LETTER **Communicated by Peer Neubert**

Simulating and Predicting Dynamical Systems With Spatial Semantic Pointers

Aaron R. Voelker
arvoelke@uwaterloo.ca
Peter Blouw
peter.blouw@appliedbrainresearch.com
Xuan Choo
xuan.choo@appliedbrainresearch.com
Applied Brain Research, Waterloo, ON N2L 3G1, Canada

Nicole Sandra-Yaffa Dumont
ns2dumont@uwaterloo.ca
Cheriton School of Computer Science, University of Waterloo, Waterloo, Ontario, N2L 3G1, Canada

Terrence C. Stewart
terrence.stewart@nrc-cnrc.gc.ca
National Research Council of Canada, University of Waterloo Collaboration Centre, Waterloo, ON N2L 3G1 Canada

Chris Eliasmith
celiasmith@uwaterloo.ca
Centre for Theoretical Neuroscience, University of Waterloo, Waterloo, Ontario, N2L 3G1, Canada

While neural networks are highly effective at learning task-relevant representations from data, they typically do not learn representations with the kind of symbolic structure that is hypothesized to support high-level cognitive processes, nor do they naturally model such structures within problem domains that are continuous in space and time. To fill these gaps, this work exploits a method for defining vector representations that bind discrete (symbol-like) entities to points in continuous topological spaces in order to simulate and predict the behavior of a range of dynamical systems. These vector representations are spatial semantic pointers (SSPs), and we demonstrate that they can (1) be used to model dynamical systems involving multiple objects represented in a symbol-like manner and (2) be integrated with deep neural networks to predict the future of

A.V. and P.B. contributed equally.

Neural Computation **33, 2033–2067** (2021)
https://doi.org/10.1162/neco_a_01410

physical trajectories. These results help unify what have traditionally appeared to be disparate approaches in machine learning.

1 Introduction

A considerable amount of recent progress in AI research has been driven by the fact that artificial neural networks are highly effective function approximators when trained with sufficient amounts of data. However, it is generally acknowledged that there remain important aspects of intelligent behavior that are not naturally described by static functions applied to discrete sets of inputs. For instance, LeCun, Bengio, and Hinton (2015) have decried the lack of methods for combining representation learning with complex reasoning (see also Bottou, 2014), an avenue of research that has traditionally motivated researchers to posit the need for structured symbolic representations (Marcus, 1998; Smolensky & Legendre, 2006; Hadley, 2009). Others have noted such methods do not effectively capture the dynamics of cognitive information processing in continuous space and time (Eliasmith, 2013; Schöner, 2014). Consequently, extending neural networks to manipulate structured symbolic representations in task contexts that involve dynamics over continuous space and time is an important unifying goal for the field.

In this work, we take a step toward this goal by exploiting a method for defining vector representations that encode blends of continuous and discrete structures in order to simulate and predict the behavior of a range of dynamical systems in which multiple objects move continuously through space and time. These vector representations are spatial semantic pointers (SSPs), and we provide analyses of both their capacity to represent complicated spatial topographies and their ability to learn and model arbitrary dynamics defined with respect to these topographies. More specifically, we show how SSPs can be used to (1) simulate continuous trajectories involving multiple objects, (2) simulate interactions between these objects and walls, and (3) learn the dynamics governing these interactions in order to predict future object positions.

Mathematically, SSPs are built on the concept of a vector symbolic architecture (VSA; Gayler, 2004), in which a set of algebraic operations is used to bind vector representations into role-filler pairs and to group such pairs into sets (Smolensky, 1990; Plate, 2003; Kanerva, 2009; Frady, Kleyko, & Sommer, 2020; Schlegel, Neubert, & Protzel, 2020). Traditionally, VSAs have been characterized as a means of capturing symbol-like discrete representational structures using vector spaces. Recent extensions to VSAs have introduced fractional binding operations that define SSPs as distributed representations in which both roles and fillers can encode continuous quantities (Komer, Stewart, Voelker, & Eliasmith, 2019; Frady, Kanerva, & Sommer, 2018). SSPs have previously been used to model spatial reasoning

tasks (Lu, Voelker, Komer, & Eliasmith, 2019; Weiss, Cheung, & Olshausen, 2016), model path planning and navigation (Komer & Eliasmith, 2020), and model grid cell and place cell firing patterns in the context of spiking neural networks (Dumont & Eliasmith, 2020). Storage capacity analyses with SSPs have also been performed (Mirus, Stewart, & Conradt, 2020). Here, we extend this prior work to model continuous dynamical systems with SSPs and thereby integrate perspectives on AI that alternatively focus on deep learning (LeCun, Bengio, and Hinton, 2015; Goodfellow, Bengio, & Courville, 2016; Schmidhuber, 2015), symbolic structure (Marcus, 1998, 2019; Hadley, 2009), and temporal dynamics (Eliasmith, 2013; Voelker, 2019; McClelland et al., 2010).

We begin by introducing VSAs and fractional binding. We then use these concepts to define SSPs and discuss methods for visualizing them. We discuss their relevance for neurobiological representations (i.e., grid cells) and feature representation in deep learning. After this introductory material, we turn to our contributions, which expose new methods for representing and learning arbitrary trajectories in neural networks. We then demonstrate how arbitrary trajectories can be simulated dynamically for both single and multiple objects. Next, we derive partial differential equations (PDEs) that can simulate continuous-time trajectories by way of linear transformations embedded in recurrent spiking neural networks. We also introduce two methods for simulating multiple objects, and compare and contrast them. Finally, we show how SSPs can be combined with Legendre memory units (LMUs; Voelker, Kajić, & Eliasmith, 2019) to predict the future trajectory of objects moving along paths with discontinuous changes in motion.

2 Structured Vector Representations

2.1 Vector Symbolic Architectures. Vector symbolic architectures (VSAs) were developed in the context of long-standing debates surrounding the question of how symbolic structures might be encoded with distributed representations of the sort manipulated by neural networks (Fodor & Pylyshyn, 1988; Marcus, 1998; Smolensky & Legendre, 2006). To provide an answer to this question, a VSA first defines a mapping from a set of primitive symbols to a set of vectors in a d-dimensional space, $\mathcal{V} \subseteq \mathbb{R}^d$. This mapping is often referred to as the VSA's "vocabulary." Typically the vectors in the vocabulary are chosen such that by the similarity measure used in the VSA, each vector is dissimilar and thus distinguishable from every other vector in the vocabulary. A common method for generating random d-dimensional vectors involves sampling each element from a normal distribution with a mean of 0 and a variance of $1/d$ (Muller, 1959). Choosing vector elements in this way ensures both that the expected L2-norm of the vector is 1 and that performing the discrete Fourier transform (DFT) on the vector results in Fourier coefficients uniformly distributed around 0 with identical variance across all frequency components.

Additionally, VSAs define algebraic operations that can be performed on vocabulary items to enable symbol manipulation. These operations can be grouped into five types: (1) a similarity operation that computes a scalar measure of how alike two vectors are; (2) a collection operation that combines two vectors into a new vector that is similar to both inputs; (3) a binding operation that compresses two vectors into a new vector that is dissimilar to both inputs; (4) an inverse operation that decompresses a vector to undo one or more binding operations; and (5) a cleanup operation that maps a noisy or decompressed vector to the most similar "clean" vector in the VSA vocabulary. We discuss each of these operations in turn and focus on a specific VSA that uses vector addition for collection and circular convolution for binding, and whose vectors are commonly referred to as holographic reduced representations (HRRs; Plate, 2003):

Similarity. To measure the similarity (s) between two vectors, VSAs typically use the inner product operation in Euclidean space (a.k.a., the dot product):

$$s = A \cdot B = A^{\mathrm{T}} B. \tag{2.1}$$

When the two vectors are unit length, this becomes identical to the "cosine similarity" measure. When this measure is used, two identical vectors have a similarity of 1, while two orthogonal vectors will have a similarity of 0. We note that when the dimensionality, d, is large, then two randomly generated unit vectors are expected to be approximately orthogonal, or dissimilar, to one another (Gosmann, 2018).

Collection. A collection operation is defined to map any pair of input vectors to an output vector that is similar to both inputs. This is useful for representing unordered sets of symbols. Vector superposition (i.e., element-wise addition) is commonly used to implement this operation.

Binding. A binding operation is defined to map any pair of input vectors to an output vector that is dissimilar to both input vectors. This is useful for representing the conjunction of multiple symbols. Common choices for a binding operation include circular convolution (Plate, 2003), element-wise multiplication (Gayler, 2004), vector-derived transformation binding (Gosmann & Eliasmith, 2019), and exclusive-or (Kanerva, 2009), though some of these choices impose requirements on the vocabularly vectors they apply to. With circular convolution ($\circledast$), the binding of A and B can be efficiently computed as

$$A \circledast B = \mathcal{F}^{-1}\{\mathcal{F}\{A\} \odot \mathcal{F}\{B\}\}, \tag{2.2}$$

where $\mathcal{F}\{\cdot\}$ is the DFT operator and $\odot$ denotes the element-wise multiplication of two complex vectors. Together, collection via addition and binding via circular convolution obey the algebraic laws of commutativity, associativity, and distributivity (Gosmann, 2018).

Additionally, because circular convolution produces an output vector for which each element is a dot product between one input vector and a permutation of the elements in the other input vector, it possible to permute the elements of A to construct a fixed "binding matrix," $T(A) \in \mathbb{R}^{d \times d}$ that can be used to implement this binding operation (Plate, 2003):

$$A \circledast B = T(A)B. \tag{2.3}$$

More specifically, $T(A)$ is a special kind of matrix called a "circulant matrix" that is fully specified by the vector A. Its first column is A, and remaining columns are cyclic permutations of A with offset equal to the column index.

Inverse. The inverse, or "unbinding," operation can be thought of as creating a vector that undoes the effect of a binding operation, such that if $\sim A$ is the inverse of A, then binding $\sim A$ to a vector that binds together A and B will return B. For a VSA that uses circular convolution, the inverse of a vector is calculated by computing the complex conjugate of its Fourier coefficients. Interestingly, performing the complex conjugate in the Fourier domain is equivalent to performing an involution operation[1] on the individual elements of the vector. Since the exact inverse of a vector must take into account the magnitude of its Fourier coefficients, while the complex conjugate does not, the inverse operation in general only computes an approximate inverse of the vector, that is, $\sim A \approx A^{-1}$, where A^{-1} is the exact inverse of A.[2]

Cleanup. When the inverse operation is approximate, binding a vector with its inverse introduces noise into the result. Performing the binding operation on a vector that has collected together multiple other vectors also introduces potentially unwanted output terms[3]

[1] The involution operation preserves the order of the first element in a vector and reverses the order of the remaining elements. As an example, the involution of the vector [0, 1, 2, 3] is [0, 3, 2, 1].

