm{Blue} \otimes \mathrm{Circle}) \oplus (\mathrm{Green} \otimes \mathrm{Triangle})\,\big]$ and hypervectors of the form $\mathrm{X}^x \otimes \mathrm{Y}^y$. Notice that the similarity is highest at the locations of the blue circle and green triangle.

We can also encode several objects by bundling their vectors,

$$\hat{M} = \sum_k \mathrm{Colour}_k \otimes \mathrm{Shape}_k \otimes \mathrm{X}^{x_k} \otimes \mathrm{Y}^{y_k} \,,$$

where Colour_k, and Shape_k are different hypervectors corresponding to different colours and shapes, and x_k and y_k are scalars. Note that in order to store $\hat{M}$ in a spiking-phasor population, its elements must first be made unit-modulus, so that $M_j = \hat{M}_j / |\hat{M}_j|$. This bundle vector, M, can then be queried with questions like, "What is at location $(-1.3, -1.1)$?", or "Where is the blue circle?".

We constructed a neural network consisting of multiple populations of 480 spiking-phasor neurons, each representing a 480-dimensional hypervector. One population encoded M, while another encoded the query $q_a = \mathrm{X}^{-1.3} \otimes \mathrm{Y}^{-1.1}$. Both M and q_a project to a phase subtraction population that unbinds q_a from M. The output of the unbinding population projects to a relay population of 480 spiking-phasor neurons, which then projects to a clean-up memory of 495 neurons (480 neurons in G for the hypervector, and 15 neurons in H for the 15 candidate vectors in the vocabulary). Figure 6(B) shows the configuration of the 3 objects used to build the memory bundle vector, M. Figure 6(A) shows the results of that query; the vector for "red square" was the most similar.

Figure 6(B) also shows the results from the query, "Where are the blue circle and green triangle?" The corresponding query vector is

$$q_b = (\mathrm{Blue} \otimes \mathrm{Circle}) \oplus (\mathrm{Green} \otimes \mathrm{Triangle}) \,.$$

The heatmap in the figure indicates the similarity if $M \oslash q_b$ to hypervectors of the form $\mathrm{X}^x \otimes \mathrm{Y}^y$ over a region of the (x, y) plane. Notice that the similarity is highest at the locations of the blue circle and green triangle.

From a biological perspective, it is interesting to note that the phases of the location basis vectors can be chosen in a way that can easily yield grid cells (Welday et al., 2011; Orchard et al., 2013; Dumont and Eliasmith, 2019). Figure 7 shows a simulation of an

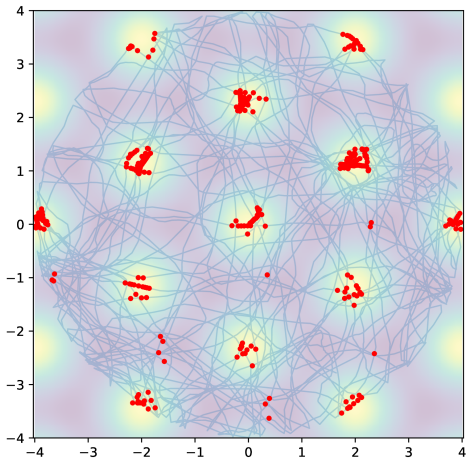

Figure 7: Grid cell. The blue line shows the trajectory of a simulated mouse running around in a pen for 16 minutes. The red dots show the spikes of a grid cell.

animal moving throughout an arena. As it moves, the spikes of one neuron are shown as red dots. That neuron is a resonate-and-fire (RF) neuron that receives spikes from 6 spiking-phasor neurons encoding position using HexSSPs (Dumont and Eliasmith, 2019), a strategy for choosing hypervectors so that their phases align at locations on a hexagonal grid (hence the name). The RF neuron behaves like a grid cell because it only fires when the spikes coming from the 6 incoming connections are synchronized.

These grid cells, which are used to represent state spaces in biological models, have also been employed as a basis for path integration that unifies symbolic representation and neural implementation (Dumont et al., 2022). Furthermore, they have been used to represent state spaces that decrease the time to convergence of reinforcement learning agents (Bartlett et al., 2022). It is worth noting that these models of biological representations of state spaces (*e.g.*, position in space, head direction cells; Sargolini et al., 2006; Langston et al., 2010) are proving useful in more traditional ML settings.

