SYDE 556/750
Simulating Neurobiological Systems
Lecture 7: Temporal Basis Functions

Andreas Stöckel and Chris Eliasmith

Based on lecture notes by
Chris Eliasmith and Terrence C. Stewart

October 21, 2024

Accompanying Readings: Chapter 3 of Neural Engineering; see references

Contents

1 Introduction **1**

2 Representing Functions **2**

2.1 Sampling 3

2.2 Basis Functions 6

3 Representing the Past: The Delay Network **8**

3.1 Implementing Delays as a Dynamical System 8

3.2 Example: Implementing the Delay Network in Nengo 11

1 Introduction

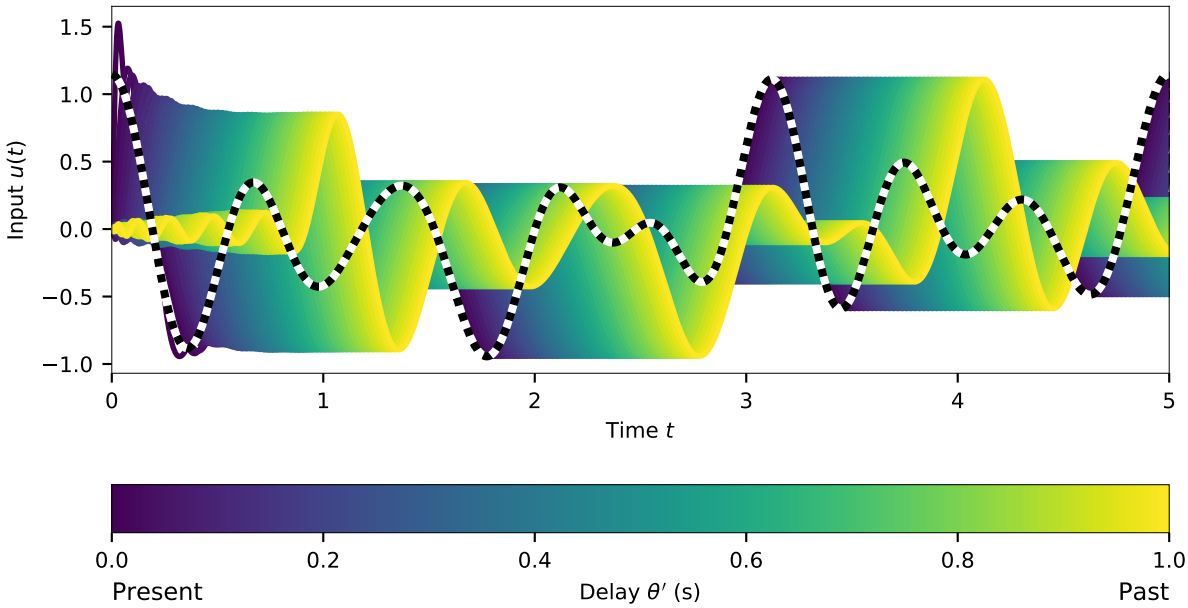


Figure 1: Example of the "delay network" we are going to discuss in the lecture. The dotted line is the input $u(t)$. The coloured lines correspond to delayed versions of the input signal. These delayed versions are all decoded from the same function representation $f_{[t-\theta,t]}(\theta')$. This diagram has been generated by decoding from a mathematically perfect implementation of the delay network. ⌨ *Code*

Note: In this lecture we discuss a recent application of the Neural Engineering Framework, the so called Delay Network developed by Aaron Voelker, first presented in [1], and discussed in more detail in his PhD thesis [2]. The Delay Network represents a function over time in a neural population. A more general version of the Delay Network has been presented at NeurIPS 2019 as the "Legendre Memory Unit" (LMU) [3]. The LMU can be used as a component within artificial Deep Neural Networks and outperforms other recurrent architectures such as LSTMs in a variety of tasks.

As humans, we often feel as if we have a good recollection of events that happened in the immediate past, where in this lecture, "immediate" refers to events happening within a few seconds or even a fraction of a second. As events move into the more distant past, it becomes increasingly harder to recall details. In this lecture, we discuss a system that similarly memorizes stimuli, remembers them for a certain period, and then gradually "forgets" them.

Mathematically speaking, what we would like to have a function $f_{[t-\theta,t]}(\theta')$ which allows us to access stimuli $u(t)$ in a time-interval from $[t-\theta, t]$, that is

$$f_{[t-\theta,t]}(\theta') = u\left(t-\theta'\right), \text{where } 0 \leq \theta' \leq \theta\,.$$

Put differently, $f_{[t-\theta,t]}(0)$ will return the present stimulus $u(t)$, whereas $f_{[t-\theta,t]}(\theta)$ will return the input θ seconds from the past.

We would like to build a biologically plausible version of such a system. That is, we would like to represent information about the immediate past in the current activity of a population of neurons $f_{[t-\theta,t]}$. In order to do this, we have to solve two problems that we discuss separately.

1. **Function representation.** So far we have seen how we can represent vectorial quantities in a neural population. But how can we represent an entire *function* of the form $f_{[t-\theta,t]}(\theta')$ in a neural ensemble?
2. **Updating the function representation.** If we were somehow able to represent functions in our neural population we still need to know how exactly to update this representation over time.

