Exploratory Analysis of Large-Scale Electrode Neural Recordings

Jiaqi Shang, Maple Tan
Department of Applied Mathematics, University of Washington, Seattle, Washington

Abstract

Many cognitive functions depend on the interactions of neurons at the population level. Recent improvements in neural recording technologies such as Neuropixels (Jun et al., 2017) have rapidly increased the number of neurons that can be simultaneously recorded from, and hence, opens questions of what scientific insights can be gained by studying the neural activities at the population level. Here, we analyze electrode recorded neural activities from 398 neurons across 7 brain regions while the animal (mouse) is presented with natural movie clips. In particular, we study whether there is area-specific activity patterns in the population, and also search for underlying low-dimensional dynamics.

Section I: Introduction and Overview

In order to study how the brain functions, it is critical to measure the joint activities of neurons across brain regions simultaneously. This has only been possible with recent technological advancements, one of which is the Neuropixel. Neuropixles are electrodes that can record the activity of hundreds of neurons across the brain during behaviour (Jun et al., 2017). This paper studies the neural data collected from a mouse during passive viewing of 81 second natural movie clips. A total of 398 neurons from 7 different brain regions are recorded as the movie clips are repeated 97 times. Spikes are extracted from raw data and the number of spikes are recorded at 30 Hz. Hence, for each repeat of the 81 second movie, there are 2430 data collection time points and the dataset is a 3D matrix with dimensions of 398 x 97 x 2430. The location of the each identified neuron is also recorded.

First, we study whether there are area-related characteristics in the population activity. To do this, we first apply four supervised classification methods (K-Nearest Neighbors, Support Vector Machine (SVM), Naive Bayes Classification, and Decision Tree) to the data and test their performance on classifying the neural activity by brain regions. We find that all 4 methods tested can correctly predict the area from neural activity to some extent (average accuracy across methods: 38.52%), with which methods having the highest accuracy (61.98%). Next, we apply unsupervised classification methods (K-means and Gaussian mixture model (GMM)) to the data to see if unsupervised clustering can reconstruct the identified brain regions. We find that all methods tested have some identified clusters that correspond to certain areas, suggesting that the neural activity contains area-specific information that can be extracted by unsupervised classification methods. Then, we used Principal Component Analysis (PCA) to explore whether there is low-dimensional dynamics under the high-dimensional neural activity. We also used the Dynamical Mode Decomposition (DMD) method to separate the slow and fast dynamics in the data.

In the following, Section II will introduce the theoretical methods used in the paper, including supervised and unsupervised classification methods, as well as PCA and DMD. Section III will explain the specific algorithm used in the analysis and computational results will be shown in Section IV.

Section II: Theoretical Background

2.1 Supervised Classification Methods

Supervised classification methods are algorithms that map inputted data to associated labels based on example pairs.

2.1.1 K-Nearest Neighbors

K-nearest neighbors creates clusters from inputed data, known as 'training data'. Given a new data point, the algorithm finds the closest cluster to that the data point and assigns it to that cluster.

2.1.2 Naive Bayes

Naive Bayes method is a supervised learning algorithm that uses Bayes' theorem with the 'naive' assumption of conditional independence between every pair of features. Given a set of data with given target variable y and set of features $x_1, x_2, ..., x_n$, we assume that:

$$P(y \mid x_1, \ldots x_n) = \frac{P(y)P(x_1, ..., x_n \mid y)}{P(x_1, \ldots, x_n)} \quad (1)$$

Using the naive conditional independence assumption that

$$P(x_i \mid y, x_1, ..., x_{i-1}, x_{i+1}, \ldots, x_n) = P(x_i \mid y) \quad (2)$$

For all x_i , and so this relationship will simplify to

$$P(y \mid x_1, \ldots x_n) = \frac{P(y)\prod_{i=1}^{n} P(x_i \mid y)}{P(x_1, \ldots, x_n)} \quad (3)$$

The naives bayes model then combines this with a decision rule and will pick the category the new inputed data that has the highest probability this new inputed data belongs to.

2.1.3 SVC

Linear SVM constructs a hyperplane that would potentially splits the data after projecting it into a higher dimension. To find the best hyperplane to split the data, the algorithm solves an optimization problem where it must choose a decision line that would minimize the amount of labelling errors for the data and also maximizes the separation between different labeled clusters. Once these hyperplanes are created, we can determine the classification of a data point by observing which side of the hyperplane it falls on.

2.1.4 Decision Tree

On a basic level, this algorithm takes a set of labels *l* and a matrix of features *f* and scans through each of the features to finds the feature f_j that is best associated with a label. After finding this feature, it then finds the best way to split the data on f_j , resulting in two branches. It then

follows to repeat this for the resulting two branches. The algorithm keeps going and then terminates once it reaches a unique cluster. Then for a new data point *x*, it simply follows the tree until it reaches the unique cluster.