3.3 Function representation

A function can be represented by a vector in a VSA using a bundle. Consider a scalar function $f : [-L, L] \to \mathbb{R}$. We can encode a single sample of that function, $(x, f(x))$, by encoding its ordinate as a hypervector, X^x, and encoding its abscissa as a hypervector, $F^{f(x)}$. The sample is then represented by binding those two hypervectors, $X^x \otimes F^{f(x)}$. If we do that for many samples, $(x_k, f(x_k)), k = 1, \ldots, K$, then we can represent the entire function using the bundle,

$$\hat{V} = \sum_k X^{x_k} \otimes F^{f(x_k)} \,. \tag{6}$$

Finally, in order to encode the vector in a population of spiking-phasor neurons, the vector $\hat{V}$ has to be converted to a unitary vector, V, by setting the modulus of each of

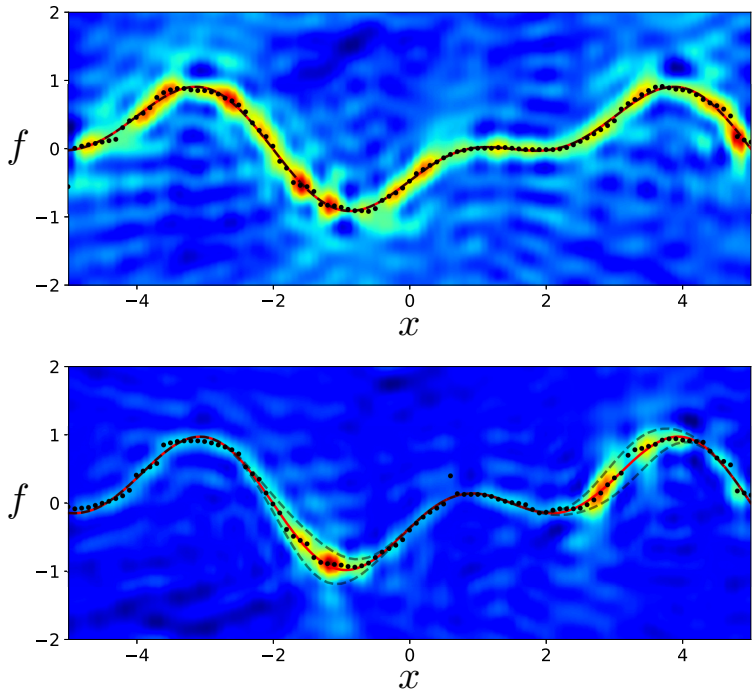

Figure 8: Function bundles. The red line shows the true function, the black dots show the values decoded from the bundle, and the heatmap in the background shows the similarity of the bundle to corresponding hypervectors $X^x \otimes F^f$. Top: deterministic function. Bottom: stochastic function, where the dashed lines show ± 2 standard deviations.

its elements to 1. This normalization operation results in loss of information. However, the vector still maintains important aspects of the function (as we will demonstrate).

Figure 8 (top) shows an example of a randomly-generated (band-limited) function (red line), as well as an illustration of its encoding into a single vector with 8640 elements. The function was sampled uniformly along its arc-length, resulting in approximately 1000 samples, all encoded and bundled using (6). The vector V was encoded into a population of 8640 spiking-phasor neurons firing at 10 Hz. Based on (Dumont and Eliasmith, 2019), we used HexSSPs in which the phases for the hypervectors X and F form a polar sampling of 2D space, with 5° angular increments, and 120 samples at each angle, covering a radius of π. The heat map in the figure shows

$$V \odot \left(X^x \otimes F^f\right), \quad \text{for } \; x \in [-5, 5], \; f \in [-2, 2]\,,$$

the similarity between the vector V and a grid of hypervectors that uniformly sample the region. For each x-value, the black dot indicates the f-value with the maximum similarity, calculated using

$$f_{\max} = \underset{f}{\operatorname{argmax}} \left[V \odot \left(X^x \otimes F^f\right)\right] \,. \tag{7}$$

The bottom pane of Fig. 8 illustrates that stochastic functions can also be represented using the same strategy. Let $\mu(x)$ be a band-limited function, and let $\sigma(x)$ be

another function that dictates the standard deviation of samples around $\mu(x)$. For each x value, 4 samples were drawn from the normal distribution $\mathcal{N}(\mu(x), \sigma(x))$ and added to the bundle. The bundle was then made into a unitary vector and encoded into a population of 8640 spiking-phasor neurons. The same method as the top plot was used to create the bottom plot. However, the dashed lines show the $\pm 2\sigma$ deviations from the mean. Notice that the bundle vector's similarity spreads out in these regions of non-zero deviation.

Others have used VSAs to represent functions (Frady et al., 2022), where they appeal to techniques developed for function representation in reproducing kernel Hilbert spaces (RKHSs). Relying on the fact that the dot product between fractionally bound values induces a kernel function, one can appeal to the representer theorem (Schölkopf et al., 2001) to show that optimal function representations exist for a set of encoded sample points.