Note: Function *representation* is different from the "transformation" principle we already discussed. In short, we are not interested in computing a function $\mathbf{y} = f(\mathbf{x})$, but instead we would like to *represent* the function f itself in a neural population, i.e. have some way of storing a mapping from a value $\mathbf{x}$ onto values $\mathbf{y}$.

2 Representing Functions

For simplicity, and because it describes the problem we are trying to solve, let's focus on scalar functions over time $f(t)$, i.e., $f : \mathbb{R} \longrightarrow \mathbb{R}$. How can we represent an interval $[0, T)$ of such a function in a neural population? We already know how a neural population can represent a vector, so let's rephrase the question. How can we represent functions as vectors?

First of all, if we have a parametrised function family $f(t; \mathbf{x})$ we want to represent (i.e., all linear functions, all affine functions, all Normal distributions with a mean μ and standard-deviation σ), then the function parameters can be described as a vector $\mathbf{x}$ that we could use:

$$
\begin{aligned}
&f(t;\mathbf{x}) = mt, && \text{Linear function } \Rightarrow \mathbf{x} = \Big(m\Big),\\
&f(t;\mathbf{x}) = mt + b, && \text{Affine function } \Rightarrow \mathbf{x} = \Big(m, b\Big),\\
&f(t;\mathbf{x}) = \exp\left(-\frac{(t-\mu)^2}{\sigma^2}\right), && \text{Gaussian } \Rightarrow \mathbf{x} = \Big(\mu, \sigma\Big).
\end{aligned}
$$

If we represent $\mathbf{x}$ and t in the same neural population, we can evaluate $f(t; \mathbf{x})$ in the connection from the pre- to the post-population according to NEF principle two.

In general, if we do not have any more information about the kind of function we would like to represent, there are two (closely related) approaches to representing functions as vectors: sampling/discretisation and basis functions. We discuss these two approaches in the following.

Note: We can generalise what we discuss to multi-variate functions with vectorial output, i.e. $f : \mathbb{R}^d \longrightarrow \mathbb{R}^{d'}$. We can treat multi-dimensional *output* $d' > 1$ as d' independent functions $f_1, \ldots, f_{d'}$. For multi-dimensional *inputs* $d > 1$, we need to sample on a higher-dimensional grid. When using basis functions, we need an appropriate basis $\varphi_i : \mathbb{R}^d \longrightarrow \mathbb{R}$.

2.1 Sampling

The idea of sampling is to measure the value of a function $f(t)$ at discrete, equally spaced points $\mathbf{x} = (x_0, \ldots, x_{N-1})$, where

$$x_i = f(t_i) = f(\Delta t i), \qquad \text{and } \Delta t \text{ is the sampling interval.}$$

The resulting vector $\mathbf{x}$ holds a representation of $f(t)$ over the interval $[0, T)$, where $T = N\Delta t$.

If we wanted to reconstruct the value of the function at a point t that was not sampled, we can in theory use one of many interpolation techniques to "guess" function values in between samples. If our function is somewhat well-behaved, we would expect the quality of these reconstructions to get better for smaller Δt. That is, the more sample points N we have over the interval $[0, T)$, the better the representation of the function (fig. 3).

Unfortunately, in general, for an arbitrary function $f(t)$, we need an infinite number of sample points to perfectly represent it. Mathematically speaking this is true for any infinitesimally small interval $T \to 0$. A function can "drastically" change its value between any two infinitesimally close points.

Example: These restrictions even apply to continuous functions! The earliest example of a "weird" continuous function that is not differentiable (i.e., not "smooth") at any point is the Weierstrass function published in 1872, defined as

$$f(x) = \sum_{n=0}^{\infty} a^n \cos(\pi b^n x), \quad \text{where } 0 < a < 1, b \text{ is a positive odd integer, and } ab > 1 + \frac{3}{2}\pi.$$

Intuitively, this function is continuous at any point—after all, it is just a sum of cosines (be careful though; counter-intuitively, an *infinite* sum of continuous functions *can* be discontinuous). One can show that this function is not differentiable at any point.

