Josue Casco-Rodriguez
Prof. Xaq Pitkow
Neural Computation

Final Report

Abstract

Computational neuroscience has foundations in dynamical system theory, since systems of neurons exhibit dynamic behavior by virtue of their leaky properties and interactions with other neurons or the environment. Simulations of dynamic systems, specifically attractor networks, thus seem quite useful for understanding phenomena in computational neuroscience. However, in order for simulations to be useful for understanding phenomena, they often need to be run many times with different parameters. This project aims to provide a convenient, user-friendly way to simulate a wide range of attractor networks, not only for the sake of simulation, but also as a means to examine the performance of a simple Kalman Filter with respect to tracking attractor network responses.

Introduction

Neural computation cannot escape dynamical system theory. Neuroscience cannot exist without neurons, and neurons do not exist in a vacuum. Neurons, in order to be useful, must be connected with other cells or other neurons. When neurons are connected to other neurons, they form attractor networks – series of neurons that excite or inhibit each other. Each neuron in the network has a response intensity at any given time, often interpreted as an average firing rate. Each neuron's response intensity changes as a function of all the neurons' response intensity (including its own), applied stimuli, and noise, resulting in a system of differential equations that describe the neurons' responses as functions of time.

The simplest way to run an attractor network simulation while varying its parameters would be to change hard-coded constants or enter commands in a prompt. However, a more user-friendly and intuitive way to run simulations would be to have a graphical user interface (GUI) from which users can easily modify system parameters. The importance of having a GUI becomes especially apparent when considering how to assign weights in attractor networks. The eigenvalues related to the synapse weight matrix are very useful for characterizing the behavior of the network, but no user would be able to easily mentally calculate them, especially as the number of neurons grows. Having a GUI wherein the user can not only modify and observe previous synaptic weights with ease, but can also be given the resulting eigenvalues before running the simulation, is an excellent example of how a GUI can streamline network simulation.

In addition to network simulation, another facet of computational neuroscience worth exploring is neural signal processing. Specifically, methods and algorithms that scientists and engineers can use for better understanding recorded neural responses. One such method is the Kalman filter. Kalman filters are inference machines that track system states using system

models and measurements, even if the measurements are noisy. At each timestep, Kalman filters use the state transition model and the previous state estimate to predict what the state will be at the current timestep, then combine the prediction with the latest measurement to create the latest state estimate.

Outside the context of a simulation, Kalman filter design requires estimating characteristics of the system being observed, then tuning those estimates in an effort to attain better state estimation accuracy. Having a GUI to not only simulate a wide variety of networks, but also a variety of Kalman filter parameters, allows for simulation of Kalman filter performance in situations where the exact characteristics of the observed system are unknown.

The main challenges with creation of such a program are to create the functioning graphical user interface, understand how to represent a wide variety of attractor networks efficiently, and designing an appropriate and tunable Kalman filter for the attractor network. A previous study [1] examined a variety of attractor networks and how they reacted to stimuli, but did not explain how the networks it analyzed were found, nor did it explain what models the neurons in the network had. This project aims to make attractor-network generation as transparent and understandable as possible.

Methods

In the attractor network proposed, each of the three neurons emits a response (average firing rate) at time t. The values of the responses from the first, second, and third neuron are referred to as r_1, r_2, and r_3, respectively. The change in each response r_i over time is divided by a common timescale τ, where large τ incur slower network dynamics and small τ incur faster dynamics. The neurons are modeled as leaky, thus each $\dot{r}$ has a -r component. To be specific, each neuron will decay exponentially from its starting value if the neuron receives no input. Each neuron i feeds neuron j with its response r_i multiplied by a synaptic weight $W_{i \to j}$. The resulting collective linear system of equations is documented in Figure 1.

$$\tau \dot{r_1} = -r_1 + W_{1\to1} * r_1 + W_{2\to1} * r_2 + W_{3\to1} * r_3$$
$$\tau \dot{r_2} = -r_2 + W_{1\to2} * r_1 + W_{2\to2} * r_2 + W_{3\to2} * r_3$$
$$\tau \dot{r_3} = -r_3 + W_{1\to3} * r_1 + W_{2\to3} * r_2 + W_{3\to3} * r_3$$

Figure 1: Basic Equations of Linear Dynamic System of Three Neurons

Representing the responses r_1, r_2, and r_3 as a vector $\vec{r}$, the resulting system simplifies to Figure 2, where W is the synaptic weight matrix and I is the identity matrix.

$$\tau \dot{\vec{r}} = -\vec{r} + W\vec{r}$$
$$\tau \dot{\vec{r}} = (-I + W)\vec{r}$$

Figure 2: Simplified Representation of Linear Dynamic System

However, attractor networks neurons are often nonlinear and are subject to noise and external stimuli. Representing the neurons' activation function as f(x), the noise applied to each neuron as $\vec{\eta}$, and the stimuli applied to each neuron as $\vec{b}$, the resulting general-case dynamics are represented in Figure 3.

$$\tau\dot{\vec{r}} = -\vec{r} + f(W\vec{r} + \vec{\eta} + \vec{b})$$

Figure 3: General-case Dynamic System

The simulation program was made in Python using FilterPy, PyQt, and standard Python libraries. The graphical user interface (GUI), shown in Figure 4, has various user-definable parameters that fall into a few main categories.