An interesting aspect of this approach is the connection with the definition of a function. A function can be uniquely defined by the set of all pairs $(x, f(x))$ over the domain $x \in \mathcal{X}$. Similarly, we can understand bundles of observations, $X^x \otimes F^{f(x)}$ as sets of those pairs, but imbued, under the dot product, with a kernel function defined by the phasors of the FHRR encoding. Furlong and Eliasmith (2022) identified that VSA representations with continuous encodings are inherently probabilistic, and what we see here is that a natural definition of a function is also inherently probabilistic. This supports the notion that these hyperdimensional computing frameworks provide a probabilistic model for neuromorphic computing.

3.4 Integrator

An integrator is a system that receives a signal, $v(t)$, and computes the time integral of that signal, $p(t) = \int_0^t v(\tau) d\tau$. We wanted to see if we could use a spiking-phasor network to perform integration.

An hypervector, $X \in \mathbb{C}^{200}$, was generated by randomly choosing 200 phases uniformly from $(-\pi, \pi)$. A population of 200 spiking-phasor neurons encoded X by spiking at those phases. Another population of 200 phase-multiplication neurons received X and multiplied their phases by $v(t)$, thus encoding $X^{v(t)}$. The input $v(t)$ changes over time, so the phase multiplication neurons were adjusting their phases every cycle according to the value of $v(t)$. Finally, that population sent its spikes to an integrating population that also had 200 neurons. Each neuron in the integrating population simply added the phase of the incoming spike to its own phase. All the populations were firing at 10 Hz, so the integrator could only update its estimate of the integral once every 100 ms. As a result, this integration process is roughly equivalent to doing discrete integration using a rectangle quadrature rule.

We ran 20 trials of integrating random band-limited signals for 5 seconds, and compared the output of the spiking-phasor integrator (denoted the "estimate") to a numerical integrator that took 100 times as many steps (denoted "ground truth"). We measured the error between the estimate and the ground truth every 100 ms over all 20 trials. The standard deviation of the error was consistently less than 5% of the standard deviation of the ground truth signal itself. The network had a total of 601 neurons: 200 for the population that encodes X, another 200 neurons that encode $X^{v(t)}$, another 200 neurons

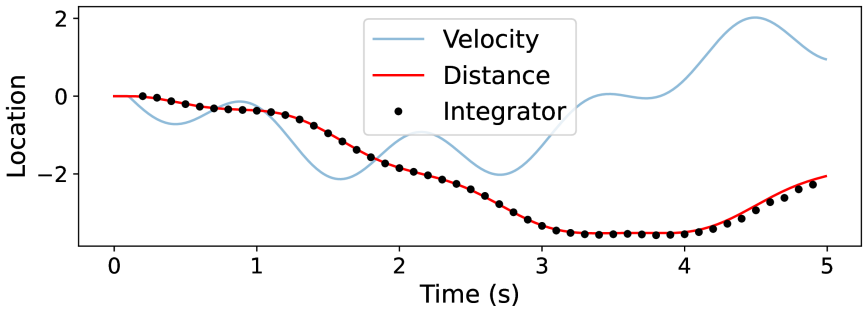

Figure 9: Integrator results. The blue line is the signal to be integrated (velocity, $v(t)$). The red line shows its ground truth integral (distance, $p(t)$). The black dots show the value encoded in the intergrator population for each 100 ms cycle (the estimate).

for the integrator population, and one more neuron that fires a single spike that resets the integrator to zero at the beginning of the run.

Figure 9 shows the result of a typical run. The velocity signal is shown in blue, and its ground truth integral ("distance") is drawn in red. The black dots show the neural integrator's estimate (at 10 Hz).

A similar type of (non-spiking) phasor-based integrator was used for visual odometry (Renner et al., 2022a). The spiking-phasor implementation presented here could be used in that work.

3.5 Legendre Delay Network

An interesting application of a neural integrator is a Legendre Delay Network (LDN). An LDN optimally encodes the recent history of a continuous-time signal into a set of time-varying coefficients. A snapshot of those coefficients can be used, in conjunction with Legendre basis polynomials, to reconstruct the previous θ seconds of the input signal (Voelker et al., 2019). Briefly, the ℓ Legendre coefficients, $\mathbf{m}(t)$, are determined by a system of ℓ differential equations of the form

$$\theta \dot{\mathbf{m}}(t) = A\mathbf{m}(t) + Bu(t)$$

where $A \in \mathbb{R}^{\ell \times \ell}$ and $B \in \mathbb{R}^{\ell}$ are constant, and $u(t)$ is the input signal to be recorded.

