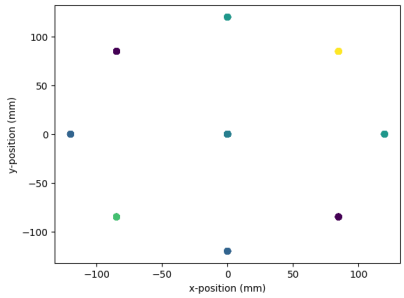
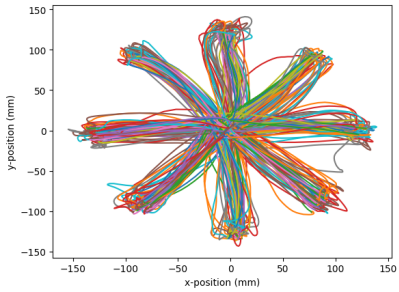
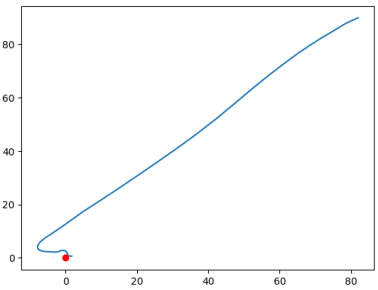
ECE 189 Final Report: Decoding intended motion from Neural Signals

Background:

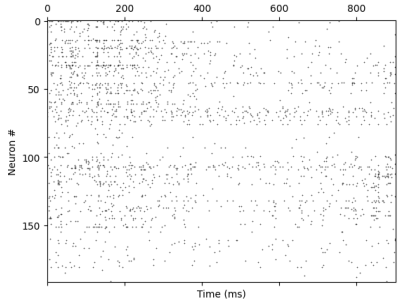
We are given a dataset with Neural data from 192 Electrodes detecting spikes sampled every 1ms. For each sample, we measure the presence or absence of a spike. There cannot be more than one spike per ms. We use this data to estimate the intention of a monkey to make a motion. We also have the “labels” in the form of the monkey's cursor positions(in 2D) sampled at 1ms.
To visualize the targets and the cursor positions, we plot(1) and (2).

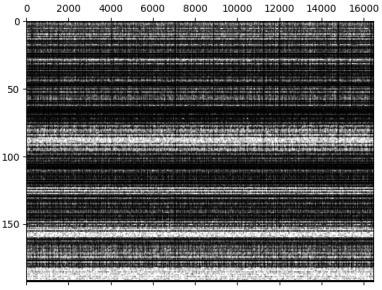


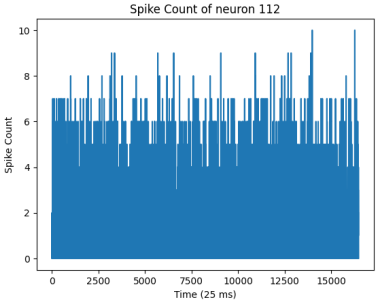




To make even more sense of this, we can plot the cursor positions and target for one trial(3), and you can see the monkey reaching for the target. We will be using cursor positions as our X, which we are trying to predict with Y.
Now to visualize the neural data: we can Spy plot the neurons. On (1) we can see the spy plot. Where we see the black dots, we know that a neuron fired in that 1ms duration for a specific neuron. For (2) we have binned the neuron data, such that every index is the number of spikes in a 25ms interval. On (3) we plot the spike count for every 25 ms for the neuron with the highest variance.