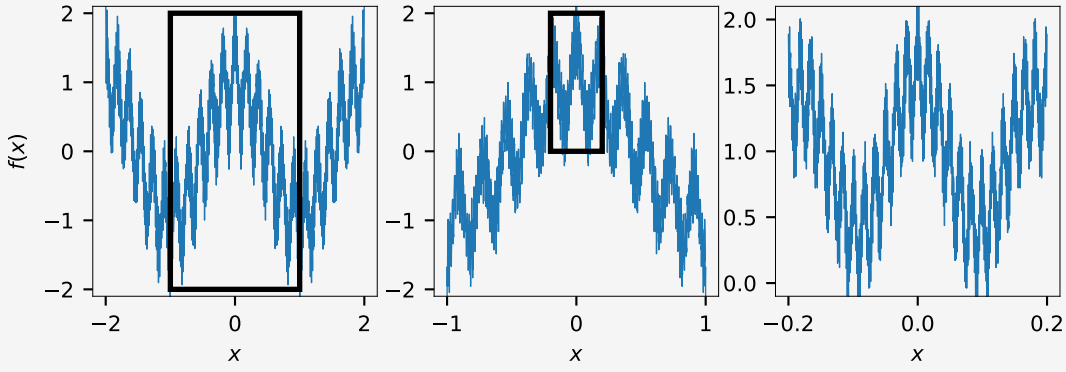
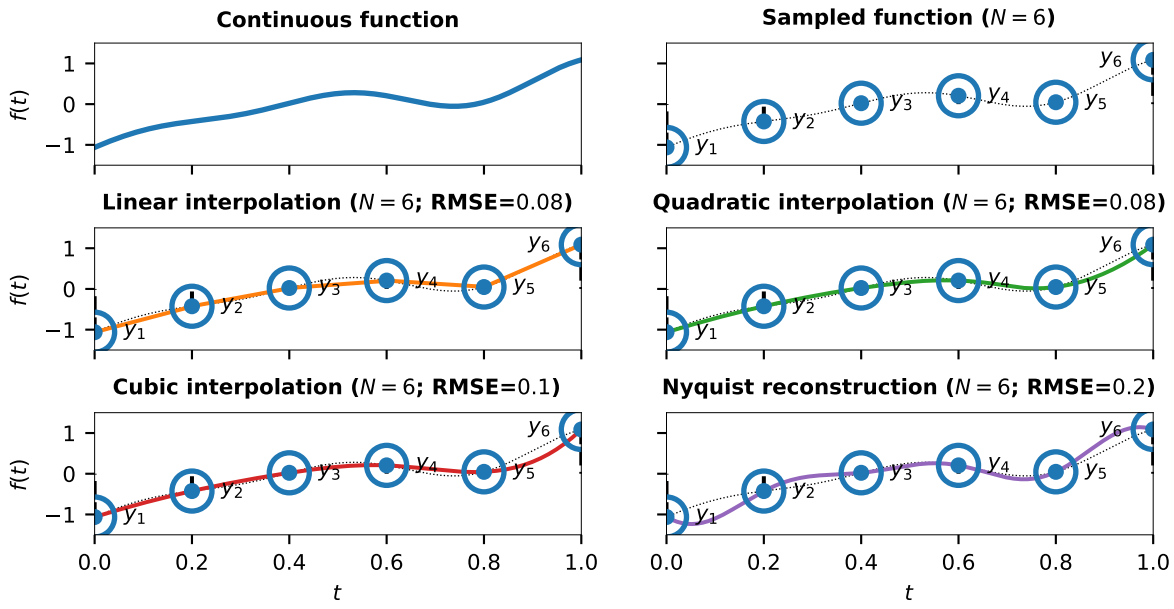
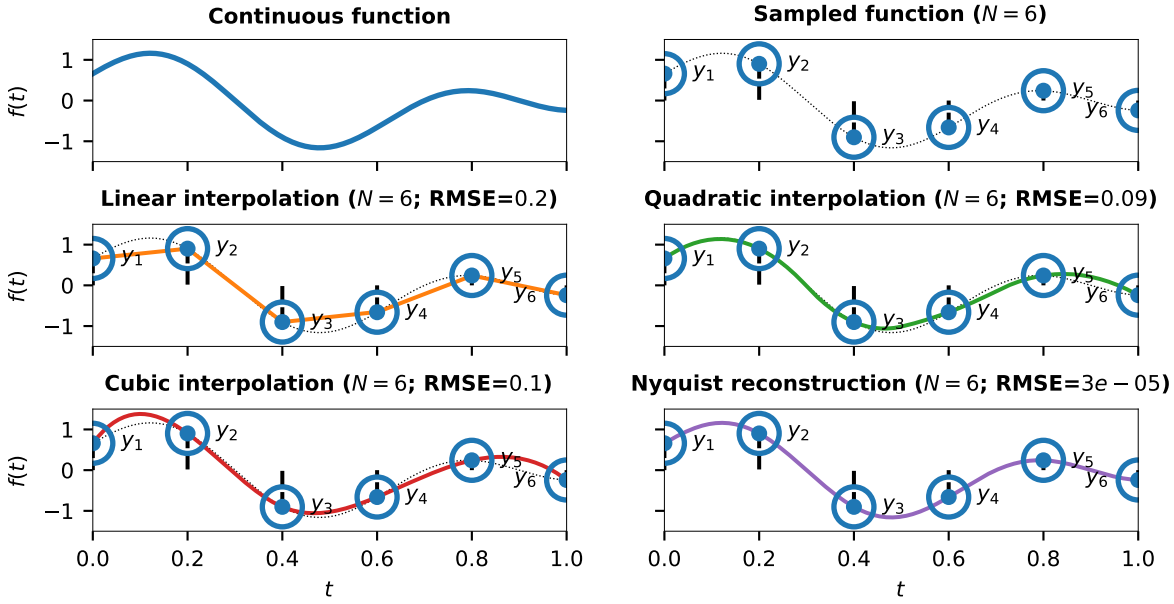


Figure 2. Visualisation of the Weierstrass function for $b = 11$, $a = \frac{1+\frac{3}{2}\pi}{10}$. Black rectangles correspond to the region shown in the neighbouring plot to the right. No matter how far we "zoom" into the function, there is an infinite amount of detail we would have to capture when sampling (this is one of the properties of a *fractal*; note that this does not necessarily imply an infinite amount of *information* in this function; the function is fully determined by two parameters a, b). *Code*



(a) Example 1: Bandlimit above 3 Hz



(b) Example 2: Bandlimit below 3 Hz

Figure 3: Example illustrating sampling. A continuous function is sampled at $N = 6$ points (represented as a "lollipop" plot). We can use various interpolation techniques to reconstruct ("guess") the values between individual sample points. In case the original function is bandlimited to a frequency below $N/2T = 3$ Hz (as in **(b)**, but not **(a)**), the sample points uniquely define the function according to the Nyquist-Shannon sampling theorem.