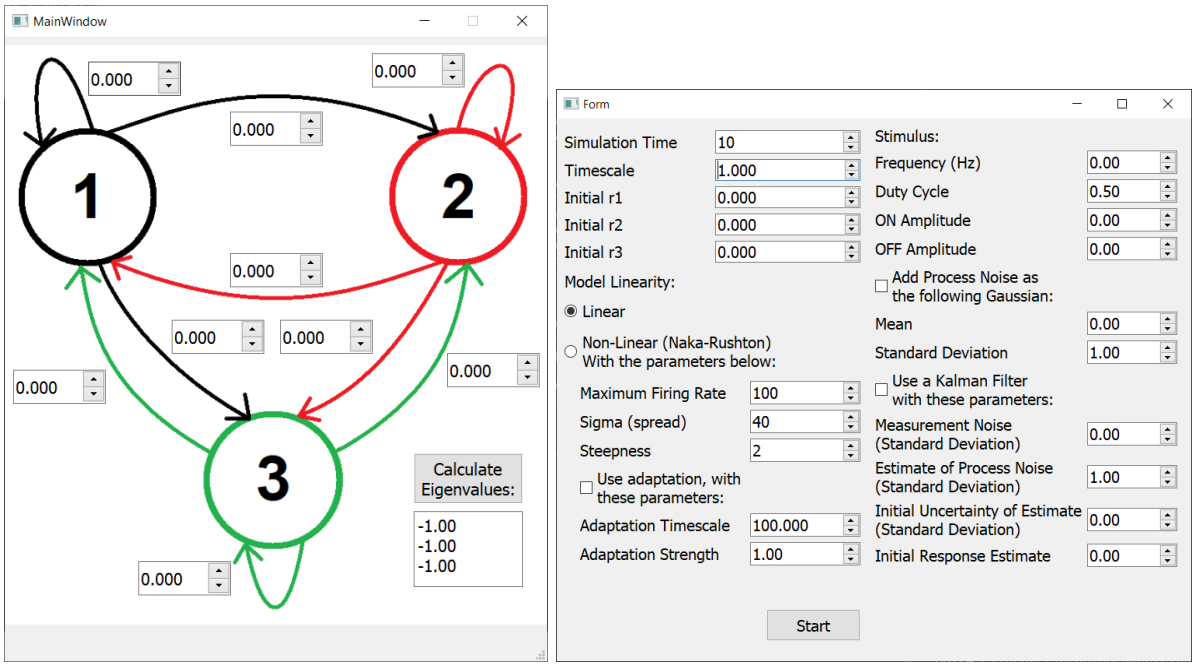

Figure 4: Graphical User Interface

The first category has its own window and consists of synapse weights that can be assigned to each connection from one neuron to another. A nearby button allows the user to see the eigenvalues that result from the matrix -I + W.

The second category consists of common parameters that every simulation needs: 1) The total simulation time in seconds. Simulation of the system of ordinary differential equations is done using Euler's method with a timestep of 0.01 seconds. 2) The timescale τ. 3) The initial response values at time t = 0.

The third category pertains to the attractor network linearity. The user can choose between a linear and nonlinear network, with the only difference being the nature of the

activation function f(x). In the linear case, f(x) = x. In the nonlinear case, f(x) is described by the Naka-Rushton function, shown in Figure 5 [3].

$$f(x) = \begin{cases} x_{max} * x^S/(\sigma^S + x^S) & x \geq 0 \\ 0 & x < 0 \end{cases}$$

Figure 5: Naka-Rushton Function

As an activation function, the Naka-Rushton is quite similar to a sigmoidal or hyperbolic tangent function, except that the function converges to 0 for any negative input. The Naka-Rushton function is characterized by three parameters: σ, the semi-saturation constant, x_{max}, the maximum value of f(x), and S, the steepness of the function. Higher σ values correspond to higher x values required to reach x_{max}, and higher S values correspond to sharper transitions between 0 and x_{max}. One optional parameter of the Naka-Rushton is an adaptation component [4]. The adaptation component, A, interacts with the Naka-Rushton function and changes as a function of itself and the response, r, as shown in Figure 6. The strength of adaptation, k_A, and the adaptation timescale, τ_A, are constants.

$$f(x) = \begin{cases} x_{max} * x^S/((\sigma + A)^S + x^S) & x \geq 0 \\ 0 & x < 0 \end{cases}$$

$$\tau_A \dot{A} = -A + k_A r$$

Figure 6: Naka Rushton with Adaptation

When the user chooses the nonlinear case, they must specify the Naka-Rushton parameters x_{max} (maximum firing rate), σ (semi-saturation constant), and S (steepness). In the nonlinear case, the user can also choose to add adaptation to the network, with τ_A (adaptation timescale) and k_A being user-defined parameters.