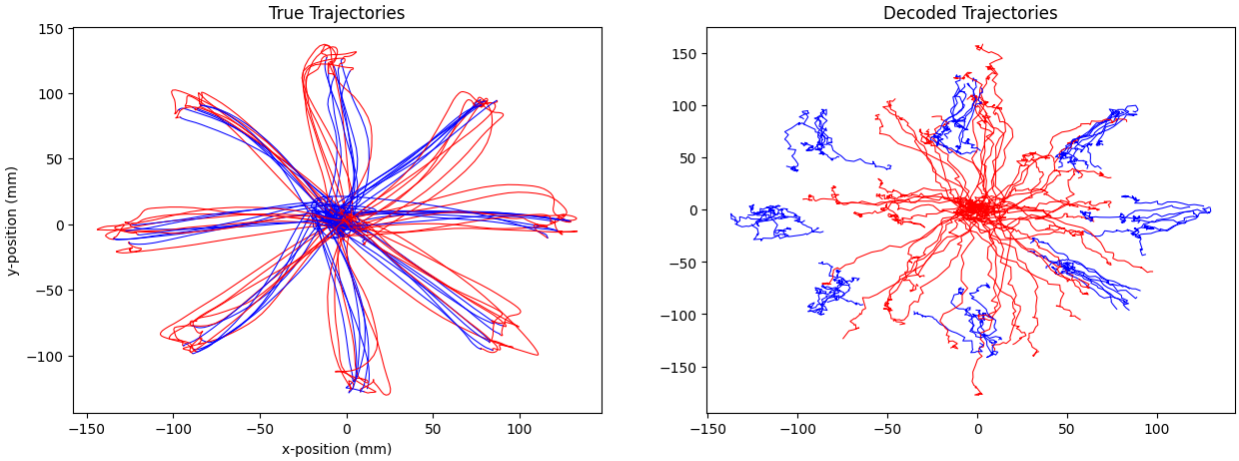




Building a decoder

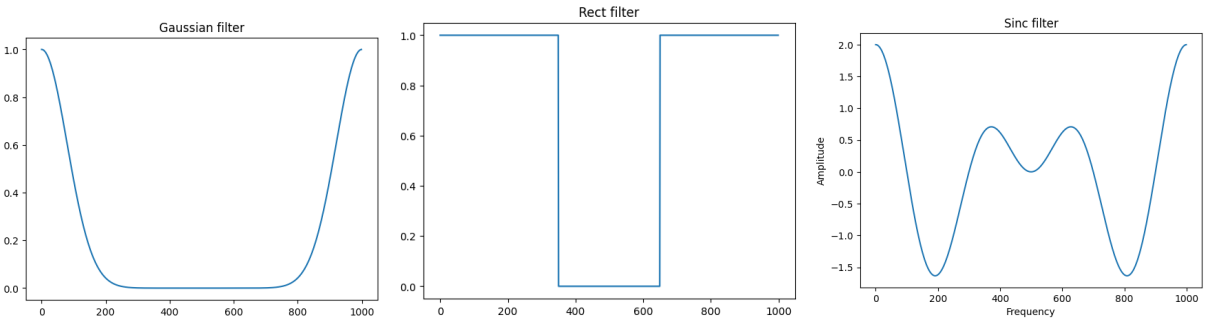
To build a decoder, we can use a simple linear model to project the trajectories. The linear model is given as **X = LY**, where **X** is the cursor velocities (vx, vy), **L** is the model, and **Y** is the matrix with the Binned Spikes for each electrode. When applying this

simple model, we can decode the intended motion with an **MSE** of **3492**. To visualize these trajectories, we can plot the true and projected trajectories and compare them.

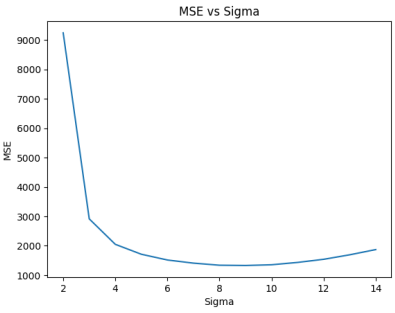


Feature #1: Low Pass Filtering

I attempted 3 different types of Low Pass Filters, a Gaussian(1), A Rect(2), and Sinc(3).

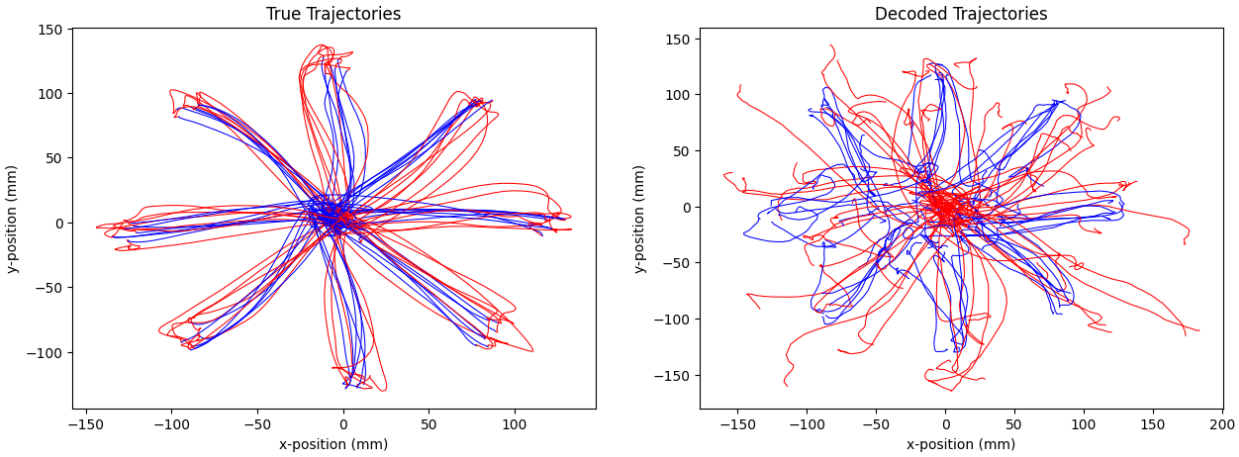


I found most success with the Sinc Filter, which I implemented as: sinc_filter(size, things, o=2) = fft(i2), where i2 = np.zeros(size) followed by i2[0 + o:things + 0] = 1. Naming this as a sinc filter is a misnomer. I looped through various values of things and found that things=9 and o=2, worked best for the filter.