Guarantees for band-limited signals While the above restriction—namely, that we need infinitely many points to represent any infinitesimally small interval of a function—is true for general, mathematical functions, we have a much better guarantee regarding the required number of required for "physical", i.e., band-limited, signals.

This guarantee is the Nyquist-Shannon sampling theorem, which connects the discrete, sampled world and continuous functions.

The Nyquist-Shannon Sampling Theorem
If $f(t)$ contains no frequencies greater than B then it is *completely* determined by samples spaced $\Delta t = \frac{1}{2B}$ apart ($N = 2BT$ equally spaced samples for a time-slice $[0, T)$). There is a *one-to-one mapping* between the samples $\mathbf{x}$ and the function $f(t)$.

Note the emphasis on "one-to-one mapping": if we have $N = 2BT$ equally spaced samples of a function with band-limit B, we can completely reconstruct the function without any losses. Of course, if we have access to the function, we can measure the sample points.

In summary, this means that there is hope that we can represent physical signals by storing just a few values. If we can guarantee that our function does not have any frequencies above B, we only need to store $N = 2BT$ samples; or, in other words we need to sample with a frequency of $f_s = 2B$ (the "Nyquist Frequency"). So for a signal that is limited to 5 Hz we only need 10 samples per second to be able to perfectly reconstruct it. Conversely, there is no reason to store significantly more than N samples—doing so would just be "wasting space".

Aside: The Nyquist-Shannon Sampling Theorem and Audio Signals. There are some engineering related reasons for sampling slightly faster than the Nyquist frequency. For example, the absolute hearing threshold for (young) humans is about $B = 20\,\text{kHz}$. Correspondingly, to record an audio signal meant for humans (i.e., music or speech recordings), we can do the following: (1) band-limit the original signal to 20 kHz in the capture device, (2) sample at $f_s = 40\,\text{kHz}$. This allows us to perfectly reconstruct the band-limited signal according to the Nyquist-Shannon sampling theorem. Since frequencies above 20 kHz are not audible, the result will appear to humans exactly as the original signal.

However, this only works if we can trust our capture device (i.e., the microphone and its amplifier circuit) to *sharply* cut off all frequencies above 20 kHz. Unfortunately, such perfect filters are impossible to implement as an analogue device. Frequencies slightly above B will still pass through, albeit being attenuated. This violates the pre-condition of the Nyquist-Shannon sampling theorem, leading to imperfect reconstructions (an effect known as *aliasing*). Hence, it is better to sample faster than Nyquist, to give room for frequencies in the signal that are (slightly) above B.

This is why common sampling rates for audio signals are 44.1 kHz (CD audio), and 48 kHz (DVD audio). Much higher sampling of audio signals intended for human listening make no sense, except for intermediate signals (i.e., for sidestepping the imperfect analogue filter problem by sampling at a very high rate and applying a cheap and precise digital filter).

Note: *Online resources.* For more information on the Nyquist-Shannon sampling theorem in general, and its implications with respect to storing audio signals in particular, refer to this material by Chris Montgomery (author/co-author of the Ogg Vorbis and Opus codecs):

- An excellent Video on the Nyquist Shannon sampling theorem, as well as quantisation: Xiph.org Digital Show and Tell, Episode 2.
- An article on "24 bit / 192 kHz Music Downloads and why they make no sense"; this may seem tangential but explains the above in much more detail.

Note: *Algorithm for the "Nyquist Reconstruction" of Functions.* The Nyquist-Shannon sampling theorem tells us that it is possible to perfectly reconstruct a function with band limit B as long as we have $N = 2BT$ equally spaced sample points. But how do we compute this reconstruction for discrete signals in practice? Put differently, how do we convert N sample points $x'_0, \dots, x'_N$ into a densely sampled signal $x_0, \dots, x_M$ with $M \geq N$. The answer is to use the Fourier transformation of the sample points.

First consider the reverse direction. We're given a signal with band limit B and $M \geq N$ equally spaced sample points $x_0, \dots, x_M$. We can then compute the corresponding Fourier coefficients $\omega_{-M/2}, \dots, \omega_0, \dots \omega_{M/2}$. If a function is band-limited with band-width B, this means that only the $N = 2BT$ frequency coefficients $\omega_{-BT}, \dots, \omega_0, \dots, \omega_{BT}$ are nonzero. Discarding the zero-coefficients and converting the remaining N frequency coefficients back to the time-domain results in $2BT$ equally spaced sample points in the time domain, $x'_0, \dots, x'_N$.

Hence, if we want to convert $x'_0, \dots, x'_N$ to $x_0, \dots, x_M$ with $M \geq N$, we perform the above steps in reverse: we convert $x'_0, \dots, x'_N$ to the Fourier domain, pad with leading and trailing zeros so we have M frequency coefficients, and convert to the time-domain.

In practice, there are more efficient ways to do this (i.e., interlacing the signal with zeros, applying the right FIR/IIR filter and selecting M samples). This process is also known as *sampling rate conversion*.