[2] Computing the exact inverse of a vector can occasionally result in large Fourier coefficient magnitudes, since the Fourier coefficients of the input vector are uniformly distributed around 0. The large Fourier coefficient magnitudes consequently generate vectors with "misbehaved" vector magnitudes (i.e., they do not conform to the assumption that the vector magnitudes should be approximately 1).

[3] An algebraic analogy to these extraneous symbolic terms is to consider using $(a + b)^2$ to compute $a^2 + b^2$. In this analogy, the expanded form of $(a + b)^2$ contains the "desired" a^2 and b^2 terms and an "extraneous" $2ab$ term.

into the symbolic computation. As a result, VSAs define a cleanup operation that can be used to reduce the noise accumulated through the application of binding and unbinding operations and to remove unwanted vector terms from the results of these operations. To perform a cleanup operation, an input vector is compared to all the vectors within a desired vocabulary, with the output of the cleanup operation being the vector within the vocabulary that has the highest similarity to the input vector. This operation can be learned from data using a deep neural network (Komer & Eliasmith, 2020) or implemented directly by combining a matrix multiplication (to compute the dot product) with a thresholding function (Stewart, Tang, & Eliasmith, 2011) or a winner-take-all mechanism (Gosmann, Voelker, & Eliasmith, 2017).

Until recently, VSAs have been largely used to map discrete structures into high-dimensional vector spaces using slot-filler representations created through the application of binding and collection operations. Such representations are quite general and capture a variety of data types familiar to neural and cognitive modelers, including lists, trees, graphs, grammars, and rules. However, there are many natural tasks for which discrete representational structures are not appropriate. Consider the example of an agent moving through an unstructured spatial environment (e.g., a forest). Ideally, the agent's internal representations of the environment would be able to incorporate arbitrary objects (e.g., notable trees or rocks) while binding these objects to arbitrary spatial locations or areas. To implement such representations with a VSA, the slots should ideally be continuous (i.e., mapping to continuous spatial locations) even if the fillers are not. Continuous slots of this sort would allow for representations that bind specific objects (e.g., a symbol-like representation of a large oak tree) to particular spatial locations. To develop this kind of continuous spatial representation, it is useful to exploit certain additional properties of VSAs that use circular convolution as binding operator.

2.1.1 Unitary Vectors. Within the set of vectors operated on by circular convolution, there exists a subset of "unitary" vectors (Plate, 2003) that exhibit the following two properties: their L2-norms are exactly 1, and the magnitudes of their Fourier coefficients are also exactly 1. Importantly, these properties ensure that (1) the approximate inverse of a unitary vector is equal to its exact inverse, hence we can use $A^{-1} = {\sim}A$ interchangeably, (2) the dot product between two unitary vectors becomes identical to their cosine similarity, and (3) binding one unitary vector with another unitary vector results in yet another unitary vector; hence, unitary vectors are "closed" under binding with circular convolution.

Since the approximate inverse is exact for these vectors, binding a unitary vector with its inverse does not introduce noise in the result. Thus,

unitary vectors support lossless binding and unbinding operations. Arbitrary sequences of these operations are perfectly reversible without the use of a cleanup operation as long as the operands that need to be inverted are known in advance.

2.1.2 Iterative Binding. One particularly useful application of unitary vectors involves iterating a specific binding operation to generate a set of points that are closed under a linear transformation (Komer, 2020). Typically, this involves binding a given vector to itself some number of times such that if k is a natural number and A is a vector, then

$$A^k = \underbrace{A \circledast A \circledast A \ldots \circledast A}_{A \text{ appears } k \text{ times}}. \tag{2.4}$$

This process is sometimes referred to as a repeated "autoconvolution" (Plate, 2003).

The significance of this definition lies in the fact that when A is unitary, iterative binding creates a closed sequence of approximately orthogonal vectors that can be easily traversed. For example, moving from the vector A^k in the sequence to the vector A^{k+1} is as simple as performing the binding $A^k \circledast A$; moving back to A^k from A^{k+1} is as simple as performing the binding $A^{k+1} \circledast A^{-1}$ due to the fact that the inverse is exact for unitary vectors. More generally, because $A^{k_1+k_2} = A^{k_1} \circledast A^{k_2}$ for all integers k_1 and k_2, a single binding operation suffices to move between any two vectors in the closed sequence corresponding to self-binding under A.

Because further binding operations can be used to associate the points in this sequence with other representations, it becomes very natural to encode a list of elements in a single vector using these techniques. For example, to encode the list $X, Y, Z \ldots$, one could bind each vector in this list to neighboring points as follows: $A^1 \circledast X + A^2 \circledast Y + A^3 \circledast Z \ldots$ The retrieval of specific elements from this list can then be performed by moving to the desired cue in the set of vectors defined by A closed under self-binding (e.g., A^2), and then unbinding this cue to extract the corresponding element from the encoded list (e.g., Y). This method has been used in several neural models of working memory (Choo, 2010; Eliasmith et al., 2012; Gosmann & Eliasmith, 2020).

2.1.3 Fractional Binding. It is possible to further generalize iterative binding by allowing k to be a real-valued number rather than an integer. Mathematically, a fractional number of binding iterations can be expressed in the Fourier domain as

$$A^k \overset{\text{DEF}}{=} \mathcal{F}^{-1}\left\{\mathcal{F}\{A\}^k\right\}, \tag{2.5}$$

where $\mathcal{F}\{A\}^k$ is the element-wise exponentiation of a set of complex Fourier coefficients. This definition is equivalent to equation 2.4 when k is a positive integer but generalizes it to allow k to be real.

Assuming A is unitary, we often find it convenient to restate this definition by using Euler's formula and the fact that exponentiating a unit-length complex number is equivalent to scaling its polar angle,

$$A^k = \mathcal{F}^{-1}\left\{e^{ik\phi}\right\}, \tag{2.6}$$

where ϕ are the polar angles of $\mathcal{F}\{A\}$. Likewise, binding two unitary vectors is equivalent to adding their polar angles, as is made implicit in equation 2.2.

One consequence of this definition is that for any $k_1, k_2 \in \mathbb{R}$, the following algebraic property holds:

$$A^{k_1} \circledast A^{k_2} = A^{k_1+k_2}. \tag{2.7}$$

Thus, fractional binding can be used to generate a closed sequence of vectors that can easily be traversed or continuously shifted by any Δk. However, the vectors are *not* all approximately orthogonal to one another. Rather, for nearby values of k, the pointers A^k will be highly similar to one another (Komer, 2020). In particular, as the dimensionality becomes sufficiently high, the expected similiarity approaches:

$$A^{k_1} \cdot A^{k_2} = \mathrm{sinc}(k_2 - k_1) \tag{2.8}$$

for unitary A with independent $\phi \sim \mathcal{U}(-\pi, \pi)$ (Voelker, 2020). We address these latter points in section A.1.

The most significant consequence of being able to perform fractional binding using real values for k is that vectors of the form A^k can encode continuous quantities. Such continuous quantities can then be bound into other representations, thereby allowing for vectors that encode arbitrary blends of continuous and discrete elements. For example, a discrete pair of continuous values could be represented as $P_1 \circledast A^{k_1} + P_2 \circledast A^{k_2}$. Similarly, a continuous three-dimensional value could be represented as $A^{k_1} \circledast B^{k_2} \circledast C^{k_3}$. The point to draw from such examples is that the use of fractional binding significantly expands the class of data structures that can be encoded and manipulated using a vector symbolic architecture, namely, to those definable over continuous spaces (i.e., with continuous slots).

2.2 Spatial Semantic Pointers. To best make use of the above features of VSAs in the context of spatial reasoning tasks, we incorporate fractional binding operations into a cognitive modeling framework called the semantic pointer architecture (SPA; Eliasmith, 2013). The SPA provides an

architecture and methodology for integrating cognitive, perceptual, and motor systems in spiking neural networks. The SPA defines vector representations, named semantic pointers (SPs), that (1) are produced via compression and collection operations involving representations from arbitrary modalities, (2) express semantic features, (3) "point to" additional representations that are accessed via decompression operations, and (4) can be neurally implemented (Eliasmith, 2013). By integrating symbolic structures of the sort defined by conventional VSAs with richer forms of sensorimotor data, the SPA has enabled the creation of what still remains the world's largest functioning model of the human brain (Eliasmith et al., 2012; Choo, 2018), along with a variety of other models focused on more specific cognitive functions (Rasmussen & Eliasmith, 2011; Stewart, Choo, & Eliasmith, 2014; Crawford, Gingerich, & Eliasmith, 2015; Blouw, Solodkin, Thagard, & Eliasmith, 2016; Gosmann & Eliasmith, 2020).

SSPs extend the class of representational structures defined within the SPA by binding arbitrarily complex representations of discrete objects to points in continuous topological spaces (Komer et al., 2019; Lu, Voelker, Komer, & Eliasmith, 2019; Komer & Eliasmith, 2020). The SPA provides methods for realizing and manipulating these representations in spiking (and nonspiking) neural networks.

For convenience, we introduce the shorthand notation for encoding a position (x, y) in a continuous space,

$$P(x, y) \stackrel{\text{DEF}}{=} X^x \circledast Y^y, \tag{2.9}$$

where $X, Y \in \mathcal{V}$ are randomly generated, but fixed, unitary vectors representing two axes in Euclidean space. Our definitions naturally generalize to multidimensional topological spaces, but we restrict our focus here to two-dimensional Euclidean space for purposes of practical illustration.

Mathematically, an SSP that encodes a set of m objects on a plane can be defined as

$$M = \sum_{i=1}^{m} \text{OBJ}_i \circledast S_i. \tag{2.10}$$

In equation 2.10, $\text{OBJ}_i \in \mathcal{V}$ is an arbitrary SP representing the ith object, and

$$S_i = P(x_i, y_i), \tag{2.11}$$

where (x_i, y_i) is its position in space, or

$$S_i = \int_{(x,y)\in R_i} P(x, y)\, dx\, dy, \tag{2.12}$$

where $R_i \subseteq \mathbb{R}^2$ is its region in two-dimensional space.

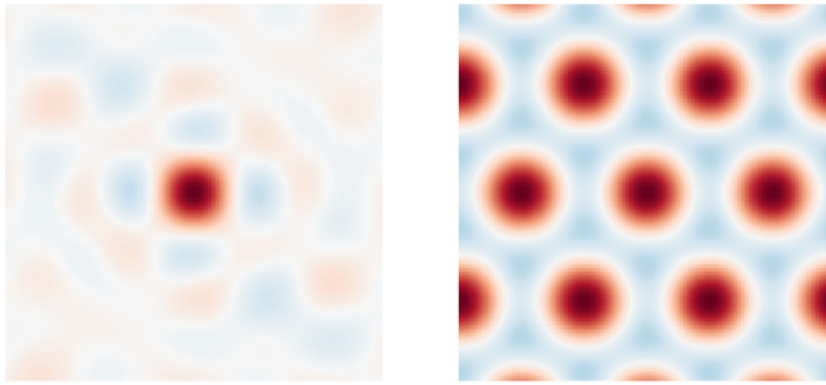


Figure 1: Example similarity maps for two different SSPs, each encoding a single object at position $(0, 0)$ on a square map of size $6\sqrt{2}$. Left: A 512-dimensional SSP is constructed using randomly generated unitary axis vectors. Right: A 7-dimensional SSP is constructed using axis vectors that produce a hexagonal lattice (see section 2.2.2).