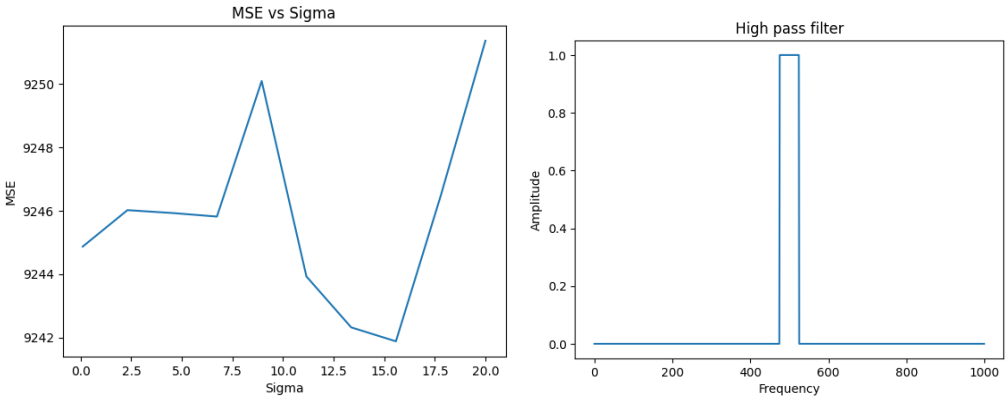
Since I was looping to find the best parameters, I chose to make these filters in the frequency domain so that the filtering operation would be multiplication and an ifft rather than convolution which increased the speed of my code. The size flag represents the size of the filter, the things flag represents the number of ‘taps’(1s) in the filter, and the ‘o’ flag represents the number of 0s the filter was padded with before convolution. The plot on the right shows the MSE vs ‘things’, and when taking the argmin, the min MSE is at things = 9, which produces an MSE of 1330. While plotting the true and decoded trajectories, we can visually see the

increase in performance from before a low pass filter.

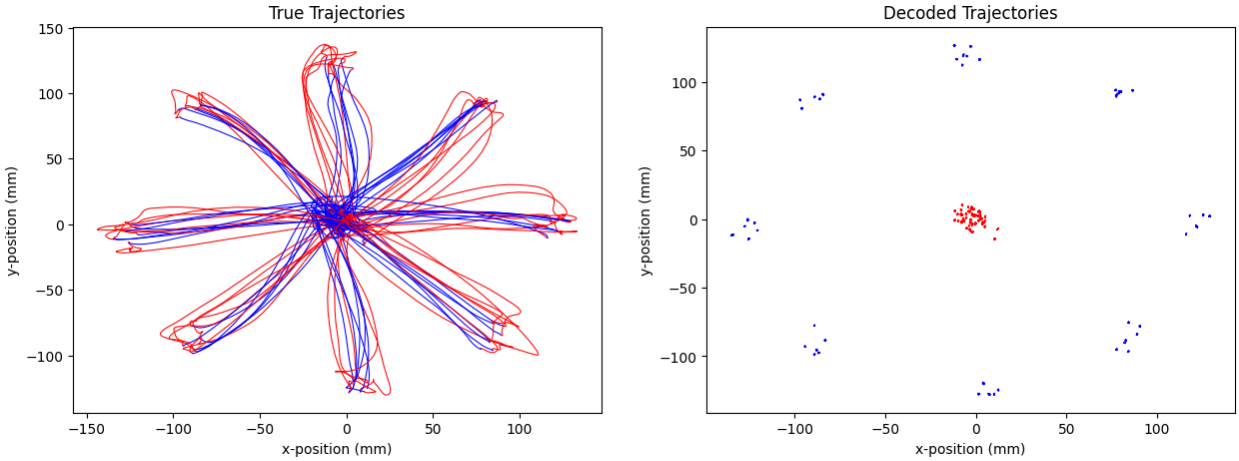


Feature 2: High Pass Filtering

Using a simple Rect High Pass Filter, we notice a large performance drop. I tried various widths for the rect of the High Pass filter and found that the performance was poor across all values.



As you can see, the MSE was more than 9000 for most of the high pass. This is because the High Pass Filter picked up the high-frequency noise components of the signal.



Feature 3: Derivative Filter

Using a derivative filter, we still notice a high-performance drop. The reason this is, is because a derivative acts like a high pass filter, such that the frequency response has a higher amplitude for high frequencies than lower frequencies. The MSE on the derivative filter was = 9208.