The fourth category of parameters characterize the stimulus, $\vec{b}$,that is applied to all neurons in the network. The stimulus varies with time and is the same for all neurons, meaning that $\vec{b}$can be described as in Figure 7, where b(t) is a user-definable pulse wave. The frequency, duty cycle, and upper and lower amplitudes are the four user-definable parameters that can characterize any pulse wave.

$$\vec{b}(t) = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} b(t)$$

Figure 7: Simplification of Stimulus

The fifth category of parameters pertain to the process noise, $\vec{\eta}$.Like the stimulus ($\vec{b}$), the noise is a function of time. However, unlike the stimulus, each neuron receives a different noise value at each time t. Each noise value is independent of each other. The noise values are independently identically distributed Gaussian random variables, with a user-definable mean and standard deviation, as shown in Figure 8.

$$\vec{\eta}(t) = \begin{pmatrix} \eta_1(t) \\ \eta_2(t) \\ \eta_3(t) \end{pmatrix}$$

$$\eta_1(t), \eta_2(t), \eta_3(t) \overset{iid}{\sim} N(\mu, \sigma^2)$$

Figure 8: Characterization of Process Noise

The sixth category of parameters characterizes the Kalman Filter that the user may choose to run on one of the responses, $r_1(t)$, of the network. Certain aspects of the Kalman Filter are user-controllable, while others are not. Kalman filters are characterized by several matrices: 1) x, the vector of estimated states. 2) P, the uncertainty covariance matrix between the states. 3) F, the state transition matrix. 3) B, the control-input model matrix. 4) u, the control vector. 5) Q, the process noise covariance. 6) H, the measurement model matrix. 7) R, the measurement noise covariance. 8) z, the measurement observed each timestep. The remaining matrices in the Kalman filter algorithm (the residual covariance S, the Kalman gain K, and the residual y) are functions of the aforementioned matrices. At each timestep, the Kalman filter generates predictions $\bar{x}$and $\bar{P}$about what the future values of x and P will be, given their previous values and the latest control u. Then, the Kalman filter updates its prediction by taking into account the latest observation z and how it relates to the predictions $\bar{x}$and $\bar{P}$,resulting in new estimates of x and P. Figure 9 summarizes and further illustrates the Kalman filter algorithm and how it uses the aforementioned matrices to generate state estimates.

Predict Step

$$\bar{\mathbf{x}} = \mathbf{F}\mathbf{x} + \mathbf{B}\mathbf{u}$$
$$\bar{\mathbf{P}} = \mathbf{F}\mathbf{P}\mathbf{F}^{\mathsf{T}} + \mathbf{Q}$$

Update Step

$$\mathbf{S} = \mathbf{H}\bar{\mathbf{P}}\mathbf{H}^{\mathsf{T}} + \mathbf{R}$$
$$\mathbf{K} = \bar{\mathbf{P}}\mathbf{H}^{\mathsf{T}}\mathbf{S}^{-1}$$
$$\mathbf{y} = \mathbf{z} - \mathbf{H}\bar{\mathbf{x}}$$
$$\mathbf{x} = \bar{\mathbf{x}} + \mathbf{K}\mathbf{y}$$
$$\mathbf{P} = (\mathbf{I} - \mathbf{K}\mathbf{H})\bar{\mathbf{P}}$$

Figure 9: Kalman Filter Algorithm

The designed Kalman filter tracks two states, r and $\dot{r}$.For simplicity, the Kalman filter assumes that r changes as a linear function of $\dot{r}$,does not know how$\dot{r}$ changes, and does not know anything about the control or stimuli being applied to the network. The two states (r and $\dot{r}$) are initially assumed to be uncorrelated and have the same initial uncertainty, $\sigma^2_{initial}$. The process noise covariance matrix Q is a function of the estimated noise variance, $\sigma^2_{process\text{-}estiamte}$. With x, F, B, u, P, F, and Q, the Kalman Filter can make predictions about the values of states, given the previous estimates of what those states were. Figure 10 contains exact descriptions of the aforementioned matrices involved in the prediction stage of the Kalman filter.

$$x = \begin{pmatrix} r \\ \dot{r} \end{pmatrix}$$

$$F = \begin{pmatrix} 1 & dt \\ 0 & 1 \end{pmatrix}, dt = 0.01$$

$$B = \begin{pmatrix} 0 & 0 \end{pmatrix}$$

$$u = \vec{b}(t)$$

$$P_{initial} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} * \sigma^2_{initial}$$

$$Q = E(\vec{n} * \vec{n}^T)$$

$$\vec{n} = \begin{pmatrix} n_1 \\ n_2 \end{pmatrix}, n_1, n_2 \overset{iid}{\sim} N(0, \sigma^2_{process-estimate})$$