2.2 Basis Functions

Another popular way of representing functions as vectors is to express them in terms of a linear combination of a set of basis functions. Let $\varphi_1, \dots, \varphi_q$ be functions $\mathbb{X} \subset \mathbb{R} \longrightarrow \mathbb{R}$. In case we chose these basis functions well, we can approximate a wide range of f with a low error over some interval $\mathbb{X}$ by linearly combining the basis functions:

$$f(t) \approx \hat{f}(t) = \sum_{i=1}^{q} x_i \varphi_i(t). \tag{1}$$

The vector $\mathbf{x} = (x_1, \dots, x_q)$ now encodes the function.

A different way of reading the above equation is that we can evaluate the encoded function $\hat{f}(t)$ at a point t by computing a dot product $\langle \boldsymbol{\alpha}, \mathbf{x} \rangle$, where $\alpha_i = \varphi_i(t)$. Put differently, if we

know the point t at which we want to evaluate $\hat{f}(t)$, we can pre-compute a $\boldsymbol{\alpha}$ to decode the represented function at this point.

Computing the vector coefficient of x for orthonormal function bases. Given the above, how do we compute the $\mathbf{x}$ that encodes a function f. To a degree, this depends on the choice of the basis functions. In the special case of an *orthonormal* function basis, that is

$$\langle \varphi_i, \varphi_j \rangle = \int_{\mathbb{X}} \varphi_i(t)\varphi_j(t)\,\mathrm{d}t = \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{otherwise}, \end{cases}$$

then each coefficient of $\mathbf{x} = (x_1, \ldots, x_q)$ is just given as

$$x_i = \langle \varphi_i, f \rangle = \int_{\mathbb{X}} f(t)\varphi_i(t)\,\mathrm{d}t\,. \tag{2}$$

Note: *Relationship between sampling and basis functions.* Sampling can be seen as a special case of using basis functions. In particular, if we choose a set of Dirac-δ functions $\varphi_i(t) = \delta(t - \Delta t i)$ as basis functions, then eq. (2) will exactly "read out" the function values at $\Delta t i$. This is just a consequence of the definition of the Dirac-δ (see the notes for lecture two and four); it holds

$$x_i = \int_{\mathbb{X}} f(t)\delta(t - \Delta t i)\,\mathrm{d}t = f(\Delta t i)\,.$$

Note: *Basis functions and the Fourier/Laplace transformation.* Notice that this is very similar to the equation underlying the Fourier or Laplace transformations that we saw earlier in the course. This is exactly because these transformations can be described in terms of an orthonormal function basis.

The same equation—i.e., computing the dot-product between the value that should be represented and the basis—is also used when representing a vector in terms of an orthonormal vector basis. In a sense, we are just computing "how similar" the represented value is to each basis vector/function, giving us the vector coefficients with respect to the new basis.

Computing the vector coefficients of x in the general case. In the general case, we can compute the coefficients of $\mathbf{x}$ by sampling and solving a least squares problem

$$\mathbf{x} = \alpha \arg\min_{\mathbf{x}} \sum_{k=1}^{N} \sum_{i=1}^{q} \big(f(t_k) - x_i\varphi_i(t_k)\big)^2\,,$$

where $t_1, \ldots, t_N$ are a set of N sample points. Note that this is exactly the optimization problem we solve when computing population decoders $\mathbf{D}$. This is not a coincidence; the tuning curves in a neuron ensemble form a set of basis functions.

Popular basis functions So what are some popular basis functions that people use to represent functions over time? As mentioned above, we have already seen the Fourier and Laplace transformation, which—at least in their discrete versions—can be thought of as a basis transformation.

Another popular (orthonormal) basis is the discrete cosine basis and the associated discrete cosine transformation (DCT, cf. fig. 4). There are multiple versions of the DCT basis. One particular version is given as

$$\varphi_i(t) = \cos\big((i-1)\pi t\big) \quad \text{for } t \in \mathbb{X} = [0, 1]\,. \qquad \textit{(Discrete Cosine Basis)}$$

Another popular set of basis functions (which are orthogonal, but not ortho*normal*) is the Legendre basis (fig. 5). This is a so called "polynomial" basis, which means that the ith basis function is a polynomial of order i when counting from zero. The (shifted) Legendre basis is given as

$$\varphi_i(t) = \tilde{P}_i(t) = (-1)^i \sum_{k=0}^{i} \binom{i}{k}\binom{i+k}{k}(-t)^k \quad \text{for } t \in \mathbb{X} = [0, 1]\,. \quad \textit{(Shifted Legendre Basis)} \tag{3}$$

These are shifted in the sense that they are symmetric over [0,1] rather than [-1,1].

3 Representing the Past: The Delay Network

We discussed two techniques for representing functions as vectors. The first technique, sampling, has taught us that there are mathematical guarantees regarding the representation of bandlimited functions. As long as our function f is band-limited, we know that we can perfectly represent an interval $\mathbb{X}$ as a finite number of coefficients $\mathbf{x} = (x_1, \ldots, x_N)$. The higher the maximum frequency B in the signal, the more coefficients are required to represent it. This relationship is less clear for the second technique we discussed, basis functions, but in practice, since many function space representations map higher frequencies onto basis functions $\varphi_i(t)$ higher indices i, similar constraints hold here as well.

However, it is still unclear how to *encode* a function f in a neural ensemble as a vector $\mathbf{x}$. If we were given a set of samples $x_1, \ldots, x_N$, we could either use those samples directly as our function representation, or alternatively, if we wanted to represent these points with respect to an orthonormal/orthogonal function basis, we can use a linear transformation to transform the samples into the corresponding function space.