The integral defining S_i can range over a set of points, in which case arbitrary regions can be encoded into the SSP (also see section 2.2.1); otherwise, S_i encodes a single point in the plane. The sum ranging from 1 to m can further include null entity and spatial representations (i.e., the identity semantic pointer) such that the SSP is able to include entity representations not associated with particular locations in space, along with spatial representations not associated with particular entities. These features of the definition allow SSPs to flexibly encode information about a wide range of spatial (and other continuous) environments.

SSPs can be manipulated to, for example, shift or locate multiple objects in space (Komer et al., 2019) and to query the spatial relationships between objects (Lu, Voelker, Komer, & Eliasmith, 2019). To provide a simple example, an object located at (x, y) can be retrieved from an SSP by computing $M \circledast P(x, y)^{-1}$ and then cleaning up the result to the most similar SP in the vocabulary $\{\text{OBJ}_i\}_{i=1}^{m}$. Likewise, the ith point or region in space, S_i, can be retrieved by computing $M \circledast \text{OBJ}_i^{-1}$, with an optional cleanup on the result. As a final example, equation 2.7 can be exploited to shift all coordinates in M by the same amount, $(\Delta x, \Delta y)$, with the binding $M \circledast P(\Delta x, \Delta y)$.

2.2.1 Visualizing SSPs with Similarity Maps. To visualize some object's point or region in space (S_i, from equations 2.11 or 2.12), we often plot what we refer to as the "similarity map" (see Figure 1). This is computed by selecting some set of two-dimensional coordinates that one wishes to evaluate and then for each (x, y) coloring the corresponding pixel by the value

of $P(x, y)^T S_i$, that is, the similarity between the pointer and the correspondingly encoded point in space. Typically the colors are chosen from a diverging color palette, normalized to $[-1, 1]$, the limits for the similarity between two unitary vectors.

More formally, the similarity map is a linear operator that consumes an SSP, S, and produces a function of (x, y) for that particular SSP, which we can denote mathematically as

$$\mathcal{M}(S) = P(x, y)^T S. \tag{2.13}$$

Figure 1 illustrates example similarity maps for two different choices of axis vectors (X, Y), giving different $P(x, y)$. In each case, we are plotting $\mathcal{M}(P(0, 0))$, where (x, y) are evenly tiled across a square grid of size $6\sqrt{2}$, centered at $(0, 0)$.

These illustrations are useful for understanding and visualizing what a particular SSP is representing. It is important to note that SSPs themselves are not discretized sets of pixels; they are essentially compressed representations thereof using Fourier basis functions. The similarity map is primarily a tool for visualization.

That said, the similarity map does provide an important insight: equations 2.11 or 2.12 are sufficient but not necessary ways of constructing SSPs that encode points or regions of space. That is, S can be any vector such that $\mathcal{M}(S)$ approximates some desired function of (x, y), so that the similarity between S and $P(x, y)$ represents that spatial map. We provide an example of an interesting application of this insight in section A.1.

The similarity map also has a number of important properties. First, shifting the similarity map of A is equivalent to shifting A by the same amount, since

$$\begin{aligned} P(x + x', y + y')^T S &= \left(P(x, y) \circledast P(x', y')\right)^T S \\ &= P(x, y)^T \left(S \circledast P(x', y')\right) \\ &= \mathcal{M}\left(S \circledast P(x', y')\right). \end{aligned} \tag{2.14}$$

Second, since the dot product is a linear operator, the function that produces a similarity map, $\mathcal{M}(\cdot)$, is in fact a linear function; taking a linear combination of similarity maps is equivalent to taking the similarity map of that same linear combination of vectors. Third, the peak(s) in the plot, over some spatial domain, correspond to the ideal target(s) for a cleanup that is to output the most similar $P(x, y)$ given that domain as its vocabulary.

2.2.2 Relevance of SSPs to Neurobiological Representations. SSPs can also be used to reproduce the same grid cell firing patterns (Dumont & Eliasmith, 2020) that have been famously identified as a basis for the brain's spatial

representation system (Moser, Kropff, & Moser, 2008). This is accomplished by generating the axis vectors, (X, Y), in the following manner.

Consider the similarity map of a d-dimensional SSP representing a single point, noting that the dot product between two vectors is equal (up to a constant) to the dot product of their Fourier transforms,

$$\mathcal{M}(P(x_0, y_0)) = P(x, y)^T P(x_0, y_0) \propto \sum_{j=1}^{d} e^{i(K_{j,1}(x+x_0)+K_{j,2}(y+y_0))}, \tag{2.15}$$

where $K_{j,1}$ is the polar angle of the jth component of the Fourier transform of the axis vector X, and $K_{j,2}$ is the polar angle for the axis vector Y.

Sorscher, Mel, Ganguli, and Ocko (2019, see equation 2.10) and Dumont and Eliasmith (2020, see equations 2.11–2.13) provide the conditions on these phases such that the similarity map is a regular hexagonal lattice pattern across the (x, y) plane. The core building block is a matrix $K \in \mathbb{R}^{3\times 2}$ that holds the three two-dimensional coordinates of an equilateral triangle inscribed in a unit circle:

$$K = \begin{pmatrix} 0 & 1 \\ \sqrt{3}/2 & -1/2 \\ -\sqrt{3}/2 & -1/2 \end{pmatrix}. \tag{2.16}$$

Using these values for the polar angles in equation 2.15 produces hexagonally tiled similarity maps. However, to ensure the resulting axis vectors are real and unitary, additional polar angles are needed. In particular, the Fourier transform of the axis vectors must have Hermitian symmetry (i.e., contain the complex conjugates of the coefficients given by K) and a zero-frequency term of one. This results in seven-dimensional axis vectors.

Scaled and/or rotated versions of the K matrix can be used to set the resolution and orientation, respectively, of the resulting hexagonal pattern. Here, scaling means multiplying K by a fixed scalar, and rotating means applying a 2D rotation matrix to the columns of K. For instance, to produce place or grid cell bumps with a diameter of $\sqrt{2}$ (see Figure 1, right), the scaling factors should have a mean of $2\pi/\sqrt{6}$.[4]

While these 7-dimensional SSPs can be used to create spiking neural networks with grid cell–like firing patterns, this patterned similarity means that such SSPs can only uniquely represent a small area of (x, y) values. To increase the representational power of the SSPs and produce place cell–like similarity maps, the dimensionality must be increased by combining

[4] Using the fact that the distance between peaks in the hexagonal lattice is $4\pi/\left(\sqrt{3}|\mathbf{k}|\right)$, where $|\mathbf{k}|$ is the scaling factor on K (Dumont & Eliasmith, 2020).

a variety of grids with different resolutions and orientations. Scaled and rotated versions of K can be stacked in the Fourier domain n times to generate the x- and y-axis vectors. The final dimensionality of the axis vectors is $d = 6n + 1$ (6 from each 3-dimensional K block of different scale/rotation and their complex conjugates, and 1 from the zero frequency term). SSPs that use such axis vectors will be referred to as hexagonal SSPs. Neurons in networks representing hexagonal SSP can pick out the different grids it contains. This reproduces grid cell–like tuning distributions and provides more accurate place cell representations using SSPs in spiking neural networks (Dumont & Eliasmith, 2020).

2.2.3 SSPs as Features for Deep Learning. SSPs, used within the greater SPA framework, provide a general method for encoding continuous variables in high-dimensional vector spaces in which algebraic operations have semantic interpretations. Hence, these vectors and operations can be used to craft complex cognitive models. Beyond this, SSPs are a useful tool in deep learning as a method for embedding data in a feature space with desirable properties.

The data used to create features and the way in which features are represented can potentially have a large impact on performance of a neural network (Komer, 2020). Here, "features" refers to the initial vector input provided to a network. Each layer of a network receives information represented as a vector and projects it into another vector space. The neurons in a layer can be thought of as the basis of its space. If half (or more) of the neurons are active, that means most of the dimensions of the vector space are needed to represent the information. Such a layer would contain a dense coding of the incoming information. If only one neuron was activated in a layer, it would represent a local coding. One-hot encoding and feature hashing are examples of heuristics for encoding discrete data that correspond to a local and dense coding, respectively. Anything in between a dense and local code is called a sparse code. Sparse codes are an advantageous balance between the two extremes; they have a higher representational capacity and better generalizability than local codes and are generally easier to learn functions from compared to dense codes (Foldiak, 2003). In other words, such feature vectors have a reasonable dimensionality and result in better performance when used in downstream tasks.

Feature learning (e.g., with an autoencoder) can be used to learn useful representations, but, as is often the case in deep learning, the result is usually uninterpretable. A well-known model for word embedding is `word2vec`, which takes a one-hot encoding of a word and outputs a lower-dimensional vector. An important property of the resultant vectors is that their cosine similarity corresponds to the encoded words' semantic similarity. Distinct inputs have a similarity close to zero. When word embeddings with this property are used, it is easy to learn a sparse coding with a single layer of neurons.

How can features with such properties be constructed from continuous variables, in particular, low-dimensional variables? Continuous numeric values can be fed directly into a neural network, but this raw input is in the form of a dense coding, and so generally larger/deeper networks are needed for accurate function approximation. To obtain a sparse encoding from continuous data, it must be mapped onto a higher-dimensional vector space. This is exactly what is done when encoding variables as SSPs. A simple autoencoder cannot solve this problem; autoencoders work by using an informational bottleneck, and if their embedding had a higher dimension than the input, it would just learn to pass the input through. Methods such as tile coding and radial basis function representation can be used to encode continuous variables in higher-dimensional spaces, but SSPs has been found to have better performance over such coding on an array of deep learning tasks (Komer, 2020). Much like `word2vec` embeddings, SSPs of variables with a high distance in Euclidean space will have a low cosine similarity, and so a sparse code can be obtained from them. Furthermore, SSPs can be bound with representations of discrete data without increasing their dimensionality to create structured symbolic representations that are easy to interpret and manipulate. These properties of SSPs motivate extending the theory of SSPs to represent trajectories and dynamic variables. This will fill the gap in methods for structured, dynamic feature representation.

3 Methods for Simulating Dynamics with SSPs

Prior work on SSPs has focused on representing two-dimensional spatial maps (Komer et al., 2019; Lu et al., 2019; Komer & Eliasmith, 2020; Dumont & Eliasmith, 2020; Komer, 2020). Some of that work has also shown that on machine learning problems requiring continuous representations, SSPs most often should be the preferred choice. Specifically, Komer (2020) compared SSPs to four other standard encoding methods across 122 different tasks and demonstrated that SSPs outperformed all other methods on 65.7% of regression and 57.2% of classification tasks. Here, we extend past work by introducing methods for representing arbitrary single-object trajectories and methods for simulating the dynamics of one or more objects following independent trajectories.