The LDN is foundational to the Legendre Memory Unit (LMU). The LMU is one of the earliest examples of what are now known as *state space models* – recurrent networks that can model long-range dependencies without the quadratic complexity of transformers' attention. LMUs have achieved comparable or better performance to transformers, with 2×-10× improvements in the number of parameters (Chilkuri et al., 2021; Chilkuri and Eliasmith, 2021). Other state space models have demonstrated improvements over transformers (Gu et al., 2021, 2022; Smith et al., 2022; Gu and Dao, 2023).

We constructed a spiking-phasor Legendre Delay Network by connecting a number of integrator populations (described in section 3.4). We used hypervectors of dimension 100. A population of 100 phase multiplication neurons was used to encode the input signal $u(t)$ using $X^{u(t)}$. The LDN itself consisted of 10 integrator populations, each

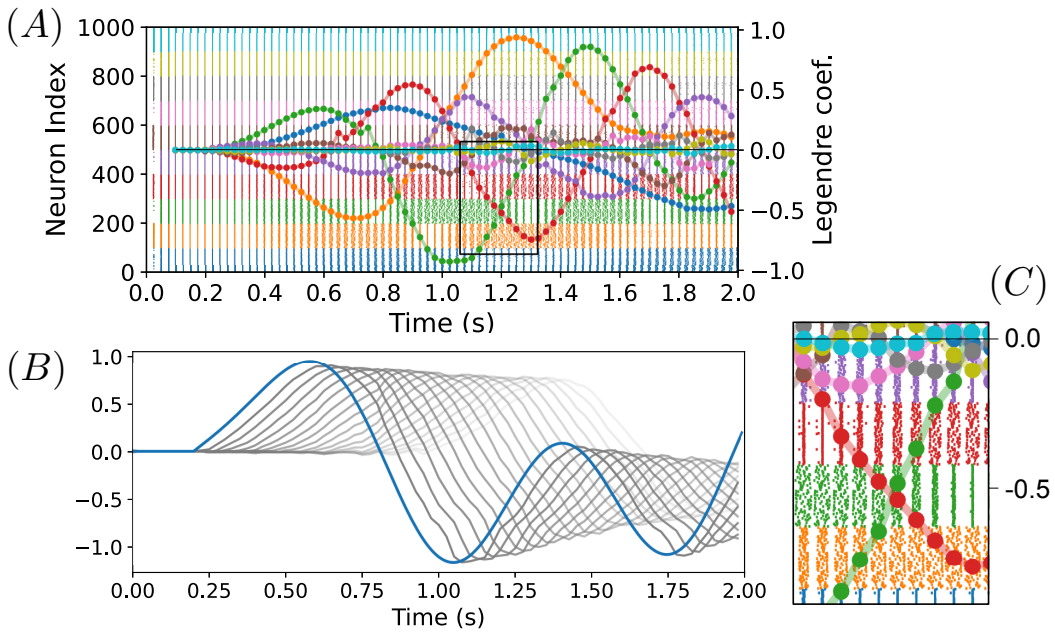

Figure 10: Legendre Delay Network results. (A) shows the spike raster plots for the 10 integrating populations (of 100 neurons each), as well as the decoded Legendre coefficients, $\mathbf{m}(t)$. The colour of the spikes matches the line for the corresponding Legendre coefficient. (B) shows the corresponding input signal (blue), as well as reconstructions with varying delays; the lighter the curve, the longer the delay. (C) shows an enlarged view of the region outlined in (A). Notice that as the green dot-line climbs toward zero (the black axis), the horizontal distribution of green spikes reduces, and as the red dot-line gets further from the axis, the red spikes disperse.

with 100 neurons, encoding hypervectors L_0 through L_9. We found that accuracy was greatly improved if we used a higher frequency, so this network cycled at 40 Hz. The entire network had 1201 neurons: 100 for the population that generates the hypervector X, another 100 neurons that encode the input signal $X^{u(t)}$, 1000 for the LDN integrators (encoding $L_0, \ldots, L_9$), and one more neuron that fires a single spike that resets all the LDN integrators to zero at the beginning of the run.

To visualize the encoding, we first had to decode the coefficients from the LDN integrator populations. This was done using $m_i = \text{argmax}_m \left(L_i \odot X^m\right)$.

Figure 10 illustrates a typical run of the LDN. Panel (A) shows the spiking activity of all 10 LDN integrator populations (organized by colour) beneath plots of the estimated Legendre coefficients (colour-matched to the spike rasters). Panel (B) uses those coefficients and reconstructs the input signal at a variety of different delays, with longer delays rendered with progressively lighter greys. Panel (C) shows a close-up of the delineated region of the raster plot.