This still does not answer the question of where these samples come from. In particular, given the goal that we want to represent a sliding window over the past, we need a technique to somehow readjust our representation $\mathbf{x}$ in each time step.

3.1 Implementing Delays as a Dynamical System

In order to solve this problem, let's start fresh and think about this from a different perspective—we will come back to what we discussed in the previous section at the end.

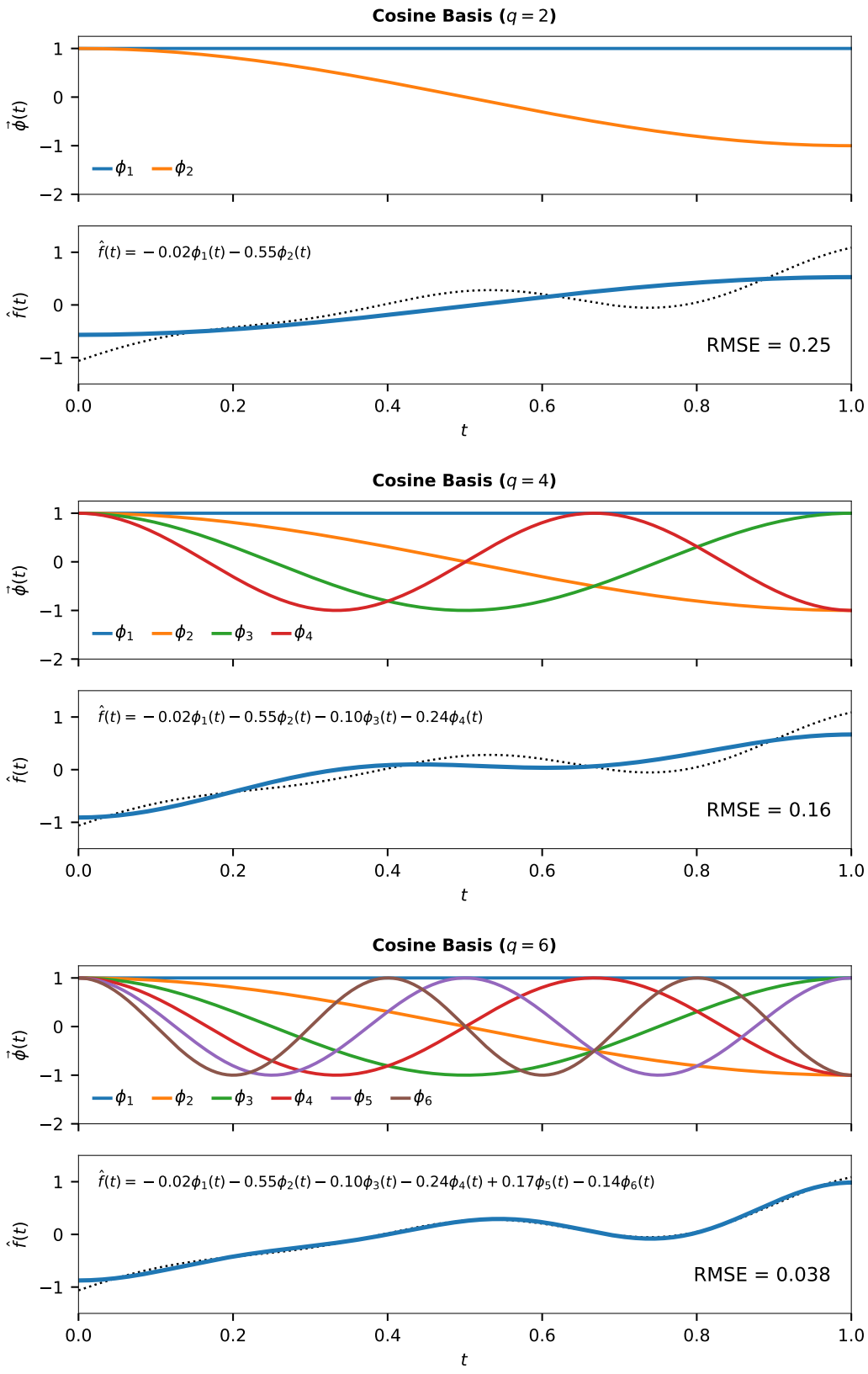


Figure 4: Representing a function (dotted line in the bottom plots) with a variable number q of cosine basis functions over the interval $\mathbb{X} = [0, 1]$. The error goes to zero as q increases.