2.2 Unsupervised Classification Methods

2.2.1 K-means

The *k*-means clustering algorithm is an iterative unsupervised algorithm with the goal of partitioning data into *k* clusters. Usually *k* must be determined, either through a different algorithm or through user choice. The algorithm starts by choosing initial values for *k* distinct means and computes the difference of each data point to each of the *k* means. Next, it labels each data point as belonging to one of the *k* means. Once completed, the algorithm then finds the the mean of the newly formed groups and starts again from the beginning with the new means. It proceeds to do this until the solution converges.

Essentially this algorithm tries to solve an optimization problem, in particular trying to minimize:

$$\sum_{j=1}^{k} \sum_{x_j \in D'_j} \| x_j - \mu_j \|^2 \quad (4)$$

Where μ_j denotes the mean of the *j*th cluster and D'_j denotes the subdomain of the data associated with the cluster. One thing to keep in mind is that a shortcoming of *k*-means clustering is that it is sensitive to the initial guess even though modern versions of the algorithm also provide more sophisticated ways to initialize the initial guess. As such, it is also important to cross validate the data with a training and test data set.

2.2.2 Gaussian Mixture Models (GMM)

GMM assumes that all the data are generated from a mixture of gaussian distributions with unknown parameters. Essentially it draws ellipsoidal clusters around the data, which match certain parameters such as their means and covariances and finds the optimal parameters for that certain data set. It will iteratively find the best parameters for the sets and these ellipsoidal clusters serve as our classifications. GMM also runs through a series of different ellipsoidal shapes and finds the best one that can fit the data.

2.3 Signal Smoothing

2.3.1 Gaussian Function

A Gaussian function is a function of the form (5) for arbitrary real constants a,b, and nonzero c. The graph of the function has a “bell curve” shape.

$$f(x) = ae^{-(x-b)^2/2c^2} \quad (5)$$

2.3.1 Signal Smoothing

Signal temporal smoothing is the idea of modifying the temporal signal at each time point with some function of the local time point data. For example, with Gaussian filtering, the signal is convolved with the Gaussian filter specified with (5).

2.4 Singular Value Decomposition (SVD) and Principal Component Analysis (PCA)

2.4.1 Singular Value Decomposition (SVD) and low-rank approximation

The singular value decomposition (SVD) is a factorization of matrix:

$$A = U\mathbf{\Sigma}V^* \quad (6)$$

with the following three matrices:

$$U \in \mathbb{C}^{m\times m} \text{ is unitary;}$$

$$V \in \mathbb{C}^{n\times n} \text{ is unitary;}$$

$$\mathbf{\Sigma} \in \mathbb{C}^{m\times n} \text{ is diagonal.}$$

Additionally, the diagonal entries of $\mathbf{\Sigma}$ are nonnegative and ordered from largest to smallest so that

$$\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_p \geq 0 \quad (7)$$

where p = min(m,n).

For any N, 0<N<r, define the partial sum

$$A_N = \sum_{j=1}^{N} \sigma_j u_j v_j^* \quad (8)$$

If N = min{m,n}, define $\sigma_{N+1} = 0$. Then

$$\|A - A_N\|_2 = \sigma_{N+1} \quad (9)$$

Thus the SVD gives a type of least-square fitting algorithm, allowing us to project the matrix onto low-dimensional representations in a formal, algorithmic way.

2.4.2 Principal Component Analysis (PCA)

Principal component analysis (PCA) is a statistical procedure that transforms observations of possibly correlated variables into a set of values of linearly uncorrelated variables called principal components. PCA is mostly used as a data analysis tool to extract potentially low-dimensional information from the data. PCA can be done by eigenvalue decomposition of a data covariance matrix or singular value decomposition of a data matrix. Here, we introduce how to use SVD to perform PCA.

Given the data matrix $X \in \mathbb{C}^{m\times n}$, where m is the number of variables and n is the number of observations, we consider its covariance matrix:

$$C_X = \frac{XX^T}{n-1} \quad (10)$$

C_X is a square, symmatrix mxm matrix. Its diagonal terms are measure variance of each variable and its off-diagonal terms are the covariance between variables. The goal is to find some basis in which the transformed variables statistically independent (hence no redundancy in the data and off-diagonal terms of C_X are zero, C_X diagonal), and the assumption is that the dynamic of interest can be described by the transformed variables with larger variances.