3.1 Representing Arbitrary Trajectories. For many problems, one needs to encode data that are dynamic, not static. Time is a continuous variable, much like space, and so it too can be included in an SSP as an additional axis,

$$P(x, y, t) = C^t \circledast X^x \circledast Y^y, \tag{3.1}$$

where C is a randomly chosen unitary SP. This time pointer can be manipulated and decoded the same way as any other SSP.

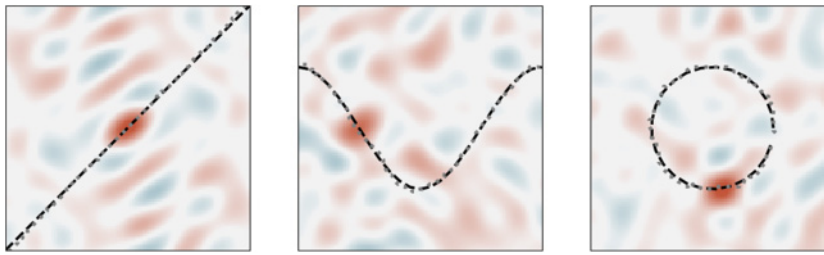


Figure 2: Decoding trajectories represented by SSPs (left: linear; middle: cosine; right: circle). Twenty-five sample points from a continuous trajectory (shown as a black dotted line) are bound with a "cue" trajectory (e.g., time) and summed together as an SSP. To decode the interpolated position from the SSP, 144 cue points are used. These decoded positions (after the cleanup procedure) are plotted with a dotted gray line. Snapshots of the similarity plots computed using decoded positions are plotted underneath.

A sequence of such time-dependent SSPs is an encoding of an entire arbitrary trajectory. The equation to encode such trajectories is

$$\sum_{i=1}^{m} C^{t_i} \circledast S_i. \tag{3.2}$$

It is a sum over a discrete collection of m points, where S_i is the encoding of each point spatially and t_i is the relative time of that point within the trajectory. If we wanted to encode an ordered sequence of SSPs, order could be used as "time," and t_i would be the position of the spatial SSPs within the list.

Continuous trajectories may also be encoded with a natural generalization of equation 3.2:

$$\int_0^t C^{\tau} \circledast S(\tau)\, d\tau, \tag{3.3}$$

where $S(\tau)$ produces the encoding for each point within some trajectory of points defined over a continuous interval of time, $\tau \in [0, t]$.

An SSP that implements a transformation that moves smoothly along this trajectory can then be used to decode information that interpolates between the points in the original nonlinear trajectory. When few and distant sample points are used, this interpolation resembles a linear interpolation, although the result of decoding outside the time range of the samples points (i.e., extrapolating) decays to zero. Figure 2 shows three continuous trajectories being decoded from SSPs. The first traces out a linear $y = x$ function, the second traces out a cosine function, and third traces out a circle.

The smoothness of the replay depends on the number of sample points used to learn the trajectory, the distances between sample points, the dimensionality of the SSP, and the scaling of the time and space variables. The amount of noise in the visualization also depends on these factors since, for example, a lower-dimensional SSP will not be able to encode a large number of trajectory sample points without incurring a significant amount of compression loss.

The method presented here for encoding an entire sequence of dynamic variables as a fixed-length vector could be used in many domains. Co-Reyes et al. (2018) provide an example of where performance in continuous state and action reinforcement learning problems can be improved using trajectory encodings. They use trained autoencoders to encode sequences of state-space observations and generate action trajectories. Encoding a trajectory with an SSP does not require any learning. Despite not being learned, SSPs have been found to perform better on navigation tasks compared to learned encodings (Komer & Eliasmith, 2020).

3.2 Simulating Arbitrary Trajectories. In this section, we move from encoding specific trajectories to simulating these trajectories online using some specification of the dynamics governing each object represented by an SSP.

3.2.1 Simulating Single Objects with Continuous PDEs. First, we illustrate how to simulate partial differential equations in the SSP space by applying transformations to an initial SSP that encodes a single object at a particular point in space. In the discrete-time case, a transformation will be applied to the SSP at every timestep to account for the movement of the object it represents. Mathematically, these updates to the initial SSP take the following form over time.

$$M_{t+\Delta t} = \left(X^{\Delta x_t} \circledast Y^{\Delta y_t}\right) \circledast M_t = P(\Delta x_t, \Delta y_t) \circledast M_t, \tag{3.4}$$

where Δx_t and Δy_t are derived from differential equations that relate x and y to t in some way. For example, if the underlying dynamics are linear, we have $\Delta x = \frac{dx}{dt}\Delta t$. Assuming $M_t = X^{x_t} \circledast Y^{y_t}$, then the algebraic properties of SSPs ensure that $X^{x_t} \circledast Y^{y_t} \circledast X^{\Delta x_t} \circledast Y^{\Delta y_t} = X^{x_t + \Delta x_t} \circledast Y^{y_t + \Delta y_t} = M_{t+\Delta t}$, as required.

An important observation is that binding with $P(\Delta x_t, \Delta y_t)$ is equivalent to applying a matrix transformation to the SSP (see equation 2.3). Specifically,

$$M_{t+\Delta t} = T\left(P(\Delta x_t, \Delta y_t)\right) M_t, \tag{3.5}$$

where $T(\cdot)$ is the linear operator that produces the binding matrix for a given SP (i.e., $T(A)B = A \circledast B$). Although we have assumed above that the differential equations are expressed over discrete time, we can also do the same over continuous time. We omit the proof and simply state the result, using ln to denote the principal branch of the matrix logarithm:

$$\begin{aligned}\frac{dM}{dt} &= \left(\ln T\left(P\left(\frac{dx}{dt}, \frac{dy}{dt}\right)\right)\right) M \\ &= \left(\frac{dx}{dt}\ln T(X) + \frac{dy}{dt}\ln T(Y)\right) M. \end{aligned} \tag{3.6}$$

Therefore, the continuous-time partial differential equation that evolves an SSP over time with respect to dx/dt and dy/dt is the addition of two fixed matrix transformations of the SSP that are each scaled by their respective derivatives. Equation 3.6 is thus equivalent to a linear dynamical system, with two time-varying integration time constants controlling the velocity of x and y—a system that can be implemented accurately and efficiently by recurrent spiking neural networks (Eliasmith & Anderson, 2003; Voelker, 2019).

We can alternatively express the above continuous-time linear system as a binding, by first introducing the following definition, which is analogous to equation 2.5:

$$\ln A \stackrel{\text{DEF}}{=} \mathcal{F}^{-1}\{\ln \mathcal{F}\{A\}\}. \tag{3.7}$$

$$\frac{dM}{dt} = \left(\frac{dx}{dt}\ln X + \frac{dy}{dt}\ln Y\right) \circledast M. \tag{3.8}$$

The system of equation 3.6 maps directly onto a recurrent neural network with two recurrent weight matrices, $\ln T(X)$ and $\ln T(Y)$, with gain factors dx/dt and dy/dt that independently scale the respective matrix-vector multiplications. In our examples, we simulate the dynamics of equation 3.8 using a spiking neural network. When simulating with spiking neurons, the presynaptic activity must be filtered to produce the postsynaptic current. This filtering must be accounted for in recurrent networks.

The neural engineering framework (NEF; Eliasmith & Anderson, 2003) provides a methodology for representing vectors via the collective activity of a population of spiking neurons and for implementing dynamics via recurrent connections. Assuming a first-order low-pass filter is used, a neural population representing a vector M can evolve according to some dynamics $\frac{dM}{dt}$ via a recurrent connection with weights set to perform the transformation $\tau\frac{dM}{dt} + M$, where τ is the filter time constant. Optimal weights to perform a transformation with one layer can be found via least-squares optimization.

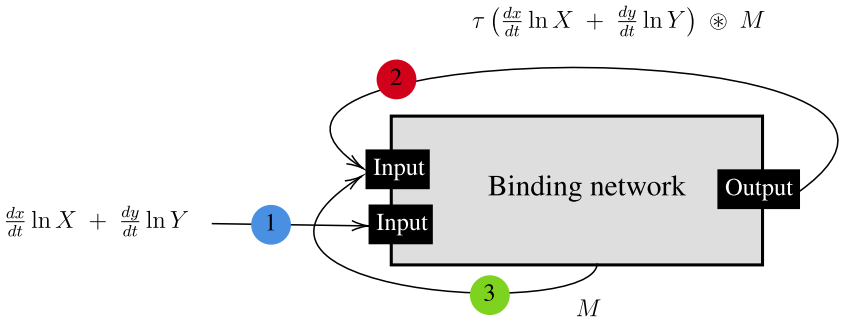


Figure 3: A neural network that implements the dynamical system of equation 3.8 using the binding network described in Stewart, Bekolay, and Eliasmith (2011). The binding network consists of $4(\lfloor \frac{d}{2} \rfloor + 1)$ neural populations (where d is the size of the SSP). The weights connecting the inputs to these populations are set to compute the Fourier transform. Each population performs one of the element-wise multiplications of the two inputs in the Fourier domain. Weights outgoing from these populations to the output of the network perform the inverse Fourier transform.

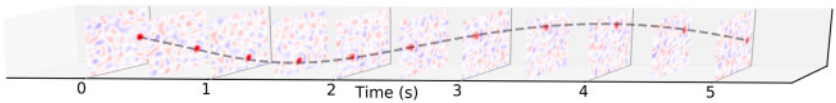


Figure 4: Using a spiking neural network to simulate a continuous-time partial differential equation that maps onto the oscillation of a single object. The neural network consists of a single recurrently connected layer of 30,400 spiking neurons.

Prior work has used the NEF to construct a neural network that preforms circular convolution (Stewart, Bekolay et al., 2011). The system of equation 3.8 maps onto such a network, as shown in Figure 3. The network binds an input vector $\frac{dx}{dt}\ln X + \frac{dy}{dt}\ln Y$ (passed in through connection 1 in the diagram) with an internal state vector that represents the SSP M. Using two recurrent connections, one that feeds the result of the binding scaled by τ back into the network as input (connection 2) and another that feeds the internally represented M (prebinding) to the input as well (connection 3), the dynamics of equation 3.8 are realized. Thus, the methods presented here can be used to build neural networks that move any SSP around space by integrating dx/dt and dy/dt over time, akin to neural models of path integration (Conklin & Eliasmith, 2005).