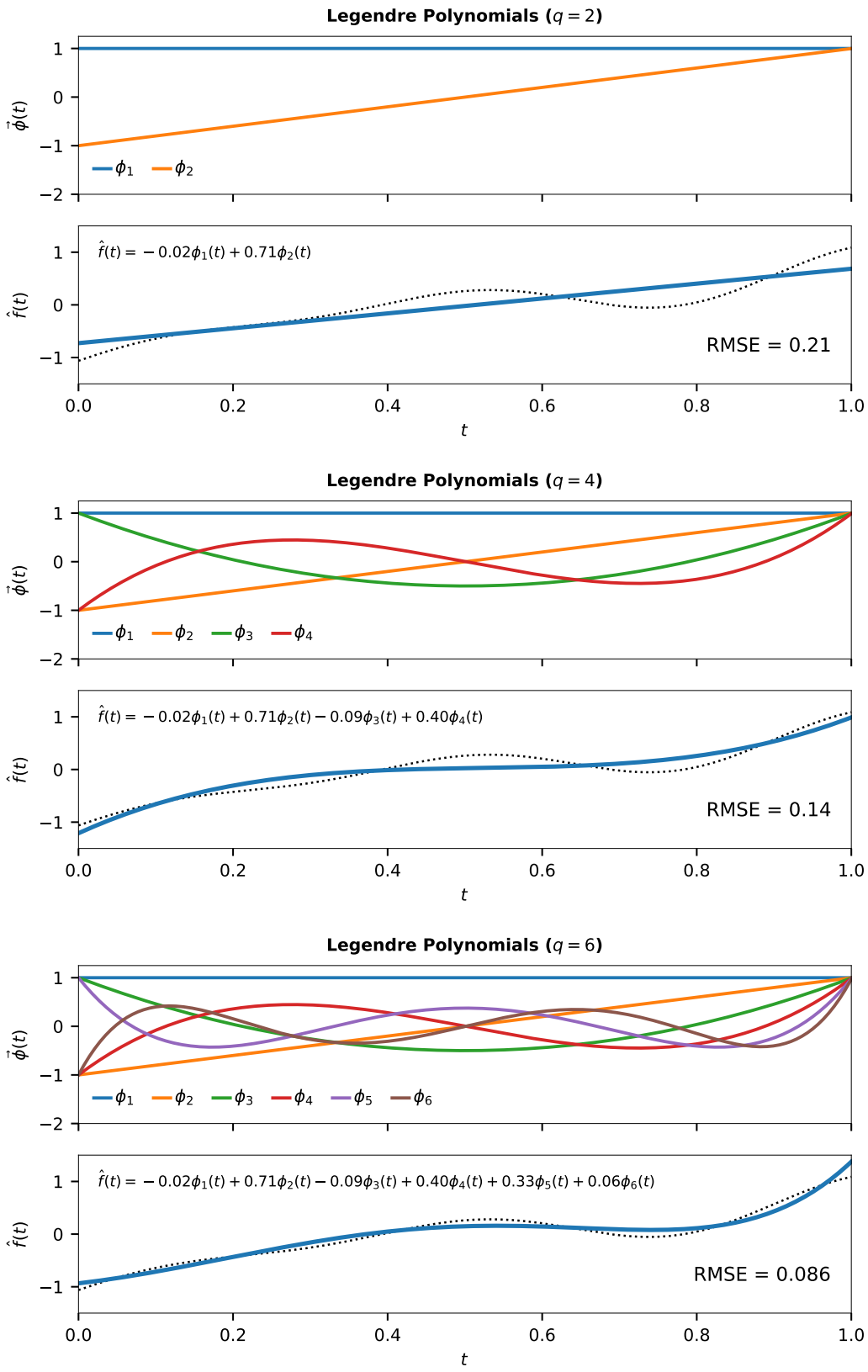


Figure 5: Representing a function (dotted line in the bottom plots) with a variable number q of shifted Legendre polynomials over the interval $\mathbb{X} = [0, 1]$. The error goes to zero as q increases.

One way of remembering a window of time $[t-\theta, t]$ is to implement a delay of θ seconds as a dynamical system. This may sound a little counter intuitive—how does implementing a *delay* help us to remember all the information within a time-window $[t-\theta, t]$? Actually, thinking about this a little more, when building a dynamical system that implements a delay, there is no way around remembering the entire time-window. There *has* to be information about all the time points in the interval $[t-\theta, t]$ in the system, because—as time t progresses—any point within that interval will eventually be located at $t-\theta$.

- **Motivation:** Implement a perfect delay of a time θ in the Neural Engineering Framework.
- This can be seen as a dynamical system. In the Laplace Domain, a perfect delay is $e^{-s\theta}$.
- Use Padé approximants up to a degree q to compute an LTI system approximating $e^{-s\theta}$. This system has a one-dimensional input $u(t)$ and an internal state $\mathbf{x}(t)$ of dimension q:

$$\mathbf{x}(t) = \mathbf{A}\mathbf{x}(t) + \mathbf{B}u(t) \tag{4}$$

$$\theta\mathbf{A} = a_{ij} \in \mathbb{R}^{q\times q}, \qquad a_{ij} = \begin{cases} (2i+1)(-1) & i < j, \\ (2i+1)(-1)^{i-j+1} & i \geq j, \end{cases} \tag{5}$$

$$\theta\mathbf{B} = b_i \in \mathbb{R}^q, \qquad b_i = (2i+1)(-1)^i. \tag{6}$$

- We can implement this LTI system as a neural ensemble using the transformation

$$\mathbf{A}' = \tau\mathbf{A} + \mathbf{I},$$
$$\mathbf{B}' = \tau\mathbf{B}.$$

- The state $\mathbf{x} \in \mathbb{R}^q$ represents more information than just the input θ seconds ago. It represents the state at every point in time up to θ seconds.
- In fact, $\mathbf{x}$ represents the function $f_{[t-\theta,t]}\left(\frac{\theta'}{\theta}\right) \approx u(t'-\theta')$ in the Legendre basis

$$u(t'-\theta') \approx f_{[t-\theta,t]}\left(\frac{\theta'}{\theta}\right) = \sum_{i=1}^{q} \tilde{P}_i\left(\frac{\theta'}{\theta}\right) x_i(t).$$

- When representing $\mathbf{x}$ in neurons, we can not only decode delays but any function using information from the past θ seconds.
- This network implements an optimal recurrent neural network remembering a slice of the past, as a so called "reservoir".