4 Discussion

Our focus on spiking implementations of VSAs is to enable their implementation on neuromorphic hardware, where the largest gains in energy efficiency are realized by spiking networks. Furthermore, we concentrate on HRR and FHRR vector symbolic architectures because they have been shown to be among the highest capacity methods (Schlegel et al., 2022), and they can support fractional binding.

4.1 Model Complexity

How many arithmetic operations ("ops") does it take to perform a binding operation? Let us consider a single phase-sum neuron, and assume that our implementation takes T times steps per cycle. Note that if the time variable is scaled so that Δt is one time step on the hardware, then there is no need to multiply by Δt in Algorithm 1. Thus, in each time step, the model requires up to 3 ops to increment the integrators: one addition to increment c, one conditional (i.e. $\dot{q} > 0$), and one addition to (conditionally) increment q. There are also 2 other conditionals per time step: $q < 0$ for spiking, and $c > L$ for end-of-cycle. Putting those all together, each time step requires at most 5 ops.

There are also some processes that (usually) happen once per cycle. When a spike arrives, it incurs up to 3 ops: 1 conditional (i.e. if $\dot{q} > 0$), another to (conditionally) assign $q \leftarrow c$, and 1 to decrement $\dot{q}$. If the neuron spikes, it causes 2 ops, and when the cycle resets, it costs 1 op.

Hence, the operation count for a single phase-sum neuron for one cycle is $5T + 7$. It can take 2 cycles for the computation to settle, so the real operation count would be a small multiple of $5T + 7$. If we assume that our hypervectors are of dimension d, then the computational complexity class for a single cycle is $\mathcal{O}(dT)$.

4.2 Potential for Neuromorphic Speedup

One advantage of spiking neural network models is that they are implementable on neuromorphic systems. Assuming that the neuron model can run in parallel on the neuromorphic chip, then we can remove d from the time complexity. In other words, binding can be done in $\mathcal{O}(T)$ ops. In our experiments, we used time steps of 0.1ms, and cycle lengths between 0.025s and 0.2s. This gives us T-values in the range 250 to 2000.

The computation involved in the neuron models is relatively simple. After the initial set-up of the network, they do not use any transcendental functions like $\sin$, $\log$, or $\exp$. The arithmetic is within a limited range, typically between -1 and 1, which places fewer demands on the precision of the number system. A fixed-point number system would probably suffice for these computations. Furthermore, most of the operations required in the neuron models can be done using addition and conditionals. For example, the multiplications in Algorithm 1 can be implemented by a conditional. There are only 2 models that require multiplication. One is the fractional binding (phase multiplication) model, which needs to compute the product $\alpha\hat{c}$ to set the threshold. However, that multiplication is only done once per cycle. The other model is the oscillators in the clean-up memory, governed by the differential equations (1) and (2).

4.3 Connection Efficiency

One of the main benefits of the spiking-phasor approach to FHRR is the sparseness of connections. Take as an example the binding operation between two vectors. Each vector is encoded by a population of spiking phasors; let us denote them as A and B. Both populations, A and B, project to a phase-sum population, C. If the vectors are d-dimensional, then each population (A, B, and C) has d neurons, for a total of $3d$ neurons. But the whole network has only $2d$ connections, since each neuron in C has only 2 incoming connections, one from its counterpart in A, and one from its counterpart in B. This simplified connectivity affords even more parallelization, whereby each Fourier coefficient essentially forms an independent channel through the network. Even the permutation operation maintains channelization. The only place that this "channelization" breaks down is in our clean-up memory, where the connections between layers in the Hopfield-like network are all-to-all. However, in a network encoding d-dimensional vectors, and a vocabulary of m vectors, the number of connections in the Hopfield network is only $2dm + m^2$.

Another approach to building spiking neural networks that implement a VSA is to use the Neural Engineering Framework (NEF) (Eliasmith and Anderson, 2003). However, the NEF implementation does not have the same benefit of channelization because its default VSA implementation is HRR (Komer et al., 2019), and uses a DFT and IDFT to implement the circular convolution for binding. The DFT and IDFT are dense transformations, requiring all-to-all connectivity. Moreover, each of the vector elements is encoded by a *population* of neurons. That is, each of the d-dimensional input vectors (a and b) is encoded using Nd leaky integrate-and-fire (LIF) neurons, with N neurons encoding each of the d vector elements. The process of circular convolution proceeds as follows: (1) The values of the vectors a and b are decoded from their populations (requiring $2Nd$ connections), (2) then Fourier transformed ($\mathcal{O}(d^2)$ connections), and (3) fed into an array of roughly $4d$ populations (with $\frac{N}{2}$ neurons each) in preparation for element-wise multiplication ($\mathcal{O}(Nd)$ connections). From there, (4) decoding from those populations takes another $\mathcal{O}(Nd)$ connections, followed by (5) an inverse Fourier transform ($\mathcal{O}(d^2)$ connections). The whole process of binding two d-dimensional vectors involves $\mathcal{O}(Nd)$ neurons, and $\mathcal{O}(d^2 + Nd)$ connections. See Fig. 6 in (Bekolay et al., 2014) for a detailed description of circular convolution in Nengo, the software library implementation of the NEF.