We use SVD to diagonalize C_X. By applying the SVD on the data matrix X, we get

$$X = U\mathbf{\Sigma}V^* \quad (11)$$

If we transform the data into the new basis:

$$Y=U^*X \quad (12)$$

then the covariance matrix for the transformed data is:

$$C_Y = \frac{\Sigma^2}{n-1} \quad (13)$$

which is diagonal. Thus, the principal component projection is Y and the diagonal variances are σ_i^2, where σ_i are diagonal terms of $\mathbf{\Sigma}$. By looking at the first few principal components, PCA provides a lower-dimensional picture of the data that explains most of the variance. One way of identifying the number of important principle components is to plot the explained variance for each mode (Equation 14).

$$\%\text{Variance\_i} = \frac{\sigma_i^2}{\sum_{k=1}^{n} \sigma_k^2} \quad (14)$$

Note that PCA is sensitive to the relative scaling of the original variables, hence is usually applied after the initial data is normalized. The normalization (z-score) of each attribute consists of subtracting each data for each variable from its mean and then divided by its standard deviation. Also, in order to get rid of the spiking variability, it is a common practice to average across trials and temporally smooth each neuron's response before applying PCA.

2.5 Dynamic mode decomposition (DMD)

2.5.1 Dynamic mode decomposition (DMD)

Dynamic mode decomposition is a mathematical method that exploits the low-dimensional dynamics of a system. Given data collected for N measurements at M time points with fixed intervals ($t_1, t_2, ..., t_m$), the DMD method approximates the modes of the Koopman operator A, the linear operator that describe the describes the mapping between each time steps

$$x_{j+1} = Ax_j \quad (15)$$

Specifically, the DMD algorithm computes the leading eigenvalues and eigenvectors of A, which maps the first m-1 timesteps of data $X_1^{m-1} = [x_1, x_2, ..., x_{m-1}]$ to the latter m-1 timesteps $X_2^m = [x_2, x_3, ..., x_m]$.

Suppose that the last data point x_m is determined by

$$x_m = \sum_{j=1}^{m-1} k_j x_j + r \quad (16)$$

where the k_j 's are the coefficients of the Krylov space basis vectors and the residual error r is orthogonal to the Krylov space. Then

$$X_2^{\,m} = X_1^{\,m-1} S + r * e_{m-1}^{\,*} \quad (17)$$

where [*] is the conjugate transpose operator, e_{m-1} is the $(m-1)^{th}$ unit vector and

$$\mathbf{S} = \begin{bmatrix} 0 & 0 & \cdots & 0 & 0 & k_1 \\ 1 & 0 & \cdots & 0 & 0 & k_2 \\ 0 & 1 & \ddots & 0 & 0 & k_3 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 & k_{m-2} \\ 0 & 0 & \cdots & 0 & 1 & k_{m-1} \end{bmatrix} \quad (18)$$

is a nx(m-1) matrix.

By our assumption that $x_{j+1} = Ax_j$ (Equation 15), we have

$$X_1^{\,m-1} = [x_1 \; Ax_1 \; A^2 x_1 ..., A^{m-2} x_1] \quad (19)$$

$$AX_1^{\,m-1} = X_2^{\,m} \quad (20)$$

Then, observe that

$$AX_1^{\,m-1} = X_2^{\,m} \approx X_1^{\,m-1} S \quad (21)$$

Thus, the eigenvalues of S approximates some of the eigenvalues of the unknown Koopman operator A. If we perform a SVD on

$$X_1^{\,m-1} = U \Sigma V^* \quad (22)$$

where

$$U \in C^{nxl} \text{ is unitary,}$$

$$\Sigma \in C^{lxl} \text{ is diagonal,}$$

$$V \in C^{(m-1)xl} \text{ is unitary,}$$

(the parameter l is chosen so as to reduce rank of $X_1^{\,m-1}$ as much as possible while retaining fundamental structures), then we can estimate:

$$X_2^{\,m} \approx X_1^{\,m-1} S \quad (23)$$

$$X_2^{\,m} \approx U \Sigma V^* S \quad (24)$$

$$S \approx V \Sigma^{-1} U^* X_2^{\,m} \quad (25)$$

Now, instead of computing the matrix S directly, we calculate the mathematically similar matrix

$$S' = U^* X_2^{\,m} V \Sigma^{-1} \quad (26)$$

From the similarity relation

$$AU \approx US' \quad (27)$$

We can determine the eigenvalues of A from the lower-rank matrix S'. Solving the eigenvalue problem:

$$S'w_j = \mu_j w_j,\ j = 1, 2, ..., l \quad (28)$$

will yield the DMD eigenvalues μ_j and the eigenvectors w_j of the matrix S'. Then, the jth DMD basis function mode, or the approximate jth eigenvector of the Koopman operator A, is:

$$\varphi_j = U w_j\ . \quad (29)$$

The extracted DMD eigenvalues can also be used to reconstruct the data x_{DMD} at time t. To do this, first convert the DMD eigenvalues to Fourier modes by

$$w_j = \frac{ln(\mu_j)}{dt} \quad (30)$$

Then, the real part of w_j regulates the growth or decay of the DMD basis function modes, while the imaginary part of w_j drives oscillations in the DMD modes. The DMD reconstruction of the data x_{DMD} at time t for any time is given by