To provide an example, we use these methods to simulate an SSP oscillating in two-dimensional space (see Figure 4), similar to a controlled cyclic attractor (Eliasmith, 2005). Hexagonal SSPs (described in section 2.2.2) are

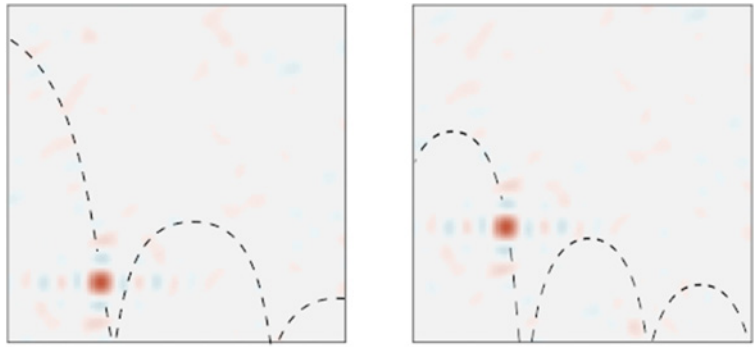


Figure 5: Simulating trajectories governed by differential equations and elastic collisions. Each trajectory is generated by binding an SSP encoding the object's state to a "transformation" SSP at every time step that encodes its instantaneous velocity in accordance with the effects of initial velocity, gravity, and collisions with the bottom of the plane.

used with $n = 25$ (from having five scalings of K uniformly ranging from 0.9 to 1.5, and five rotations; this results in $d = 151$) and 100 spiking integrate-and-fire neurons per element-wise product in the Fourier domain (30,400 neurons total). In this example, the partial derivatives dx/dt and dy/dt are determined by a 0.2 Hz oscillation with a radius of five spatial units.

It is also possible to impose collision dynamics whenever the simulated trajectory of a represented object encounters a solid surface (e.g., a floor or a wall). Specifically, the instantaneous velocity imposed by a given transformation is inverted and scaled on impact with a surface in accordance with a simple model of the physics of elastic collisions. In other words, the $(\Delta x_t, \Delta y_t)$ or $(dx/dt, dy/dt)$ corresponding to times at which collisions occur can be determined in a context-dependent manner. In Figure 5, snapshots from simulations of balls being dropped (left) and tossed (right) with different initial conditions before bouncing on a hard surface are shown. Here, sequences of transformation SSPs are derived assuming (1) an initial object position and velocity, (2) a differential equation for the effect of gravity, and (3) a simple model of elastic collisions that reverses and scales the object's movement along the y-axis when it reaches of the bottom of the represented plane.

To summarize, we can model both discrete time and continuous time differential equations over space by using linear transformations (see equations 3.5 and 3.6) or bindings (equations 3.4 and 3.8) that are applied to the SSP over time. Note that the dynamics themselves are driven by the velocity inputs. The purpose of the methods presented here is to simulate such dynamics in the space of SSPs and demonstrate that this is possible using spiking neural networks. These methods can be used in various ways.

In the case where the dynamics are self-motion of an animal, this simulation is a biologically plausible path integration model, the result of which could be used in downstream cognitive tasks. When hexagonal SSPs are used, the model will have grid cell–like firing patterns, consistent with neurobiological findings. Furthermore, simulating dynamics in the SSP space circumvents the issue of limits on the representational radius of neural populations, which prevents dynamics in Euclidean space from being simulated directly with such networks. In addition, encodings of dynamically evolving quantities can be used as features in a deep learning network. A dynamic encoding could be quite useful for certain problems, such as an online reinforcement learning task, where keeping track of moving obstacles may be important, or problems that involve predicting future trajectories.

3.2.2 Predicting Future Object Positions. While simulating a dynamical system can be used to perform prediction, we can also directly tackle the problem without using knowledge of underlying dynamics by training a network to output a prediction of the future state of a moving object. Here, we briefly describe a technique that exploits SSP representations to enable prediction of a bouncing ball in a square environment. Specifically, for this network, we provided the SSP representation of a ball bouncing around a square environment as an online input time series to a recently developed type of recurrent neural network called a Legendre memory unit (LMU; Voelker, Kajić, & Eliasmith, 2019).

The LMU has been shown to outperform other kinds of recurrent networks in a variety of benchmark tasks for time series processing. It has also been shown to be optimal at compressing continuous signals over arbitrary time windows, which lends itself well to dynamic prediction. The LMU works by using a linear time-invariant system (defined by matrices $\mathbf{A} \in \mathbb{R}^{n \times n}$ and $\mathbf{B} \in \mathbb{R}^{n \times 1}$) to orthogonalize input signals across a sliding time window of some fixed length θ. The windowed signal is projected to a basis that resembles the first n Legendre polynomials (that are shifted, normalized, and discretized).

Assume we have an SSP varying over time, $M_t \in \mathbb{R}^d$, and let $[M_t]_j$ be its jth component. At each time step, the LMU will update its internal memory, $\mathbf{m}_t^{(j)}$, for each component of the SSP as follows:

$$\mathbf{m}_t^{(j)} = \mathbf{A}\mathbf{m}_{t-1}^{(j)} + \mathbf{B}\,[M_t]_j\,. \tag{3.9}$$

The matrices that defined the LTI system are given by

$$[A]_{ij} = \frac{(2i+1)}{\theta} \begin{cases} -1 & \text{if } i \leq j \\ (-1)^{i-j+1} & \text{else} \end{cases} \tag{3.10}$$

$$[B]_i = \frac{(2i+1)(-1)^i}{\theta}. \tag{3.11}$$

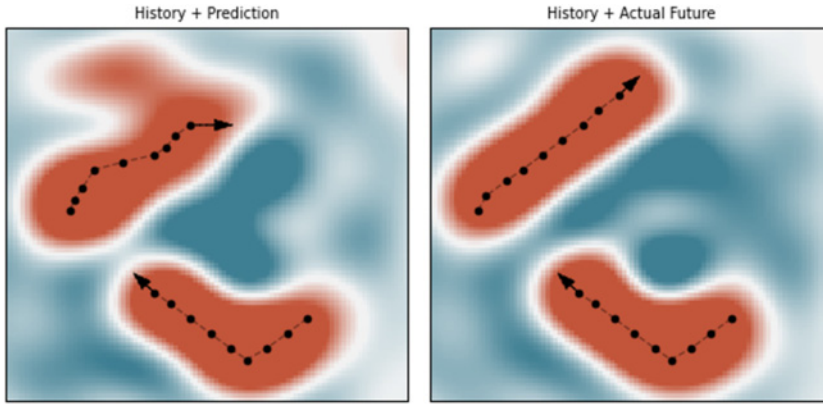


Figure 6: Similarity maps of the sum of the history of a bouncing ball (a time window of SSPs) and (right) its future or (left) a prediction of this future computed using the Legendre memory unit (LMU) and the history. Plotted are dashed lines indicating the positions represented by these time windows of SSPs, which are computed using a cleanup procedure.

The LMU that we employ here has a history window θ of 4 s and uses 12 Legendre bases (e.g., $n = 12$). A hexagonal 541-dimensional SSP is used with 10 scalings of K uniformly ranging from 0.9 to 3.5, and 9 rotations.

The model predicts the same window of time but 6 s into the future (hence, there is a 2 s gap between the current time and the start of the predicted future states). The output of the LMU is fed into a neural network with three dense layers. Input to the network are the d memory vectors computed via the LMU, $\mathbf{m}_t^{(j)} \in \mathbb{R}^{n\times 1}$, one for each dimension of the SSP, as a single flattened vector with size 6492. The first two layers have a hidden size of 1024 and are followed by an ReLU activation function. The last layer has a hidden size of 6492. The output of this network is projected onto n Legendre polynomials to obtain a prediction of the SSP at 10 time points equally spaced over the future window. The network has a total of 14,352,732 trainable parameters.

The network is trained on 4000 s of bouncing dynamics within a 1 by 1 box. The training data are a time series of SSPs encoding the ball's position at 0.4 s intervals. The ball's trajectory is generated by a simulation with random initial conditions (position within the box and velocity) and the dynamics of boundary collisions.

An example simulation is shown in Figure 6, in which the LMU prediction and the actual future SSP representations are compared. We found that it was able to accurately represent sliding windows of the ball's history while simultaneously predicting the sliding window of its future. We consider this a proof-of-principle demonstrating the potential to successfully

marry SSPs with deep learning methods for dynamic simulation and prediction. As a demonstration, we leave quantitative analysis and comparison to integration with other deep networks for future work (see Komer, 2020, for evidence of the general utility of SSPs for regression and classification problems).

3.3 Simulating Multiple Objects. While equation 2.10 establishes that a single SSP can be used to encode and decode the positions of multiple objects, further steps are required to simulate continuous dynamics defined with respect to these objects. Here, we illustrate that it is possible to define unique operators or functions that concurrently simulate the dynamics of multiple objects, such that each object follows its own independent trajectory. Specifically, we describe a method for simulating the dynamics of multiple objects algebraically, along with a method for simulating such dynamics using a trained function approximator.

In the context of SSPs that encode multiple objects, it is essential to use cleanup operations given that whenever a specific object's position is extracted from an SSP, the extracted position will be a lossy approximation of the true position, since $M \circledast \text{OBJ}_i^{-1} \approx X^{x_i} \circledast Y^{y_i}$. Moreover, the amount of compression loss observed in such cases will be proportional to the number of encoded objects, making the application of transformations to SSPs encoding multiple objects difficult. However, by leveraging knowledge about the specific methods used to generate these SSPs, it is possible to define cleanup methods that significantly ameliorate these challenges.

3.3.1 Cleanup Methods. In this section we introduce two cleanup methods for SSPs. The first method involves precomputing $X^{x_i} \circledast Y^{y_i}$ at a discrete number of points along a fixed grid, and then taking the dot product of each vector with $M \circledast \text{OBJ}^{-1}$ to return the point (x_i, y_i) with maximal cosine similarity to the given SSP. This can be implemented by a single layer in a neural network, whose weights are $X^{x_i} \circledast Y^{y_i}$, followed by an argmax, threshold, or some other winner-take-all mechanism. The main disadvantage of this approach is that it does not naturally handle continuous outputs, although one can weigh together neighboring dot product similarities to interpolate between the grid points.

The second cleanup method takes advantage of the fact that it is usually only important to retrieve $X^{x_i} \circledast Y^{y_i}$ and not necessarily (x_i, y_i). Specifically, we need to learn a function f that takes the noisy result of unbinding an object from an SSP as input and outputs a "clean" SSP:

$$f(M \circledast \text{OBJ}_i^{-1}) = X^{x_i} \circledast Y^{y_i}, \tag{3.12}$$

$$\text{where} \quad M \circledast \text{OBJ}_i^{-1} = X^{x_i} \circledast Y^{y_i} + \eta \circledast \text{OBJ}_i^{-1}, \tag{3.13}$$

and η is some "noisy" vector standing for the superposition of all other objects in the SSP to which the unbinding operation applies. This function f can be trained in the same manner as an autoencoding denoiser, with mean squared error (MSE) loss and gaussian noise ($\sigma = 0.1/d$) included on the training inputs (Komer & Eliasmith, 2020). We find that a network comprising one layer of ReLUs, with as many neurons as there are dimensions in the SSP, is sufficient. We also normalize the output of the cleanup to force it to remain unitary. The benefit of this approach is that it has a continuous output (as opposed to a discrete set of possible outputs). The cleanup can also be trained to return a continuous (x_i, y_i) as opposed to $X^{x_i} \circledast Y^{y_i}$.