For a concrete comparison, consider binding two vectors of dimension 100. Using the NEF with $N = 50$, it would take about 20,000 LIF neurons, and optimistically take 150,000 connection weights (assuming that low-rank connection-weight matrices can be implemented as a sequence of connections through "phantom" nodes). However, the same binding network using spiking phasors would take only 300 neurons, and 200 connection weights. Or, using vectors of dimension $d = 480$ (like for the spatial-memory network in section 3.2), the NEF would involve 96,000 neurons and 2,908,800 connections, while the spiking-phasor implementation used 1,440 neurons, and 960 connections. It should be noted that spiking-phasor neurons are more complex than LIF neurons. However, the arithmetic operations carried out inside these spiking-phasor neurons is relatively simple, requiring linear integrators, a value copy, and simple logic.

4.4 Silicon Costs

Purely algebraic implementations of VSA algorithms, in and of themselves, can provide computational benefits compared to traditional algorithm formulations, as surveyed by Kleyko et al. (2023). Similarly, they have been proposed as a framework for programming neuromorphic computing (Kleyko et al., 2021).

However, due to the intermingling of memory and processing that is typical in neuromorphic computers, it is difficult to exploit cheaper memory hardware, like that found in the memory hierarchies of von Neumann computers. For example, the Loihi processor relies on SRAM to store state variables with neurons, which is 100 times more expensive than the DRAM that is used to provide off-chip memory in traditional computers (Davies et al., 2021; Davies, 2021). This high cost has spurred researchers to explore alternative memory technologies, but these cheaper memories come with problems in variability and reliability that need to be resolved (Rajendran and Alibart, 2016; Adam et al., 2018).

Until those problems can be suitably mitigated, there is a pressing need to reduce the number of neurons required in a given network implementation. Consequently, this makes our spiking phasor implemenation of VSA algorithms all the more timely – it permits one to take advantage of the benefits of VSA algorithms and neuromorphic computers while not unnecessarily consuming on-chip resources. This means that one can either fit more VSA algorithms on one chip of a fixed capacity, or one can design smaller chips with fewer neurons, reducing costs per chip while still being able to implement VSA-style algorithms.

5 Conclusions and Future Work

We have shown how a relatively simple set of spiking neuron models can perform all the basic operations of an FHRR. The spiking-phasor implementation behaves just like its complex-valued counterpart. Numerical experiments using complex-valued hypervectors (not shown) often yielded indistinguishable results from our spiking-phasor implementation. The differences between the two emerged in the context of dynamic scenarios. The discrete-time nature of the spiking-phasors – the fact that a cycle was required to encode a phase angle – resulted in inaccuracies in the integrators. Choosing a higher firing rate (like 40 Hz) helped in those situations.

These methods all use local information only and, for the most part, involve a small number of synaptic connections; only 2 neurons synapse onto 1 neuron for phase summation and subtraction, and neurons are connected 1-to-1 for phase multiplication. The denser synaptic connections are in the Hopfield clean-up memory, which involves all-to-all d-to-m connections for d-dimensional hypervectors and m vocabulary vectors, and in the LDN connections, which involve ℓ-ℓ connections for ℓ Legendre coefficients.

The spiking nature of these networks, and the fact that the connection structure tends to be sparse, makes this spiking-phasor framework an excellent candidate for neuromorphic implementation. Indeed, a similar implementation of spiking-phasor binding was demonstrated on Intel's Loihi chip (Renner et al., 2022b). However, this paper extends that work with an unbinding neuron model, a fractional binding neuron model, and

a Hopfield-like clean-up memory, and demonstrates their capabilities on a number of foundational algorithms, such as spatial memory, function representation, signal integration, and signal replay.

This work raises a number of follow-up investigations. The sensitivity of these methods to random jitter in the spike times should be studied. To what extent do small disturbances in the spike times disrupt the overall functioning of these networks? This is a topic for immediate future work. Also, a neuromorphic implementation of the spiking-phasor FHRR methods – even in emulation mode – would allow us to study the impact of reduced precision floating-point or fixed-point number systems. It would also be enlightening to do a detailed analysis of the expected execution speed on neuromorphic hardware. How do the relative costs of internal neural dynamics, spike delivery, and neuron count affect the speed and power efficiency on a given neuromorphic platform? Finally, it would be valuable to devise a method for synchronizing the start/end of the cycles. With the current solution – every neuron tracking its own cycle using its clock variable – might be prone to drift. We plan to investigate mechanisms for synchronization across many spiking-phasor neurons.