Figure 10: Matrices Used in Kalman Filter Prediction Stage

As for the update stage, only two extra matrices are required. The designed Kalman filter only observes a noisy form of r and cannot directly observe $\dot{r}$,only estimate it. Thus, the measurement z is simply a scalar, and the measurement function H must also return a scalar when multiplied by the state vector x. The measurement noise covariance matrix R simply becomes a scalar, $\sigma^2_{measurement}$. Figure 11 summarizes the additional matrices the Kalman filter uses to adjust its predictions to account for measurements.

$$z = r + N(0, \sigma^2_{measurement})$$

$$H = \begin{pmatrix} 1 & 0 \end{pmatrix}$$

$$R = \sigma^2_{measurement}$$

Figure 11: Matrices Used in Kalman Filter Update Stage

The GUI user must define the measurement noise $\sigma^2_{measurement}$, the estimate of process noise $\sigma^2_{process\text{-}estimate}$, initial estimate of r_1 at t = 0, and the initial uncertainty of the state estimates

$\sigma^2_{initial}$. The designed Kalman Filter assumes that the measurement noise distribution is known, while the process noise distribution is estimated (the user can opt to give the true and estimated process noises the same variance if they so desire).

Results

Before applying a Kalman filter to one of the responses, the effects of the non-Kalman parameters on the system were explored. Nonlinearities allow for systems like winner-take-all networks to arise. Meanwhile, adaptation did not seem to have a profound effect on the systems, except in extreme cases. The lack of effect is likely because τ_A (adaptation timescale) is usually much bigger than τ (network timescale), meaning that any effects that adaptation would have on the system would take effect very gradually. Nonetheless, an extreme example (high adaptation strength k_A and low adaptation timescale τ_A) is shown in Figure 12, wherein a nonlinear network that had formerly converged to a value begins to steer away from the value once very strong adaptation is introduced.

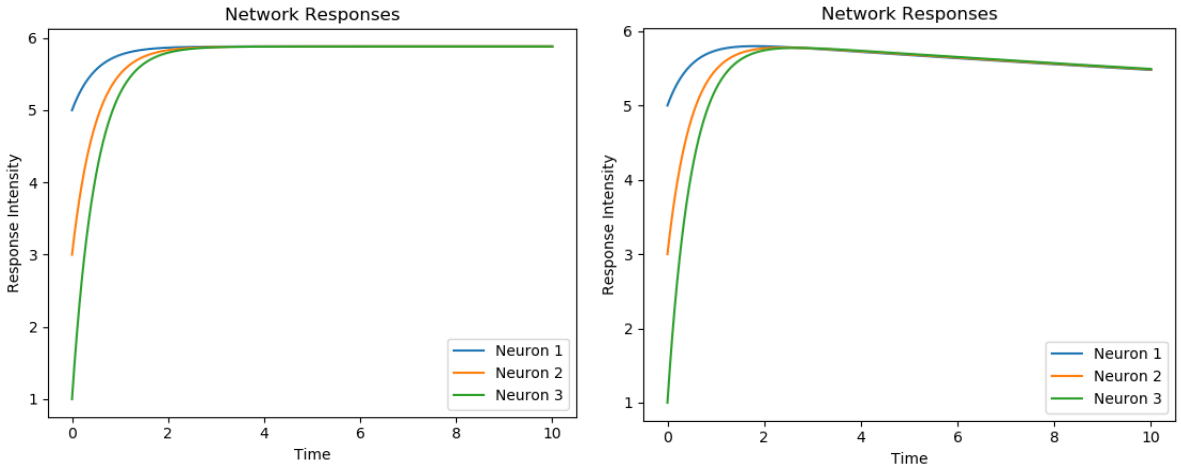

Figure 12: Stable Nonlinear Network Without (Left) and With (Right) Strong Adaptation

The effect of stimuli was most notable in building nonlinear networks. Specifically, strong stimuli were of utmost importance for building networks wherein the stable fixed points of the network involved all or just one of the neurons firing at a high response value. As for process noise, its effects depended on its characteristics. The noise variance σ^2 corresponded to how noisy the response trajectories were, while the noise mean μ essentially acted like a constant stimulus of intensity μ. One outcome of interest involving process noise is that, in extreme cases, it could make nonlinear networks behave differently than expected. For example, in Figure 13, a winner-take-all network is simulated wherein Neuron 1 should dominate the system every time, but if heavy noise is added, then sometimes other neurons will win by chance.