3.3.2 Simulating Multiple Objects Algebraically. The algebraic properties of SSPs introduce a number of possibilities for applying transformations to the spatial positions of multiple objects. Recall from section 2.2 that for an SSP encoding a set of object positions, it is possible to shift all of these positions by Δx and Δy as follows:

$$M = M \circledast X^{\Delta x} \circledast Y^{\Delta y}. \tag{3.14}$$

The drawback of this method is that it makes it impossible to apply different updates to each object position, since there is no way to distinguish which update should be applied to which position (due to the fact that addition is commutative).

An alternative method involves tagging each pair of coordinates with a representation of the corresponding object and then applying transformations to update the position of each object independently. The use of tagging allows each object position to be extracted and updated separately, after which these updated positions can rebound to their corresponding objects and summed together to produce a new SSP. Mathematically, this process is described as follows,

$$M \leftarrow \sum_{i=1}^{m} \mathrm{OBJ}_i \circledast f(M \circledast \mathrm{OBJ}_i^{-1}) \circledast X^{\Delta x_i} \circledast Y^{\Delta y_i}, \tag{3.15}$$

where f is the cleanup from equation 3.12. We find that this approach works well as long as the cleanup function is sufficiently accurate. The main drawback is that m applications of the cleanup are required at every time step, although these can be applied in parallel and added together.

An even more effective method involves tagging each set of coordinates with a representation of the corresponding object and then applying additive shifts to update the position of each object independently as follows:

$$M \leftarrow M \circledast \mathrm{OBJ}_i \circledast \Delta P_i, \tag{3.16}$$

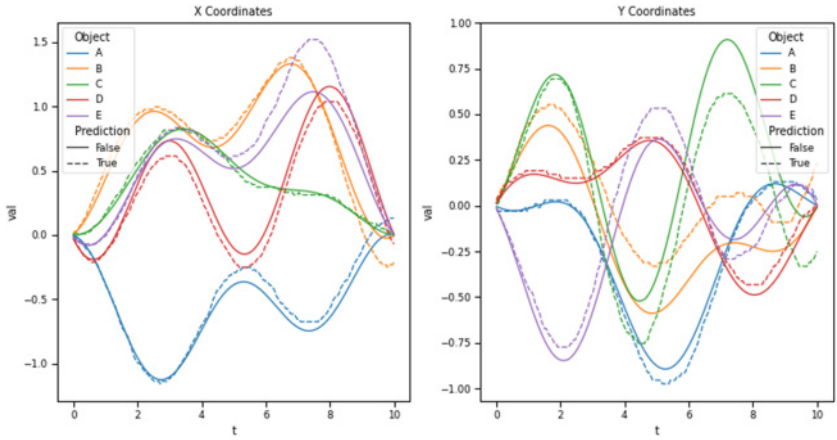


Figure 7: Using separate algebraic updates (see equation 3.16) with the first cleanup method to simulate the trajectories of five different objects within a single SSP. One starts with an SSP representation of all five object initial positions. At each 0.05 s time step, the SSP is algebraically updated to simulate the objects' motion. The dashed lines represent the object trajectories decoded from this simulation, while the solid lines represent the ground-truth object trajectories.

where

$$\Delta P_i = X^{x_i+\Delta x_i} \circledast Y^{y_i+\Delta y_i} - f(M \circledast \mathrm{OBJ}_i^{-1}). \tag{3.17}$$

This works because

$$\begin{aligned} &\mathrm{OBJ}_i \circledast X^{x_i} \circledast Y^{y_i} + \mathrm{OBJ}_i \circledast (X^{x_i+\Delta x_i} \circledast Y^{y_i+\Delta y_i} - f(M \circledast \mathrm{OBJ}_i^{-1})) \\ &= \mathrm{OBJ}_i \circledast X^{x_i+\Delta x_i} \circledast Y^{y_i+\Delta y_i}. \end{aligned} \tag{3.18}$$

The drawback of this approach is that multiple operations are needed to update the SSP (one for each encoded object). Otherwise the approach is similar to the previous method, although the update defined here is iterative (subtracting out each past position one at a time), in contrast to having all past position information replaced at once with a new sum. Both algebraic methods can be naturally implemented as a neural network. Figure 7 demonstrates the use of additive shifts to move five objects along distinct trajectories within an SSP. The decoded trajectories are accurate over short time spans but eventually drift due to accumulation of errors. Given its good performance characteristics, we use this method of additive shifts as our favored algebraic approach for simulating the motion of multiple objects (see the benchmarking results described below).

3.3.3 Simulating Multiple Objects with a Learned Model. An alternative to relying on the algebraic properties of SSPs involves approximating a function that repeatedly transforms an SSP by mapping it to a new SSP corresponding to the next point along some trajectory:

$$M \leftarrow g(M, \Delta M), \tag{3.19}$$

where g is a neural network trained to minimize the MSE loss in equation 3.19 from separate training data, and ΔM encodes all of the object velocities as follows:

$$\Delta M = \sum_{i=1}^{m} \mathrm{OBJ}_i \circledast X^{\Delta x_i} \circledast Y^{\Delta y_i}. \tag{3.20}$$

In essence, this method is equivalent to learning the algebraic shifts described above such that the updates are performed in a single "step." We simulate trajectories using a linear model for g (e.g., $g(M, \Delta M) = A \begin{bmatrix} M \\ \Delta M \end{bmatrix} + b$ with matrix $A \in \mathbb{R}^{d \times 2d}$ and bias vector $b \in \mathbb{R}^{d \times 1}$), and with g being a multilayer perceptron (MLP). The MLP's first layer has a number of neurons equal to four times the SSP dimension and is followed by a ReLU nonlinearity, and its second layer's number of neurons is equal to the SSP dimension. An example trajectory containing five objects and simulated with the MLP trained on five trajectories is shown in Figure 8. The first cleanup procedure is used on the output of the MLP at every time step. If the cleanup fails (i.e., a decoded object's position is dissimilar to all SSPs in the area checked), then the position estimate at the previous time step is carried over. This method performs quite well on training data. However, even here, the decoded positions flat-line at times. For some input, unbinding the objects' SPs from the model's output results in a vector that does not lie in the space of SSPs.

3.3.4 Results of Simulating Multiple Objects. To perform an analysis of SSP capacity and accuracy trade-offs when simulating multiobject dynamics, we focus on a comparison between the algebraic approach of performing additive shifts and learned approaches involving a function approximator, $g(M, \Delta M)$.

The models are trained on 100 random trajectories, each with 200 time steps and represented by a sequence of SSPs and velocity SSPs (M_t and ΔM_t). To be precise, these random trajectories are two-dimensional band-limited white noise signals that lie in a two-by-two box. Test performance is reported as the RMSE between the true time series of SSPs representing the trajectory and the trajectories simulated using the algebraic method or g, averaged over 50 random test trajectories.

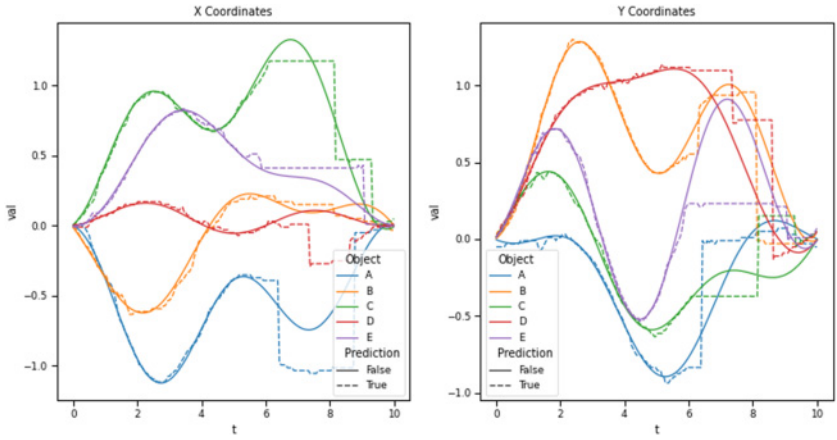


Figure 8: Using a multilayer perciptron (MLP) in combination with the first cleanup method to simulate the trajectories of five different objects within a single SSP. The MLP takes in the last estimate of SSP and the object's velocities represented as a single SSP as input and estimates the SSP at the next time step. Starting with the initial SSP, at each 0.05 s time step, the SSP is updated using the MLP and then "cleaned up" to simulate the objects' motion. The dashed lines represent the object trajectories decoded from this simulation, while the solid lines represent the ground-truth object trajectories.

Figure 9 shows the results of the comparison between the algebraic and learned function approximator approaches while both varying the number of encoded objects while holding the dimensionality of the SSPs fixed, and varying the dimensionality of the SSPs while holding the number of encoded objects fixed. Both the linear model and MLP perform adequately on small training sets but do not generalize to the test data. Interestingly, the linear model performs better than the MLP when the dimensionality of the SSP is high enough.

Overall, this suggests that the algebraic method performs better than the learned models from an MSE perspective. Another advantage of the algebraic approach is that it is compositional, which can be critical for flexibly scaling to larger problems. Specifically, once any one trajectory has been learned, it can be combined with other trajectories in parallel, while adding together the SSP representations. The only limits on the compositionality are determined by the dimensionality of the SSP. As shown in Figure 9, 512 dimensions can effectively handle at least five objects.

In conclusion, by using SSPs, a structured, symbol-like representation, neural networks can be constructed to perform algebraic operations that use this structure to accurately perform computations. The resultant network is completely explainable; it is no black box. Structured representations in

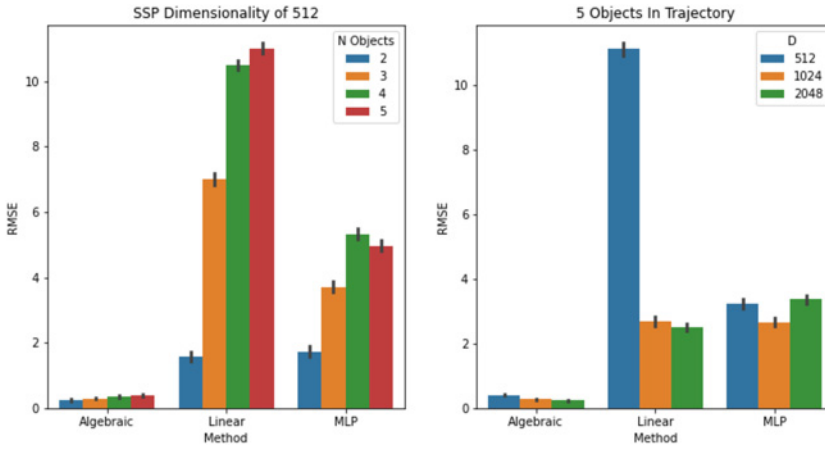


Figure 9: Benchmarking the simulation of separate dynamical trajectories applied to a number of objects encoded in a single SSP. The algebraic approach (i.e., additive shifting) is compared against several methods of learning a model of the trajectory. Left: Model error as a function of the number of objects in the SSP. Right: Model error as a function of SSP dimensionality.

vector form, like SSPs, allow for this marriage of symbol-like reasoning and deep networks.