$$x_{DMD}(t) = A^t x_1 = \sum_{j=1}^{l} b_j \varphi_j e^{w_j t} = \Phi \Omega^t b \quad (31)$$

where

$$\mathbf{\Omega} = \begin{bmatrix} e^{\omega_1} & 0 & \cdots & 0 \\ 0 & e^{\omega_2} & \ddots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & e^{\omega_\ell} \end{bmatrix}, \quad (32)$$

and the vector

$$b \approx \Phi^{-1} x_1\ (\ b = (\Phi^* \Phi)^{-1} \Phi^* x_1\ ) \quad (33)$$

contains the initial amplitudes for the modes.

2.5.2 DMD slow and fast dynamics separation

Many complex systems contain dynamics at multiple scales. Given data X can be reconstructed with the DMD technique:

$$X_{DMD} = \sum_{j=1}^{l} b_j \varphi_j e^{w_j t} := \Phi \Omega^t b \ (34).$$

Then, the portion of the data that does not change in time must have an associated Fourier mode w_j that is located near the origin in complex space; and the portion of the data that changes a lot in time have an associated Fourier mode with large norms. Thus, we can extract the slow and fast dynamics in the data by plotting the trajectories that correspond to Fourier modes with large/small imaginary parts.

Section III: Algorithm Implementation and Development

We start by loading our data, which comes in the form of a .npy file, using the load function from the numpy library. Using a for loop, we next average the activity for each neuron over the 97 trials, ending up with a 398 x 2430 matrix which matches the number of neurons recorded and signal length respectively.

3.1 Classification

3.1.1 Supervised Classification

SVD is performed using the svd function from the numpy.linalg module. SVM is performed using the LinearSVC function from the sklearn.svm module. kNN is performed using the KNeighborsClassifier function from the sklearn.neighbors module. Naive Bayes is performed using the GaussianNB function from the sklearn.naive_bayes module.

In each test, temporal data for 250 neurons are randomly selected as training set to train each of the four classification models and data for randomly selected 100 neurons are used to test the trained models. In order to test the accuracy of each classification model, models are trained 100 times with different randomly chosen train and test data sets and the performance across trials is averaged.

3.1.2 Unsupervised Classification

KMeans is performed using the Kmeans function from the sklearn.cluster module. GMM is performed using the GMM function from the sklearn.mixture module. The number of cluster is kept as 7, which is the number of brain areas included in the data.

For each identified cluster, the brain area for each area are identified and a pie chart of the percentage of neurons from different neurons for each identified cluster is plotted.

3.2 PCA

3.2.1 Data Preprocessing for PCA

The data contains the recorded activity for N=398 neurons at M=2430 consecutive time points for K=97 trials. Thus, the data is of dimension 398 x 97 x 2430. First, neurons with low mean firing (less than one spike per second) are excluded. This is because near zero variance for any neuron can lead to numerical instability, and we are also primarily interested in neurons driven by the stimulus. A total of 14 neurons are removed and the data has shape 384 x 2430. For PCA, we also z-score the activity of each neuron. That is, for each neuron, we first subtract the data by its mean, and then divide by its standard deviation. We used the z-score function from sciy.stats module to perform z-score. The activity is each neuron is normalized (z-scored) because otherwise PCA can be dominated by neurons with the highest modulation. Next, the data can be smoothed through across time. We implemented both Gaussian kernel smoothing and taking binned spike counts with different width. For Gaussian smooth,

$$g(t) = e^{-a(t-\tau)^2} \quad (34)$$

we take a=0.01 and 0.1. For bined spike counts, we take every 5 bins and 10 bins (⅛ sec and ⅓ sec). The resulting processed data is of dimension N x M, where N=398 is the number of neurons and M (varies) is the number of time points.

3.2.2 PCA

Given the data matrix N x M, where N is the number of neurons and M is the number of time points, we use SVD (from numpy.linalg module) to factorize the data matrix.

$$X = U\mathbf{\Sigma}V^* \quad (35)$$

The projection of the system onto the principal component is given by

$$Y = U^*X, \text{ or } \mathbf{\Sigma}V^* \quad (36)$$

which is a NxT matrix and each row is the value of the system in each principal component through time. Thus, the evolution of the first few principal components (the first few row of Y) provides us with a lower-dimensional picture of the neural population dynamics. To get a sense of how many principal components are necessary to understand the system, we can plot the normalized variance (see section 2.4.2) for each principal component.