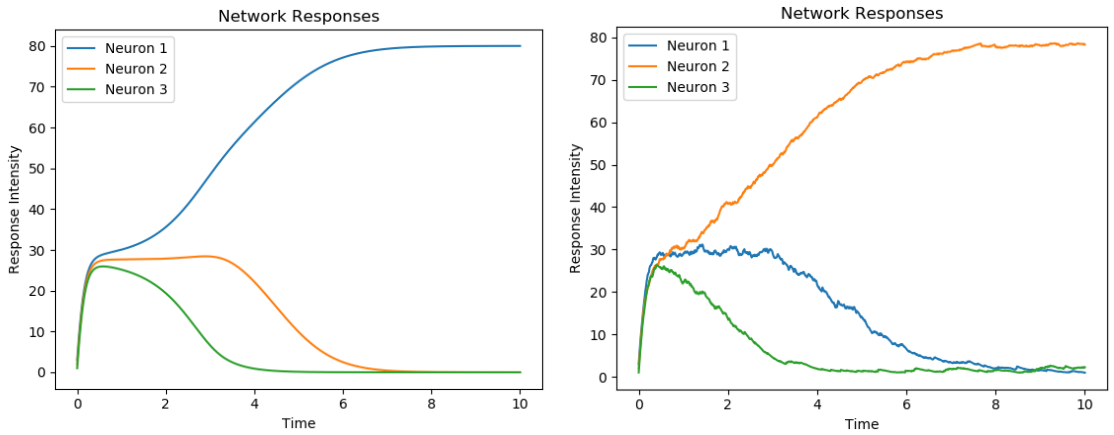

Figure 13: Winner-Take-All Network Without (Left) and With (Right) Strong Process Noise

The performance of the Kalman filter varied depending on the nature of the system being tracked. The designed Kalman filter intentionally has little knowledge of the system dynamics and only tracks one of the responses instead of all three. The following figures contain two plots each. Each plot has three lines: the true response value, the noisy measurements of the response (referred to as sensor measurements), and the estimated response value. Both plots have the same lines and the same data, except that one plot has the noise illustrated as a line (to illustrate how noisy the measurements are), while the other has the noise illustrated as small pixel points (to highlight the differences between the true response values and the estimated response values).

A. Insert a bunch of graphs. Make sure they are well-labeled and explained.

One of the simplest networks to observe is a linear network whose responses simply start at nonzero values and then exponentially decay to zero. In this case, as shown in Figure 14, the Kalman filter successfully tracks the overall shape of the response, but does have an overshoot when the response starts to settle at zero. With time, the Kalman filter updates itself to the correct value of zero.

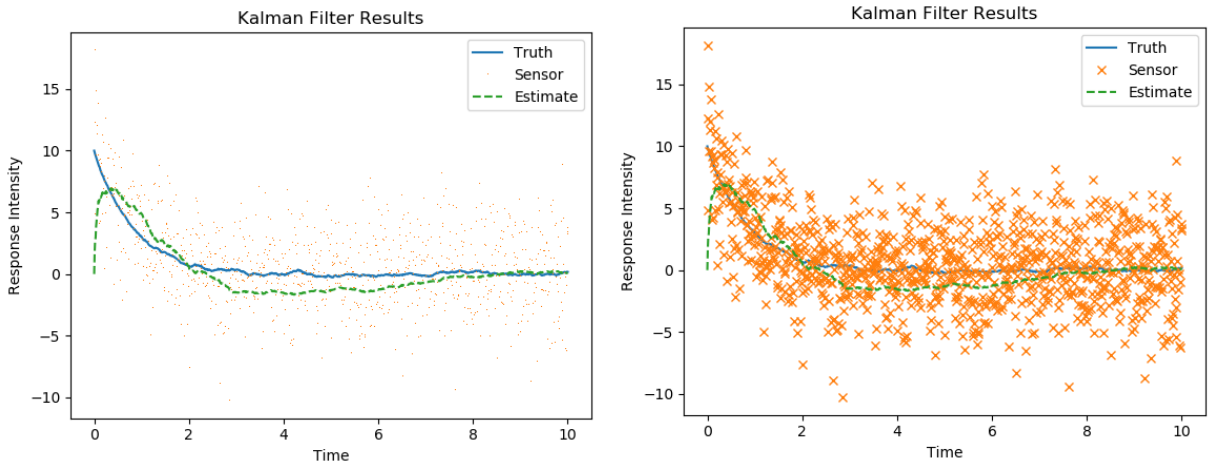

Figure 14: Kalman Filter Tracking Linear Exponentially Decaying Response

A more interesting linear attractor network is an oscillator. The designed Kalman filter struggles much more with oscillatory responses, since the designed filter's assumption that the change in response is constant does not match the reality of the system. In an oscillatory network, the derivative of the responses are themselves sinusoidal and constantly changing. At best, the designed filter will track the response with a delayed phase shift, as shown in Figure 15. In the worst case, which happens as the frequency of the response increases, the designed filter is unable to keep track of the rapid changes, and thus outputs as its estimate a delayed and strongly attenuated version of the true response.