4 Conclusion

Spatial semantic pointers (SSPs) are an effective means of constructing continuous spatial representations that can be naturally extended to representations of dynamical systems. We have extended recent SSP methods to capture greater spatial detail and encode dynamical systems defined by differential equations or arbitrary trajectories. We have applied these methods to simulate systems with multiple objects and predict the future of trajectories with discontinuous changes in motion.

More specifically, we show that it is possible to represent and continuously update the trajectories of multiple objects with minimal error using a single SSP when that SSP is updated using an appropriate algebraic construction and cleanup. Additionally, we showed that coupling SSPs with the LMU allowed the prediction of the future trajectory of a moving object as it was being observed.

There are several ways in which these results could be extended. For instance, it would be interesting to experiment with dynamics defined over higher-dimensional spaces and with more complex representations of the objects located in these spaces. For instance, it might be possible to define attributes that determine the dynamics of specific object types, such that novel

object dynamics could be predicted solely on the basis of these attributes. It would be useful to extend our work on predicting object interactions to improve both the range of different interaction types that can be predicted and the accuracy with which they can be predicted.

On a more theoretical level, SSPs help to unify currently disparate approaches to modeling intelligent behavior within a common mathematical framework. In particular, the combination of binding operations that support continuous role-filler structure with neural networks provides a point of common ground between approaches to AI that focus on machine learning, dynamical systems, and symbol processing. Exploiting this common ground is likely to be necessary for continued progress in the field over the long term.

Software Packages

Utilities for generating and manipulating spatial semantic pointers (SSPs) are built into the software package NengoSPA (Applied Brain Research, 2020), which supports neural implementations for a number of VSAs and is part of the Nengo ecosystem (Bekolay et al., 2014), a general-purpose Python software package for building and simulating both spiking and nonspiking neural networks.

To integrate Nengo models with deep learning, NengoDL (Rasmussen, 2019) provides a TensorFlow (Abadi et al., 2016) back end that enables Nengo models to be trained using backpropagation, run on GPUs, and interfaced with other networks such as Legendre memory units (LMUs; Voelker et al., 2019). We make use of Nengo, NengoSPA, TensorFlow, and LMUs throughout.

Software and data for reproducing reported results are available at https://github.com/abr/neco-dynamics.

Appendix: Additional Simulation Experiments

A.1 Recursive Spatial Semantic Pointers (rSSPs). An important limitation of SSPs is that $P(x_1, y_1)$ is similar to $P(x_2, y_2)$ when $(x_2 - x_1)^2 + (y_2 - y_1)^2 \leq 2$, that is, the dot product of nearby points is close to one (Komer, 2020). This occurs regardless of the dimensionality (d) of the SP.

More generally, there is a fundamental limitation regarding the kinds of similarity maps that can be represented via the function produced by equation 2.13, as determined by linear combinations of vectors in $\{P(x, y) : (x, y) \in R\}$, $R \subseteq \mathbb{R}^2$; this set of vectors does not necessarily span all of $\mathbb{R}^d$. In other words, since vectors that are encoding different parts of the space are often linearly dependent, there are fewer than d degrees of freedom that determine the kinds of spatial maps that can be represented given fixed axis vectors within finite regions of space.

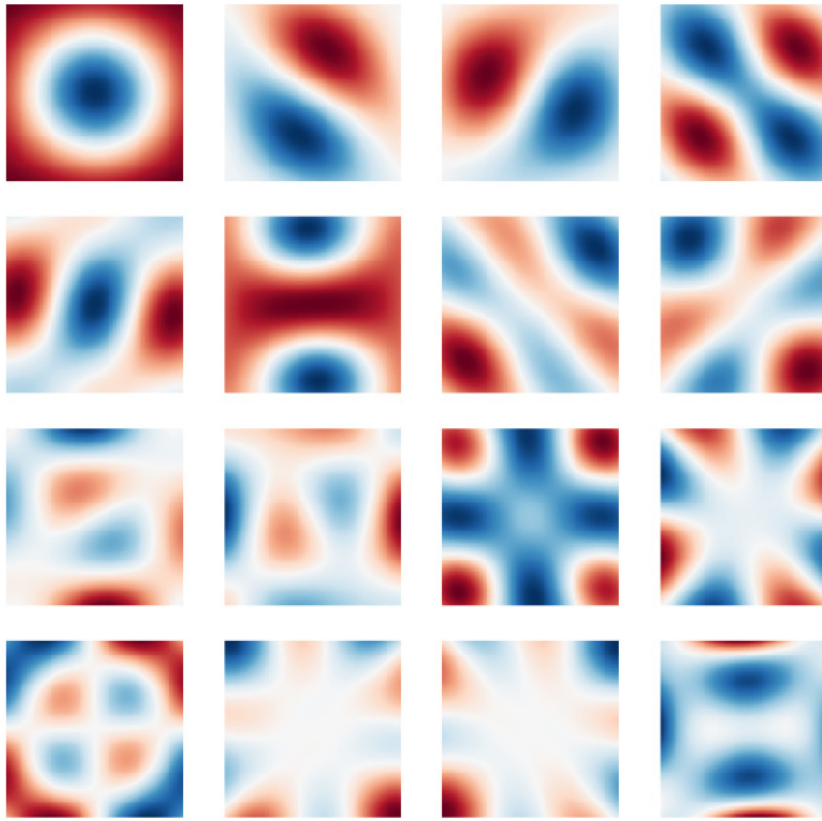


Figure 10: First 16 principal components of $X^x \circledast Y^y$ for $x^2 + y^2 \leq 2$. These basis vectors are independent of dimensionality and independent of the absolute position in space. Only the first 21 vectors have singular values greater than one.

Figure 10 makes this precise by plotting the principal components within a circle of radius $\sqrt{2}$, that is, the left singular vectors of $\{P(x, y) : x^2 + y^2 \leq 2\}$. A circle of radius $\sqrt{2}$ approximates the invertible domain of $\text{sinc}(x)\,\text{sinc}(y)$ (Voelker, 2020; also see Komer, 2020, Figure 3.21). Intuitively, any $P(x, y)$ within this circle can be accurately represented by a linear combination of this spatial basis. We find that—assuming d is sufficiently large—only $\approx$21 principal components are needed to accurately represent this subspace (singular values greater than 1); there are far fewer degrees of freedom than there are dimensions in the SSP (e.g., $d = 1024$ in Figure 10), and, most important, increasing d does not create any new degrees of freedom.

One solution is to simply rescale all (x, y) coordinates such that points that must be nearly orthogonal to one another are separated by a Euclidean distance of at least $\sqrt{2}$. However, this is not practical if the SSP has already been encoded (there is no known way of rescaling the coordinates without first cleaning the SSP up to some particular vocabulary) and leads to unfavorable scaling in d for certain kinds of spatial maps through inefficient use of the continuous surface.

An alternative solution is to vary the axis vectors, (X, Y), as a function of (x, y). A convenient way to do this is to use SSPs in place of each axis vector. We refer to this as a recursive spatial semantic pointer (rSSP). In general, an rSSP uses the following formula to encode positions:

$$P(x, y) = \Phi(x, y; X)^x \circledast \Phi(x, y; Y)^y, \tag{A.1}$$

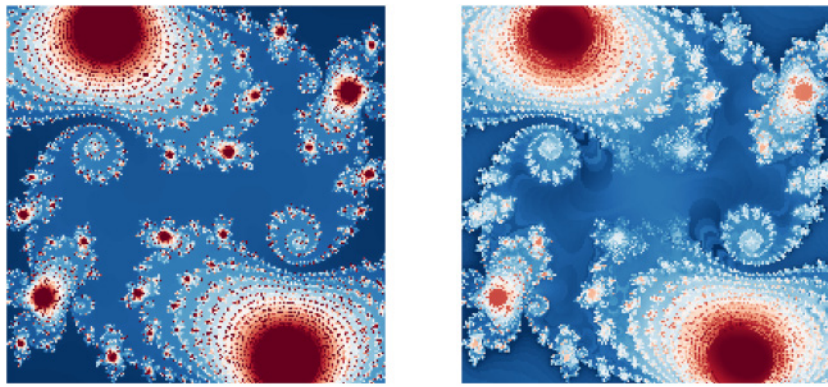
where $\Phi(x, y; A)$ is some function that produces an SSP. $\Phi(\cdot)$ may be predetermined or computed by a neural network—feedforward or recurrent—that is optionally trained via backpropagation. For example,

$$\Phi(x, y; A) = A^{|h(x,y)|/3+2} \tag{A.2}$$

generates axes that oscillate between A^2 and A^3 using the hexagonal lattice from equations 2.15 and 2.16, which are then used to encode (x, y) as usual via equation A.1. We find that this increases the degrees of freedom from $\approx$21 to $\approx$36 within a radius of $\sqrt{2}$, given the same dimensionality, map scale, and singular value cutoff.

In essence, this formulation enables the encoding to use different axis vectors in different parts of the space. Since $(A^a)^b \neq A^{ab}$ in general for $a, b \in \mathbb{R}$, this is not the same as rescaling (x, y). Rather, since $\Phi(\cdot)$ is a nonlinear operation with potentially infinite complex branching, this increases the representational flexibility of the surface by decreasing the linear dependence in $P(x, y)$ and thus allows rSSPs to capture fine-scaled spatial structures through additional degrees of freedom.

We illustrate by encoding a fractal from the Julia (1918) set—a complex dynamical system—using a single rSSP ($d = 1{,}024$). Specifically we define $\Phi(x, y; A) = A^{J(x+iy)}$ where $J(z)$ is an affine transformation of the number of iterations required for $z \leftarrow z^2 + c$ to escape the set $|z| \leq R$ ($c = -0.1 + 0.65i$, $R \approx 1.4527$, 1000 iterations). We then solve for the vector that best reconstructs $J(x + iy)$ in its similarity map by optimizing a regularized least-squares problem, which is possible because equation 2.13 is a linear operator. The solution produces a map that is perceptually quite similar to the ideal fractal (see Figure 11), thus demonstrating the effectiveness of rSSPs in representing intricate spatial structures using relatively few dimensions.



(a) Ideal Julia Set Fractal

(b) Recursive Spatial Semantic Pointer (rSSP)

Figure 11: Representing a fractal from the Julia set within the similarity map of a recursively-generated SSP ($d = 1{,}024$). Rendered images are 501×501 (i.e., 251,001 pixels). See text for details.