3.3 DMD

3.3.1 DMD

To perform DMD given a data of matrix X = NxM, the data is first split into

$$X_1^{m-1} = [x1, x2, \ldots, x\_M\text{-}1] \quad (37)$$

and

$$X_2^{m} = [x2, x3, \ldots, x\_M] \quad (38)$$

Then a SVD is performed on

$$X_1^{m-1} = U\Sigma V^* \quad (39)$$

and function svd from np.linalg module is used. Note that the reduced SVD is used.

$$X_1^{m-1} = U\Sigma V^* \quad (40)$$

where

$$U \in C^{nxl} \text{ is unitary,}$$
$$\Sigma \in C^{lxl} \text{ is diagonal,}$$
$$V \in C^{(m-1)xl} \text{ is unitary.}$$

The singular values s are plotted to choose l so as to reduce rank of X_1^{m-1} as much as possible while retaining fundamental structures. L is chosen such that 90% variance is explained.

Next, compute the matrix S'.

$$S' = U^* X_2^{m} V \Sigma^{-1} \quad (41)$$

and find its eigenvalues and eigenvectors. Function eig from the np.linalg is used. Will yield the DMD eigenvalues μ_j and the eigenvectors w_j of the matrix S'. Then, find the approximate jth eigenvector of A using

$$\varphi_j = Uw_j \quad (42)$$

The eigenvalues for A are the same for S'. The initial amplitude for the modes can be computed with

$$b = (\Phi^*\Phi)^{-1}\Phi^* x_1 \quad (43).$$

The lstsq function from the np.linalg module is used for calculating b. Fourier modes can be calculated by

$$w_j = \frac{ln(\mu_j)}{dt} \quad (44).$$

Then, the DMD reconstruction of the data x_{DMD} at time t for any time is given by

$$x_{DMD}(t) = \Phi\Omega^t b \quad (45),$$

where

$$\Omega = \begin{bmatrix} e^{\omega_1} & 0 & \cdots & 0 \\ 0 & e^{\omega_2} & \ddots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & e^{\omega_\ell} \end{bmatrix} \quad (46).$$

3.3.2 DMD fast and slow dynamics separation

In order to identify the slow and fast dynamics separated by DMD, we first plot the approximated Fourier modes for the Koopman operator A. From section 2.5, Fourier modes with larger imaginary parts correspond to faster oscillations in the activity while fourier modes with smaller imaginary parts correspond to slower oscillations in the activity. Then, we can reconstruct the dynamics of the data with only the few Fourier modes with large imaginary parts using Equation 45. The eigenvectors of A also gives the initial condition for each neuron. We also try to plot the initial conditions of the neurons by area to test whether there is any area-specific clustering.

Section IV: Computational Results

4.0 Raw Data

The data contains the recorded activity for N=398 neurons at M=2430 consecutive time points for K=97 trials. Thus, the data is of dimension 398 x 97 x 2430. The raw data is shown in Figure 1. The 398 neurons are from 7 different brain regions and the number of neurons recorded from each region is shown in Table 1. The data is collected at the Allen Institute and provided by J.Shang.

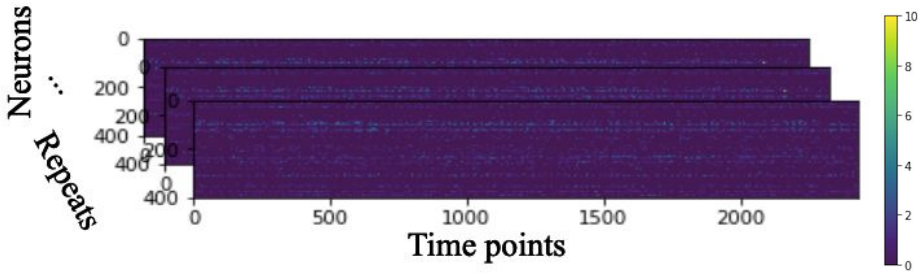


Figure 1: Raw data of dimension 398 (number of neurons) x 97 (number of repeats) x 2430 (number of data collection time points).

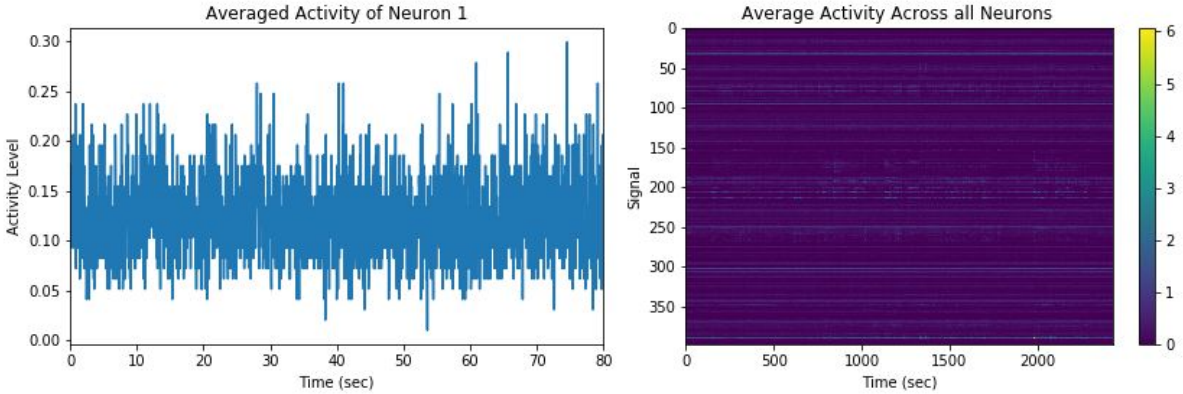


Figure 2: Trial-averaged data. Left: averaged activity for neuron 1 through time. Right: Averaged activity across trials for all neurons.