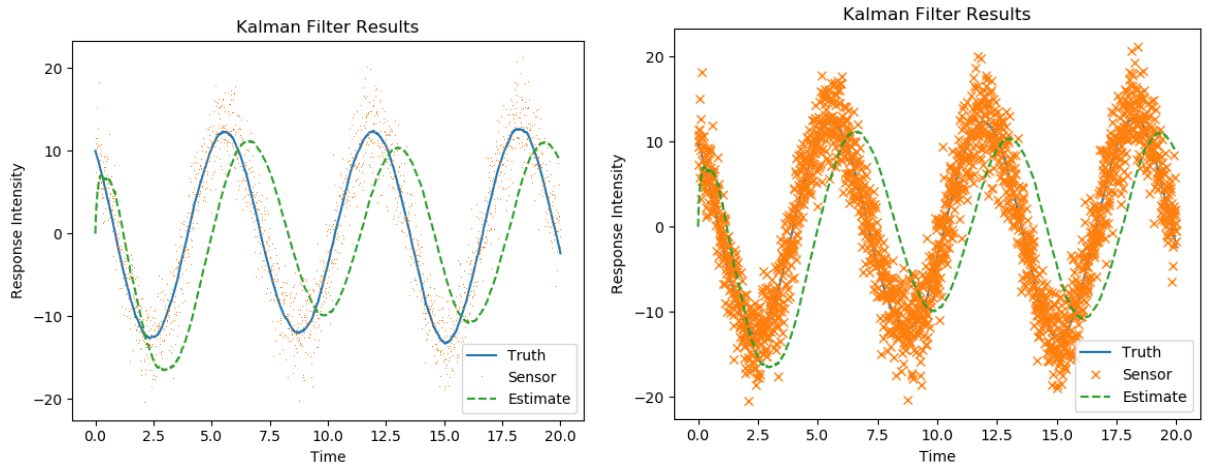

Figure 15: Kalman Filter Tracking Stable Oscillatory Response

In unstable systems, the Kalman filter can successfully track exponential growth, but again struggles with oscillatory values. In Figure 16, the Kalman filter does not fail to increase the magnitude of its response estimate as the measurements increase in magnitude. However, the phase shift in the Kalman filter results in a delay in the exponential growth of the filter's response estimate; the magnitude of the estimate lags behind the magnitude of the response by slightly more than one period.

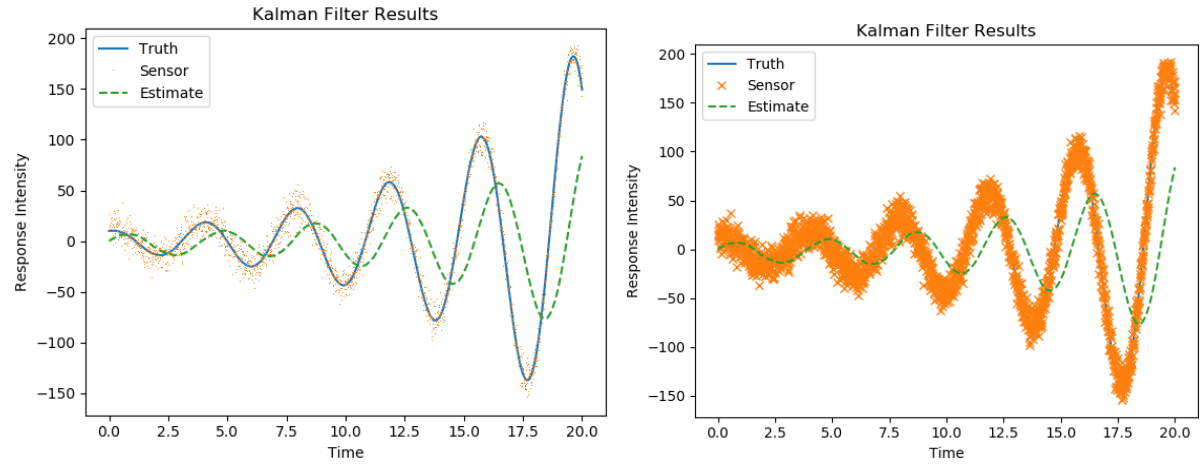

Figure 16: Kalman Filter Tracking Unstable Oscillatory Response

As for nonlinear systems, the Kalman filter again mainly struggles with sharp changes in value. In Figure 17, the designed Kalman filter is tracking a simple nonlinear response that would normally converge to a value between 40 and 50, but strong adaptation leads the response to decay after briefly reaching said value. The designed filter ultimately reaches the correct value of the response, but the estimate has a strong overshoot component when the response changes trajectories.