A.2 Axis-Specific Algebraic SSP Updates. One option for applying different updates to different objects encoded in an SSP involves representing the axes of the plane for each object separately as follows:

$$\text{SSP} = X_1^{x_1} \circledast Y_1^{y_1} + X_2^{x_2} \circledast Y_2^{y_2} \ldots \tag{A.3}$$

With this representation, it is possible to define a transformation SSP that acts on each object separately:

$$\text{SSP} = \text{SSP} \circledast (X_1^{\Delta x_1} \circledast Y_1^{\Delta y_1} + X_2^{\Delta x_2} \circledast Y_2^{\Delta y_2} \ldots). \tag{A.4}$$

The effect of applying this transformation is to produce an updated SSP with $m^2 - m$ noise terms if m is the number of encoded objects, since each term in the bracketed sum would apply to a corresponding term in SSP and combine with the other $m - 1$ terms in the SSP to yield noise. There are m terms in the bracketed sum, so the creation of $m - 1$ noise terms occurs m times, leading to a total of $O(m^2)$ noise terms overall. Such scaling is unlikely to permit the successful manipulation of SSPs encoding large numbers of objects, even if the noise is zero-mean.

Formally, we can describe the growth of noise terms over time as follows: as just mentioned, each term in the bracketed sum described above applies to one corresponding term in SSP and combines with the other $m - 1$ terms in SSP to yield noise. There are m terms in the bracketed sum, so the creation

of $m-1$ noise terms occurs m times, leading to a total of m^2-m noise terms after a single time step. On the next time step, each term in the bracketed sum applies to one corresponding term in the SSP, and then combines with the $m-1$ other encoding terms and the m^2-m noise terms created by the previous time step; again, these combinations occur m times. This yields a total of $m(m-1+m^2-m = O(m^3)$ noise terms on the second time step. After t time steps, the number of noise terms is $O(m^{t+1})$.

Overall, axis-specific shifts are unsatisfactory in the absence of cleanup methods, since the noise terms compound at each time step, quickly pushing the signal-to-noise ratio towards zero.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., . . . Zheng, X. (2016). TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation* (pp. 265–283), PeerJ.

Applied Brain Research. (2020). Nengo spa. https://www.nengo.ai/nengo-spa/.

Bekolay, T., Bergstra, J., Hunsberger, E., DeWolf, T., Stewart, T. C., Rasmussen, D., & Eliasmith, C. (2014). Nengo: A Python tool for building large-scale functional brain models. *Frontiers in Neuroinformatics*, *7*, 48.

Blouw, P., Solodkin, E., Thagard, P., & Eliasmith, C. (2016). Concepts as semantic pointers: A framework and computational model. *Cognitive Science*, *40*(5), 1128–1162.

Bottou, L. (2014). From machine learning to machine reasoning: An essay. *Machine Learning*, *94*(2), 133–149.

Choo, X. (2010). *The ordinal serial encoding model: Serial memory in spiking neurons*. Master's thesis, University of Waterloo.

Choo, X. (2018). *Spaun 2.0: Extending the World's Largest Functional Brain Model*. PhD diss., University of Waterloo.

Co-Reyes, J. D., Liu, Y., Gupta, A., Eysenbach, B., Abbeel, P., & Levine, S. (2018). *Self-consistent trajectory autoencoder: Hierarchical reinforcement learning with trajectory embeddings*. arXiv:1806.02813.

Conklin, J., & Eliasmith, C. (2005). A controlled attractor network model of path integration in the rat. *Journal of Computational Neuroscience*, *18*(2), 183–203.

Crawford, E., Gingerich, M., & Eliasmith, C. (2015). Biologically plausible, human-scale knowledge representation. *Cognitive Science*, *40*, 782–821.

Dumont, N. S.-Y., & Eliasmith, C. (2020). Accurate representation for spatial cognition using grid cells. In *Proceedings of the 42nd Annual Meeting of the Cognitive Science Society*. Red Hook, NY: Curran.

Eliasmith, C. (2005). A unified approach to building and controlling spiking attractor networks. *Neural Computation*, *17*(6), 1276–1314.

Eliasmith, C. (2013). *How to build a brain: A neural architecture for biological cognition*. New York: Oxford University Press.

Eliasmith, C., & Anderson, C. H. (2003). *Neural engineering: Computation, representation, and dynamics in neurobiological systems*. Cambridge, MA: MIT Press.

Eliasmith, C., Stewart, T. C., Choo, X., Bekolay, T., DeWolf, T., Tang, Y., & Rasmussen, D. (2012). A large-scale model of the functioning brain. *Science*, *338*, 1202–1205.

Fodor, J., & Pylyshyn, Z. (1988). Connectionism and cognitive architecture: A critical analysis. *Cognition*, *28*(1–2), 3–71.

Foldiak, P. (2003). Sparse coding in the primate cortex. In M. A. Arbib (Ed.), *The handbook of brain theory and neural networks*. Cambridge, MA: MIT Press.

Frady, E. P., Kleyko, D., & Sommer, F. (2020). *Variable binding for sparse distributed representations: Theory and applications*. arXiv:abs/2009.06734.

Frady, P., Kanerva, P., & Sommer, F. (2018). A framework for linking computations and rhythm-based timing patterns in neural firing, such as phase precession in hippocampal place cells. In *Proceedings of the Conference on Cognitive Computational Neuroscience*.

Gayler, R. (2004). *Vector symbolic architectures answer Jackendoff's challenges for cognitive neuroscience*. arXiv preprint cs/0412059.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*, vol. 1. Cambridge, MA: MIT Press.

Gosmann, J. (2018). *An integrated model of context, short-term, and long-term memory*. PhD diss., University of Waterloo.

Gosmann, J., & Eliasmith, C. (2019). Vector-derived transformation binding: An improved binding operation for deep symbol-like processing in neural networks. *Neural Computation*, *31*(5), 849–869.

Gosmann, J., & Eliasmith, C. (2020). CUE: A unified spiking neuron model of short-term and long-term memory. *Psych. Review*, *128*, 104–124.

Gosmann, J., Voelker, A. R., & Eliasmith, C. (2017). A spiking independent accumulator model for winner-take-all computation. In *Proceedings of the 39th Annual Conference of the Cognitive Science Society*. London, UK: Cognitive Science Society.

Hadley, R. (2009). The problems of rapid variable creation. *Neural Computation*, *21*, 510–532.

Julia, G. (1918). Memoire sur l'itération des fonctions rationnelles. *J. Math. Pures Appl.*, *8*, 47–245.

Kanerva, P. (2009). Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors. *Cognitive Computation*, *1*(2), 139–159.

Komer, B. (2020). *Biologically inspired spatial representation*. PhD diss., University of Waterloo.

Komer, B., & Eliasmith, C. (2020). Efficient navigation using a scalable, biologically inspired spatial representation. In *Proceedings of the 42nd Annual Meeting of the Cognitive Science Society*. London, UK: Cognitive Science Society.

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

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, *521*(7553), 436–444.

Lu, T., Voelker, A. R., Komer, B., & Eliasmith, C. (2019). Representing spatial relations with fractional binding. In *Proceedings of the 41st Annual Meeting of the Cognitive Science Society*. London, UK: Cognitive Science Society.

Marcus, G. (1998). Rethinking eliminative connectionism. *Cognitive Psychology*, *37*, 243–282.

Marcus, G. F. (2019). *The algebraic mind: Integrating connectionism and cognitive science*. Cambridge, MA: MIT Press.

McClelland, J., Botvinick, M., Noelle, D., Plaut, D., Rogers, T., Seidenberg, M., & Smith, L. (2010). Letting structure emerge: Connectionist and dynamical systems approaches to cognitive modelling. *Trends in Cognitive Sciences*, *14*(8), 348–356.

Mirus, F., Stewart, T., & Conradt, J. (2020). Analyzing the capacity of distributed vector representations to encode spatial information. In *Proceedings of the 2020 International Joint Conference on Neural Networks* (pp. 1–7). Piscataway, NJ: IEEE.

Moser, E. I., Kropff, E., & Moser, M.-B. (2008). Place cells, grid cells, and the brain's spatial representation system. *Annu. Rev. Neurosci.*, *31*, 69–89.

Muller, M. E. (1959). A note on a method for generating points uniformly on *n*-dimensional spheres. *Comm. Assoc. Comput. Mach.*, *2*(1–2), 19–20.

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

Rasmussen, D. (2019). NengoDL: Combining deep learning and neuromorphic modelling methods. *Neuroinformatics*, *17*(4), 611–628.

Rasmussen, D., & Eliasmith, C. (2011). A neural model of rule generation in inductive reasoning. *Topics in Cognitive Science*, *3*(1), 140–153.

Schlegel, K., Neubert, P., & Protzel, P. (2020). *A comparison of vector symbolic architectures*. arXiv:2001.11797.

Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural Networks*, *61*, 85–117.

Schöner, G. (2014). Embodied cognition, neural field models of. In *Encyclopedia of computational neuroscience* (pp. 1084–1092). Berlin: Springer.

Smolensky, P. (1990). Tensor product variable binding and the representation of symbolic structures in connectionist systems. *Artificial Intelligence*, *46*(1–2), 159–216.

Smolensky, P., & Legendre, G. (2006). *The harmonic mind: From neural computation to optimality-theoretic grammar*. Cambridge, MA: MIT Press.

Sorscher, B., Mel, G., Ganguli, S., & Ocko, S. (2019). A unified theory for the origin of grid cells through the lens of pattern formation. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, & R. Garnett (Eds.), *Advances in neural information processing systems*, *32* (pp. 10003–10013). Red Hook, NY: Curran.

Stewart, T., Bekolay, T., & Eliasmith, C. (2011). Neural representations of compositional structures: Representing and manipulating vector spaces with spiking neurons. *Connection Science*, *23*, 145–153.

Stewart, T. C., Choo, X., & Eliasmith, C. (2014). Sentence processing in spiking neurons: A biologically plausible left-corner parser. In *Proceedings of the 36th Annual Conference of the Cognitive Science Society* (pp. 1533–1538). London, U.K.: Cognitive Science Society.

Stewart, T. C., Tang, Y., & Eliasmith, C. (2011). A biologically realistic cleanup memory: Autoassociation in spiking neurons. *Cognitive Systems Research*, *12*(2), 84–92.

Voelker, A. R. (2019). *Dynamical systems in spiking neuromorphic hardware*. PhD diss., University of Waterloo.

Voelker, A. R. (2020). *A short letter on the dot product between rotated Fourier transforms*. arXiv:2007.13462.

Voelker, A. R., Kajić, I., & Eliasmith, C. (2019). Legendre memory units: Continuous-time representation in recurrent neural networks. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, & R. Garnett (Eds.), *Advances in neural information processing systems*, *32* (pp. 15544–15553). Red Hook, NY: Curran.

Weiss, E., Cheung, B., & Olshausen, B. (2016). A neural architecture for representing and reasoning about spatial relationships. In *Proceedings of the International Conference on Learning Representations—Workshop Track*. La Jolla, CA: ICLR.

Received November 4, 2020; accepted March 15, 2021.