Brain Region	Number of Neurons
Cornus Ammonus (CA)	110
Anteromedial visual area (VISam)	55
Dentate gyrus (DG)	50
Primary visual area (VISp)	20
Lateral visual area (VISl)	71
Anterolateral visual area, layer 1 (VISal)	41
Rostrolateral area (VISrl)	51

Table 1: Number of neurons recorded from each brain region.

4.1 Classification

4.1.1 Supervised Classification

For each supervised classification methods included (K-Nearest Neighbors, Support Vector Machine (SVM), Naive Bayes Classification, and Decision Tree), 100 trials of classification are ran for different randomly selected train and test datasets. We then plot the accuracy levels, along with their means and standard deviations as shown in Figure 3. We find that all four methods reaches classification accuracy higher than chance level (1/7=14.3%). Moreover, the K-neighborhood method reaches the highest accuracy of 61.98%. This shows that the population activity patterns contain area-specific information that can be extracted by the supervised classification methods applied for classification.

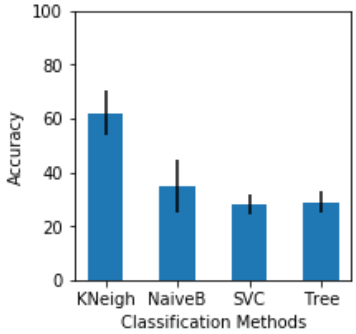


Figure 3: Averaged classification accuracy for each supervised classification method.

4.1.2 Unsupervised Classification

For unsupervised classifications, we compare how the algorithm groups the different neurons to their actual locations. We see that certain clusters consist entirely of one region, such as Cluster 6 in K-means and Clusters 5 and 6 in GMM. However, the number of neurons assigned to each cluster is relatively small compared to the rest of the assignments. We do see some instances of regions being more likely to be assigned to certain clusters such as the b'DG' region to Cluster 4 in the k-means algorithm or the b'VISI' region to Cluster 2 outside of Cluster 6.

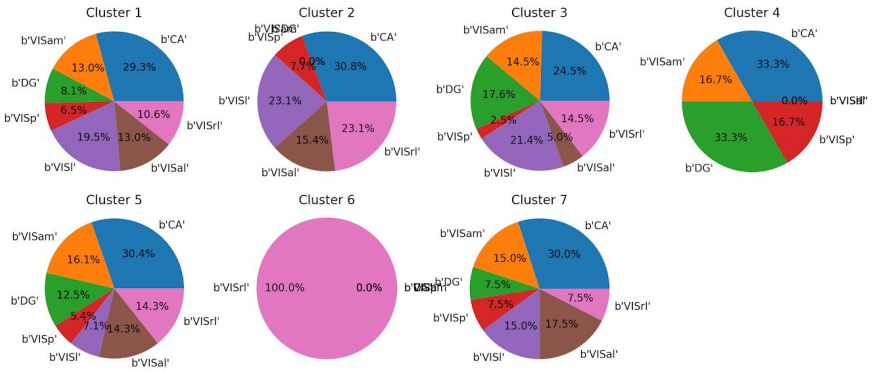


Figure 4: Classification using K-means, split to show what portion of neurons from each location belongs to each cluster

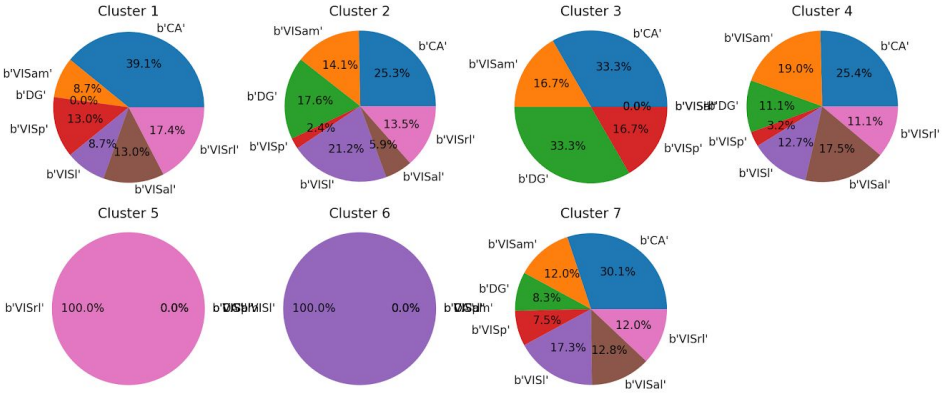


Figure 5: Classification using GMM, split to show what portion of neurons from each location belongs to each cluster

4.2 PCA

4.2.1 Data Preprocessing for PCA

The data is first averaged across trials and then normalized (z-scored, see Section 2.4.2) before PCA is applied. The normalized data is shown in Figure 6.