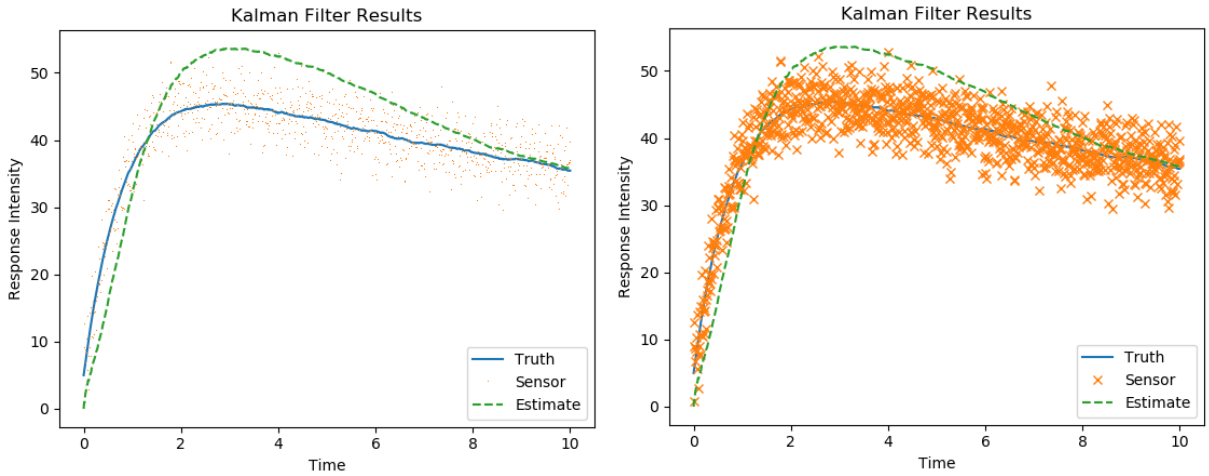

Figure 17: Kalman Filter Tracking Nonlinear Response With Adaptation

Another nonlinear system of interest is a winner-take-all network. In Figure 18, the designed filter is tracking the winner of a winner-take-all network. As in Figure 17, the designed filter suffers from an overshoot component, but throughout the rest of the execution time, the state estimate follows the true response quite closely.

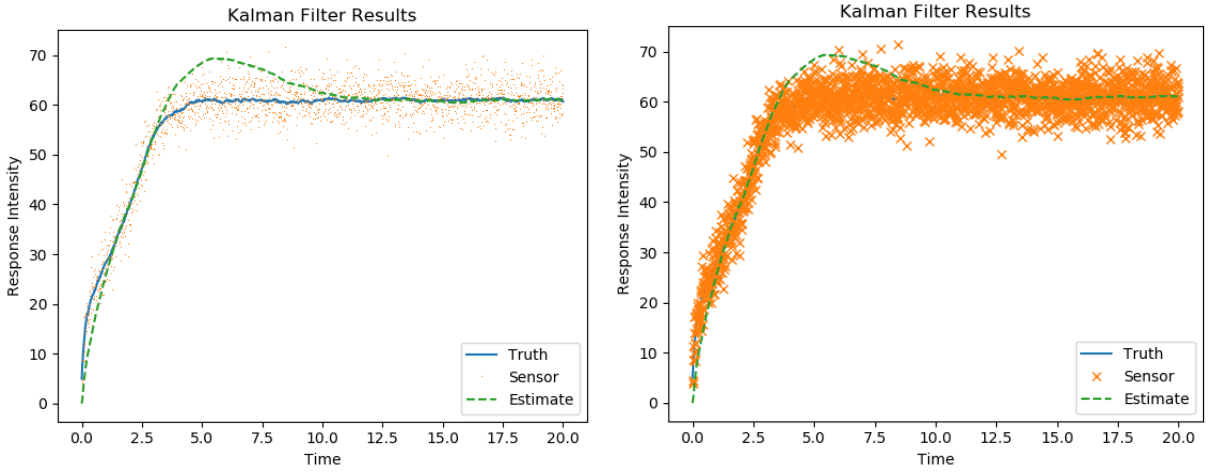

Figure 18: Kalman Filter Tracking Winner of Winner-Take-All Network

In the case where an oscillatory stimulus is applied to a network the Kalman filter faces the same challenges as when it tracks an oscillatory network without stimulation. In Figure 19, a simple, exponentially-decaying network has a pulse wave applied to it, resulting in responses

that resemble lowpass-filtered square waves. The designed filter's estimate resembles a triangle wave. As with previous oscillatory networks, the best-case performance for the designed filter resembles the true response delayed by a phase shift, while the worst-case performance adds attenuation.