Acknowledgements

This work was supported by the Natural Sciences and Engineering Research Council (NSERC) of Canada, and the National Research Council (NRC) of Canada. PMF was additionally funded by AFOSR grant FA9550-23-1-0644.

References

Adam, G. C., Khiat, A., and Prodromakis, T. (2018). Challenges hindering memristive neuromorphic hardware from going mainstream. *Nature communications*, 9(1):5267.

Bartlett, M., Stewart, T. C., and Orchard, J. (2022). Biologically-based neural representations enable fast online shallow reinforcement learning. In *Proceedings of the Annual Meeting of the Cognitive Science Society*, volume 44.

Bekolay, T., Bergstra, J., Hunsberger, E., DeWolf, T., Stewart, T. C., Rasmussen, D., Choo, X., Voelker, A. R., and Eliasmith, C. (2014). Nengo: A Python tool for building large-scale functional brain models. *Frontiers in Neuroinformatics*, 7.

Bent, G., Simpkin, C., Li, Y., and Preece, A. (2022). Hyperdimensional computing using time-to-spike neuromorphic circuits. In *Proceedings of the International Joint Conference on Neural Networks*, pages 1–8.

Chilkuri, N., Hunsberger, E., Voelker, A., Malik, G., and Eliasmith, C. (2021). Language modeling using LMUs: 10x better data efficiency or improved scaling compared to transformers. *arXiv:2110.02402*.

Chilkuri, N. R. and Eliasmith, C. (2021). Parallelizing Legendre memory unit training. In *International Conference on Machine Learning*, pages 1898–1907.

Davies, M. (2021). Lessons from Loihi: Progress in neuromorphic computing. In *2021 Symposium on VLSI Circuits*, pages 1–2.

Davies, M., Wild, A., Orchard, G., Sandamirskaya, Y., Guerra, G. A. F., Joshi, P., Plank, P., and Risbud, S. R. (2021). Advancing neuromorphic computing with Loihi: A survey of results and outlook. *Proceedings of the IEEE*, 109(5):911–934.

Dumont, N. and Eliasmith, C. (2019). Accurate representation for spatial cognition using grid cells. In *Joint International Conference on Cognitive Science*.

Dumont, N. S.-Y., Orchard, J., and Eliasmith, C. (2022). A model of path integration that connects neural and symbolic representation. In *Proceedings of the Annual Meeting of the Cognitive Science Society*, volume 44.

Eliasmith, C. and Anderson, C. H. (2003). *Neural Engineering: Computation, Representation, and Dynamics in Neurobiological Systems*. MIT Press.

Frady, E. P., Kanerva, P., and Sommer, F. T. (2018). A framework for linking computations and rhythm-based timing patterns in neural firing, such as phase precession in hippocampal place cells. In *Conference on Cognitive Computational Neuroscience*, pages 1–5.

Frady, E. P., Kleyko, D., Kymn, C. J., Olshausen, B. A., and Sommer, F. T. (2021). Computing on functions using randomized vector representations. *arXiv:2109.03429*.

Frady, E. P., Kleyko, D., Kymn, C. J., Olshausen, B. A., and Sommer, F. T. (2022). Computing on functions using randomized vector representations (in brief). In *ACM International Conference Proceeding Series*, pages 115–122.

Frady, E. P. and Sommer, F. T. (2019). Robust computation with rhythmic spike patterns. *Proceedings of the National Academy of Sciences of the United States of America*, 116:18050–18059.

Furlong, M. and Eliasmith, C. (2022). Fractional binding in vector symbolic architectures as quasi-probability statements. In *Proceedings of the Annual Meeting of the Cognitive Science Society*, volume 44.

Gayler, R. W. (2003). Vector symbolic architectures answer Jackendoff's challenges for cognitive neuroscience. In *Joint International Conference on Cognitive Science*, pages 133–138.

Gu, A. and Dao, T. (2023). Mamba: Linear-time sequence modeling with selective state spaces. *arXiv:2312.00752*.

Gu, A., Goel, K., and Re, C. (2021). Efficiently modeling long sequences with structured state spaces. In *International Conference on Learning Representations*.

Gu, A., Gupta, A., Goel, K., and Ré, C. (2022). On the parameterization and initialization of diagonal state space models. *arXiv:2206.11893*.

Kanerva, P. (2009). Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors. *Cognitive computation*, 1:139–159.