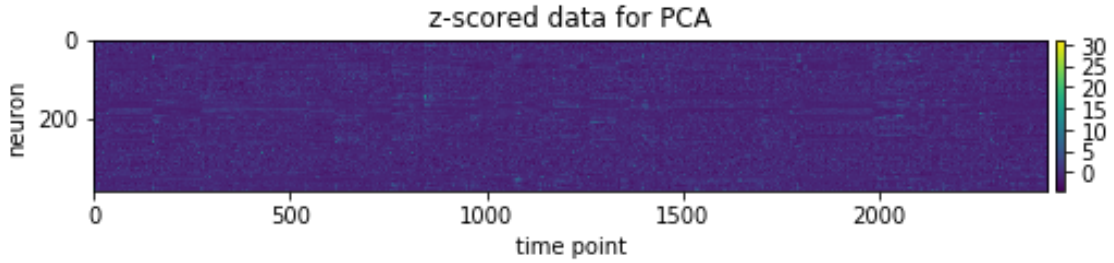


Figure 6: Preprocessed (trial-averaged and z-scored) data for PCA.

The data is then temporally smoothed, either with a Gaussian kernel with a=0.01 and 0.1, or binned by taking the averaged firing rates of every 5 or 10 data collection time points. The Gaussian kernel used are shown in Figure 7.

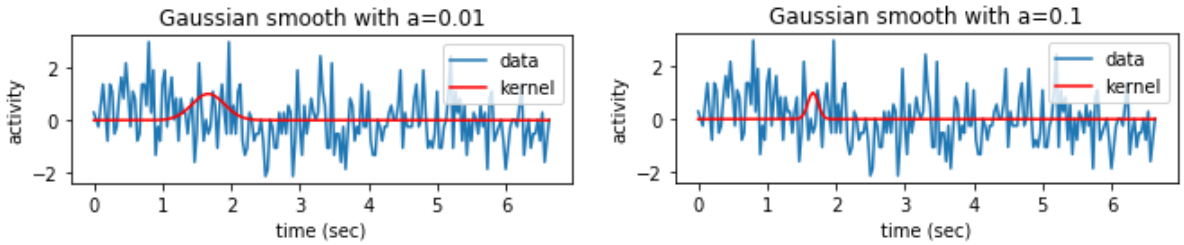


Figure 7: Gaussian kernel used for a=0.01 and 0.1 (Equation 34) overplotted on data.

4.2.2 PCA extraction of low-dimensional dynamics

For given data (with or without temporal smoothing), PCA is performed (see section 2.4.2). First, the singular values of the covariance matrix are plotted, along with the percentage variance explained to see how many principal components are important to describe the dynamics of the system. Then, the temporal evolution of the first three principal components are plotted, along with the trajectory of the temporal evolution for the first two principal components in 2D and for the first three principal components in 3D. The results for each data (without temporal smoothing and with various temporal smoothing) are shown the following Figures 8-12.

We find that the data without temporal smoothing shows the most spread of variance. This suggests a less dominant low-rank dynamics of the system, potentially due to the noisy fluctuation in the data. In contrast, data after temporal smoothing shows that the first few principal component explains more percentage variance (explained variance for the first few principal component is higher in the top left plots). The temporal smoothing with Gaussian kernel of a=0.01 shows the highest explained variance for the first few principal components. The temporal projection trajectory for the first two or three principle components is also smoother than the trajection from data with other temporal smoothings.