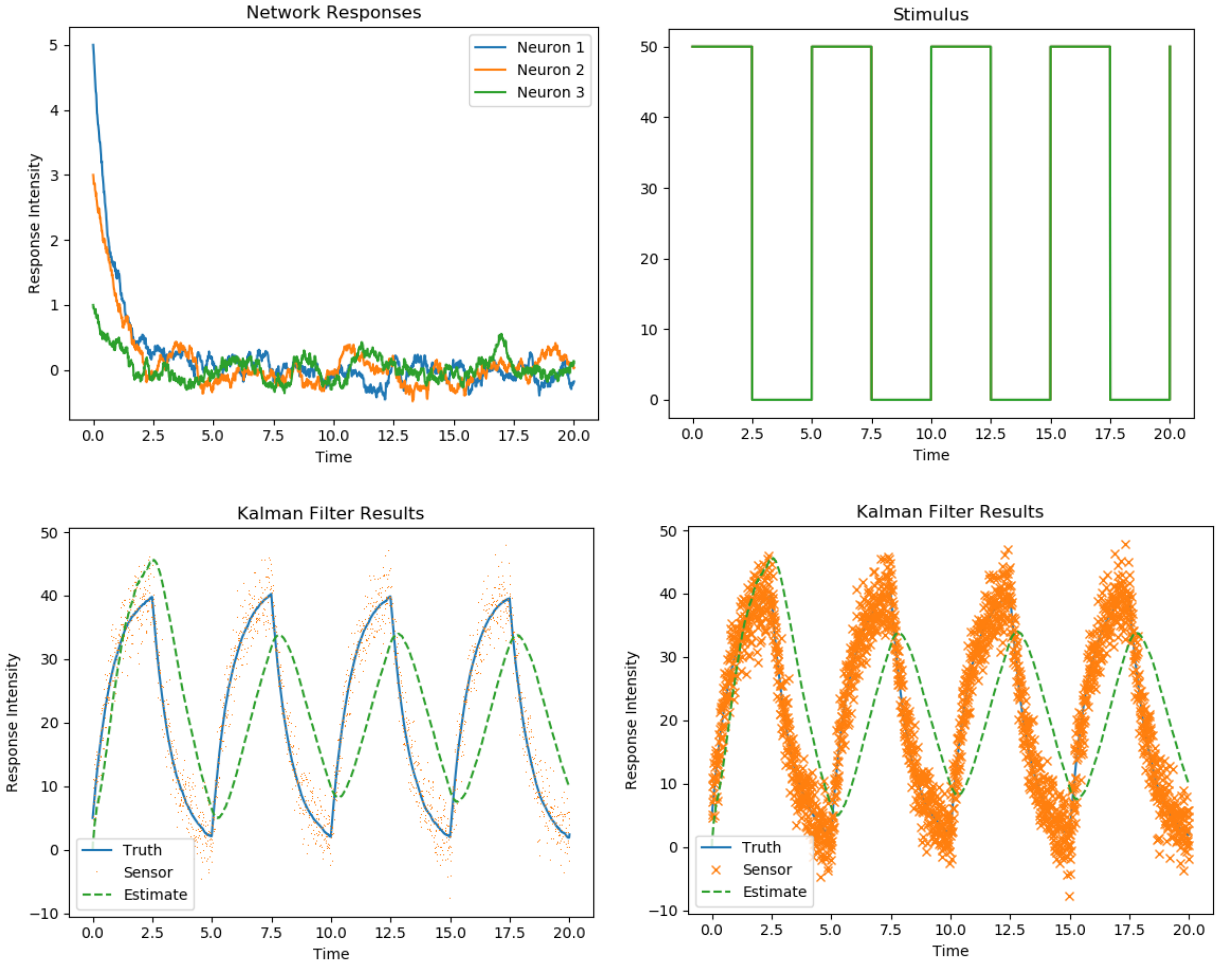

Figure 19: Kalman Filter Tracking Linear System With Stimulus. The Network Responses plot (top-left) depicts that, if no stimulus is applied, the network responses will exponentially decay to 0. The Stimulus plot (top-right) shows the stimulus to be applied to the network. The Kalman Filter Results plots (bottom-left and bottom-right) show what the designed filter's estimate of the response was, when the stimulus was applied to the network.

Conclusion

Producing the GUI was an excellent and rigorous exercise in Python GUI creation. Testing the effects of neuron weights and model linearity reinforced knowledge of how to create systems like oscillators or winner-take-all networks. Adding parameters allowed for exploration and understanding of the effects of adaptation and process noise on attractor networks.

The Kalman filter, despite its intentional lack of information about the system, does surprisingly well at tracking a response from the attractor network. Since the designed Kalman filter only tracks the value and derivative of the response, the filter does well with constantly changing values, whether they stem from linear or nonlinear dynamics, with or without stimulus. However, as a result, the Kalman filter struggles with unstable, oscillatory , or sharply changing values, since they involve response derivatives that are not constant.

Designing the Kalman filter was a rigorous introduction to state estimation methods, and seeing the limitations of the designed filter resulted in a better understanding of how important the assumptions made about a system and its noise are in state estimation. Adding more states or designing the filter with more information about the system dynamics likely would have made the filter more able to track the responses.

References

---

[1] P. Miller, "Dynamical systems, attractors, and neural circuits." *F1000Research*, vol. 5, May, 2016. [Online serial]. Available: https://f1000research.com/articles/5-992/v1. [Accessed Apr. 1, 2021].

[2] R. R. Labbe Jr., "Kalman and Bayesian Filters in Python," October, 2020. [Online]. Available: https://nbviewer.jupyter.org/github/rlabbe/Kalman-and-Bayesian-Filters-in-Python/blob/master/table_of_contents.ipynb. [Accessed Apr. 1, 2021].

[3] H. R. Wilson, *Spikes, Decisions, and Actions: Dynamical Foundations of Neuroscience*. New York: Oxford University Press, 1999, pp. 19-22.

[4] H. R. Wilson, *Spikes, Decisions, and Actions: Dynamical Foundations of Neuroscience*. New York: Oxford University Press, 1999, pp. 81-82.

The GUI is on GitHub: https://github.com/Josuelmet/Neuron-GUI-2021