Kleyko, D., Davies, M., Frady, E. P., Kanerva, P., Kent, S. J., Olshausen, B. A., Osipov, E., Rabaey, J. M., Rachkovskij, D. A., Rahimi, A., et al. (2021). Vector symbolic architectures as a computing framework for nanoscale hardware. *arXiv:2106.05268*.

Kleyko, D., Rachkovskij, D., Osipov, E., and Rahimi, A. (2023). A survey on hyperdimensional computing aka vector symbolic architectures, part ii: Applications, cognitive models, and challenges. *ACM Computing Surveys*, 55(9):1–52.

Komer, B., Stewart, T. C., and Eliasmith, C. (2019). A neural representation of continuous space using fractional binding. In *Proceedings of the 41st Annual Meeting of the Cognitive Science Society*.

Krotov, D. and Hopfield, J. (2021). Large associative memory problem in neurobiology and machine learning. In *ICLR*, pages 1–12.

Laiho, M., Poikonen, J. H., Kanerva, P., and Lehtonen, E. (2015). High-dimensional computing with sparse vectors. In *IEEE Biomedical Circuits and Systems Conference: Engineering for Healthy Minds and Able Bodies, BioCAS 2015 - Proceedings*.

Langston, R. F., Ainge, J. A., Couey, J. J., Canto, C. B., Bjerknes, T. L., Witter, M. P., Moser, E. I., and Moser, M.-B. (2010). Development of the spatial representation system in the rat. *Science*, 328(5985):1576–1580.

Lu, T., Voelker, A. R., Komer, B., and Eliasmith, C. (2019). Representing spatial relations with fractional binding. In *Proc. of the Cognitive Science Society*.

Noest, A. J. (1988). Phasor neural networks. In Anderson, D., editor, *Neural Information Processing Systems 1987*, pages 584–591.

Orchard, J. and Jarvis, R. (2023). Hyperdimensional computing with spiking-phasor neurons. In *Proceedings of the 2023 International Conference on Neuromorphic Systems*, ICONS '23, New York, NY, USA. Association for Computing Machinery.

Orchard, J., Yang, H., and Ji, X. (2013). Does the entorhinal cortex use the Fourier transform? *Frontiers in computational neuroscience*, 7.

Plate, T. A. (1995). Holographic reduced representations. *IEEE Transactions on Neural Networks*, 6:623–641.

Plate, T. A. (2003). *Holographic Reduced Representation: Distributed Representation for Cognitive Structures*. CSLI Publications.

Rajendran, B. and Alibart, F. (2016). Neuromorphic computing based on emerging memory technologies. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 6(2):198–211.

Renner, A., Supic, L., Danielescu, A., Indiveri, G., Frady, E. P., Sommer, F. T., and Sandamirskaya, Y. (2022a). Neuromorphic visual odometry with resonator networks. *arXiv:2209.02000*.

Renner, A., Supic, L., Danielescu, A., Indiveri, G., Olshausen, B. A., Sandamirskaya, Y., Sommer, F. T., and Frady, E. P. (2022b). Neuromorphic visual scene understanding with resonator networks. *arXiv:2208.12880*.

Sargolini, F., Fyhn, M., Hafting, T., McNaughton, B. L., Witter, M. P., Moser, M.-B., and Moser, E. I. (2006). Conjunctive representation of position, direction, and velocity in entorhinal cortex. *Science*, 312(5774):758–762.

Schlegel, K., Neubert, P., and Protzel, P. (2022). A comparison of vector symbolic architectures. *Artificial Intelligence Review*, 55:4523–4555.

Schölkopf, B., Herbrich, R., and Smola, A. J. (2001). A generalized representer theorem. In *Conference on Computational Learning Theory, COLT 2001*, pages 416–426.

Smith, J. T., Warrington, A., and Linderman, S. W. (2022). Simplified state space layers for sequence modeling. *arXiv preprint arXiv:2208.04933*.

Stewart, T. C., Tang, Y., and Eliasmith, C. (2011). A biologically realistic cleanup memory: Autoassociation in spiking neurons. *Cognitive Systems Research*, 12:84–92.

Stimberg, M., Brette, R., and Goodman, D. F. (2019). Brian 2, an intuitive and efficient neural simulator. *eLife*, 8:e47314.

Voelker, A., Kajić, I., and Eliasmith, C. (2019). Legendre memory units: Continuous-time representation in recurrent neural networks. *Advances in neural information processing systems*, 32.

Welday, A. C., Shlifer, I. G., Bloom, M. L., Zhang, K., and Blair, H. T. (2011). Cosine directional tuning of theta cell burst frequencies: Evidence for spatial coding by oscillatory interference. *Journal of Neuroscience*, 31:16157–16176.