3.2 Example: Implementing the Delay Network in Nengo

The above may seem a little abstract—correspondingly, we use this section to describe how to actually implement the Delay Network in Python.

Step 1: Computing the matrices A, B. First, we need to compute **A** and **B** as defined in eq. (6). These matrices depend on two paramters: q, the number of state dimensions, and θ, the length of the time window. The following code has been adapted from Aaron Voelker's `nengolib`[1] library and computes these matrices in a few lines of Python.

```
def make_delay_network(q, theta):
    Q = np.arange(q, dtype=np.float64)
    R = (2 * Q + 1)[:, None] / theta
    j, i = np.meshgrid(Q, Q)
    A = np.where(i < j, -1, (-1.)**(i - j + 1)) * R
    B = (-1.)**Q[:, None] * R
    return A, B
```

Step 2: Computing the equivalent neural LTI system matrices. We can then turn these matrices into the feedback and input matrices **A**′ and **B**′ for a recurrently connected neural ensemble as discussed in the previous lecture. To this end, we use a generic function that takes τ, **A**, and **B** and outputs the adapted matrices **A**′ and **B**′.

```
def make_nef_lti(tau, A, B):
    Ap = tau * A + np.eye(A.shape[0])
    Bp = tau * B
    return Ap, Bp
```

Step 3: Computing a Delay Decoder. In order to decode a delay θ' from the function representation **x** according to eq. (1), we need to evaluate the Legendre polynomials at $\frac{\theta'}{\theta}$. Numpy implements Legendre polynomials in its `np.polynomial` module, so we do not have to worry about writing code for eq. (3) ourselves.

```
def make_delay_decoder(q, thetap, theta=1.0):
    ts = np.array(thetap / theta) # Compute the ratio between thetap and theta
    return np.array([np.atleast_1d(
        # Evaluate the Legendre polynomial of order "i" shifted to [0, 1] at "ts"
        np.polynomial.Legendre([0] * i + [1], [0, 1])(ts)) for i in range(q)])
```

Step 4: Building the network. Using Nengo, we use the system matrices **A**′ and **B**′ to implement the recurrent neural network.

```
q, theta, tau = 6, 0.5, 0.1
Ap, Bp = make_nef_lti(tau, *make_delay_network(q, theta)) #
dec_250 = make_delay_decoder(q, 0.25, theta) # 250ms delay
with nengo.Network() as model:
```

[1] `https://github.com/arvoelke/nengolib`

```
nd_in = nengo.Node(nengo.processes.WhiteSignal(
    high=2.0, period=10.0, rms=0.5))
nd_out = nengo.Node(size_in=1)
ens_x = nengo.Ensemble(
    n_neurons=500, dimensions=q,
    intercepts=nengo.dists.CosineSimilarity(q + 2))
nengo.Connection(nd_in, ens_x, transform=Bp, synapse=tau)
nengo.Connection(ens_x, ens_x, transform=Ap, synapse=tau)
nengo.Connection(ens_x, nd_out, transform=dec_250.T)
```

When executing this network, `ens_x` represents the q-dimensional state vector $\mathbf{x}$ that encodes the past θ seconds in the Legendre function space, i.e., $f_{[t-\theta,t]}$ in the notation used above. We could for example use the PES learning rule to learn a function that depends on $f_{[t-\theta,t]}$. In the above example, we just chose to compute a delay—the output node `nd_out` will contain the input delayed by 250 ms.

Note: *Using the cosine similarity distribution as intercept distribution.* You may have noticed the line `intercepts=nengo.dists.CosineSimilarity(q + 2)` in the above code. This selects the tuning curve intercepts according to the distribution of the cosine similarity between random vectors in $q + 2$ dimensions. Doing this is useful when building high-dimensional neural ensembles—without adapting the intercepts, many neurons end up not firing at all over large ranges of the represented space. With the adapted intercepts, the input current distribution stays approximately constant. See this Jupyter Notebook by Terrence C. Stewart and this tech report [4] for more details.

References

[1] Aaron R. Voelker and Chris Eliasmith. “Improving Spiking Dynamical Networks: Accurate Delays, Higher-Order Synapses, and Time Cells”. In: *Neural Computation* 30.3 (Mar. 2018), pp. 569–609. DOI: 10.1162/neco_a_01046. URL: https://www.mitpressjournals.org/doi/abs/10.1162/neco_a_01046.

[2] Aaron R. Voelker. “Dynamical Systems in Spiking Neuromorphic Hardware”. PhD thesis. Waterloo, ON: University of Waterloo, 2019. URL: http://hdl.handle.net/10012/14625.

[3] Aaron R. Voelker, Ivana Kaji, and Chris Eliasmith. “Legendre Memory Units: Continuous-Time Representation in Recurrent Neural Networks”. In: *Advances in Neural Information Processing Systems*. 2019.

[4] Aaron R. Voelker, Jan Gosmann, and Terrence C. Stewart. *Efficiently Sampling Vectors and Coordinates from the N-Sphere and n-Ball*. Waterloo, ON: Centre for Theoretical Neuroscience, Jan. 2017. DOI: 10.13140/RG.2.2.15829.01767/1. URL: https://www.researchgate.net/publication/312056739_Efficiently_sampling_vectors_and_coordinates_from_the_n-sphere_and_n-ball.