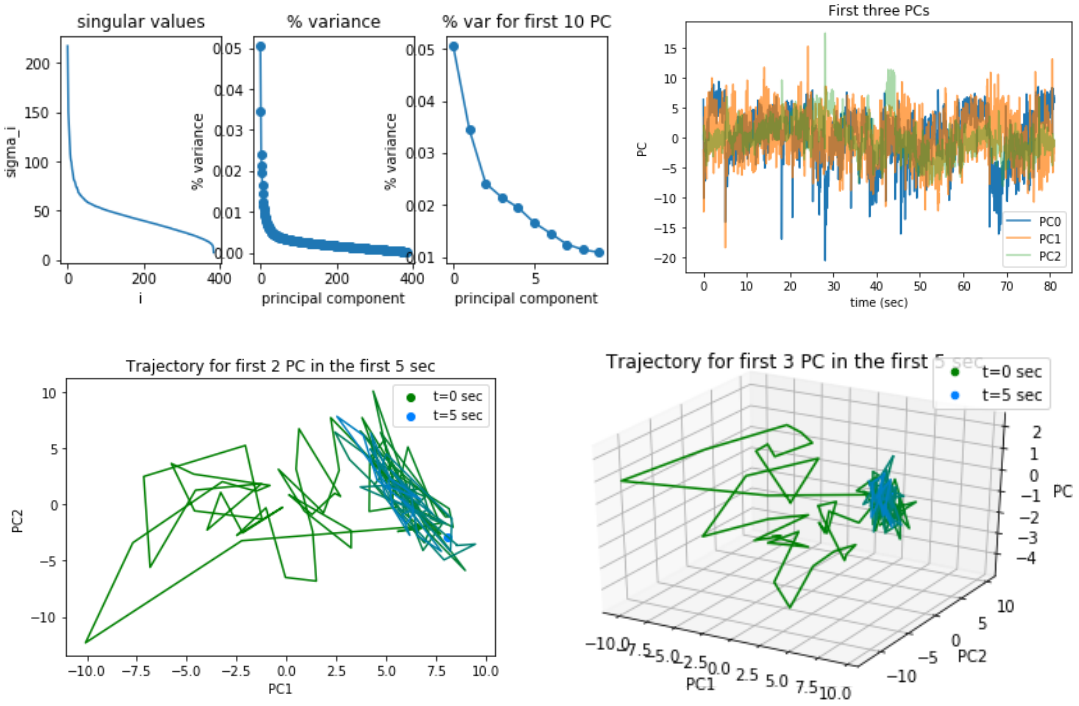


Figure 8: PCA results for data without temporal smoothing. Top left: plots of singular values and percentage variance explained. Top right: temporal evolution for the first three principle components. Bottom left: Projection trajectories for the first two principal components in 2D. Bottom right: Projection trajectories for the first three principal components in 3D.

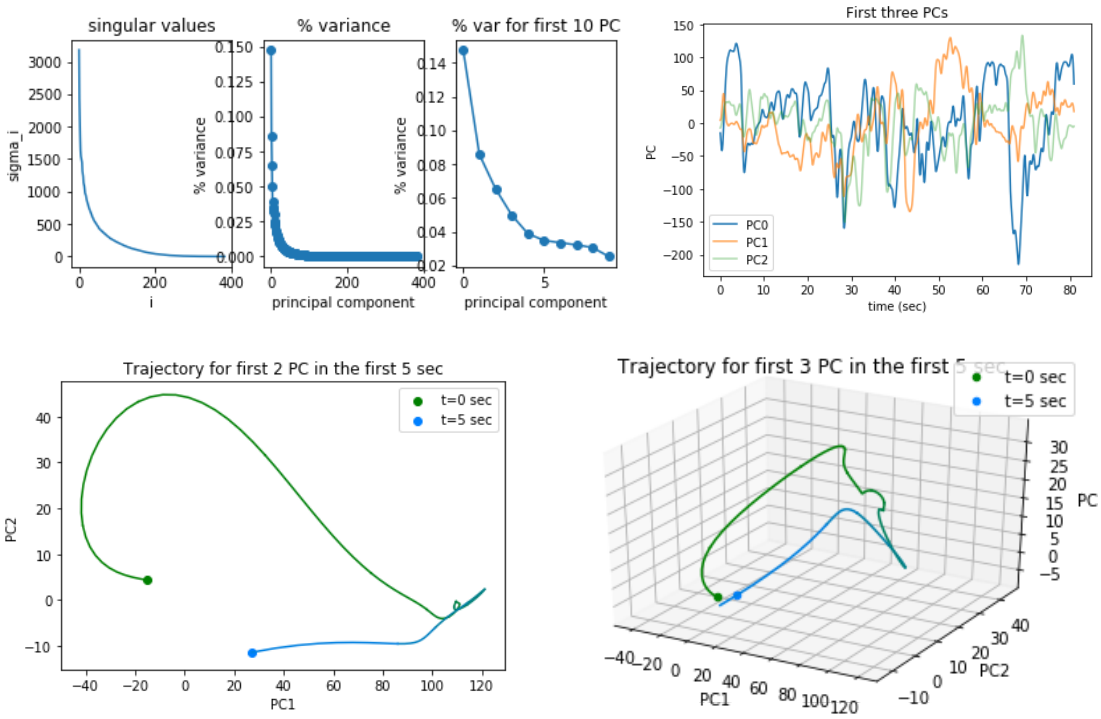


Figure 9: PCA results for data with Gaussian smoothing with a=0.01.

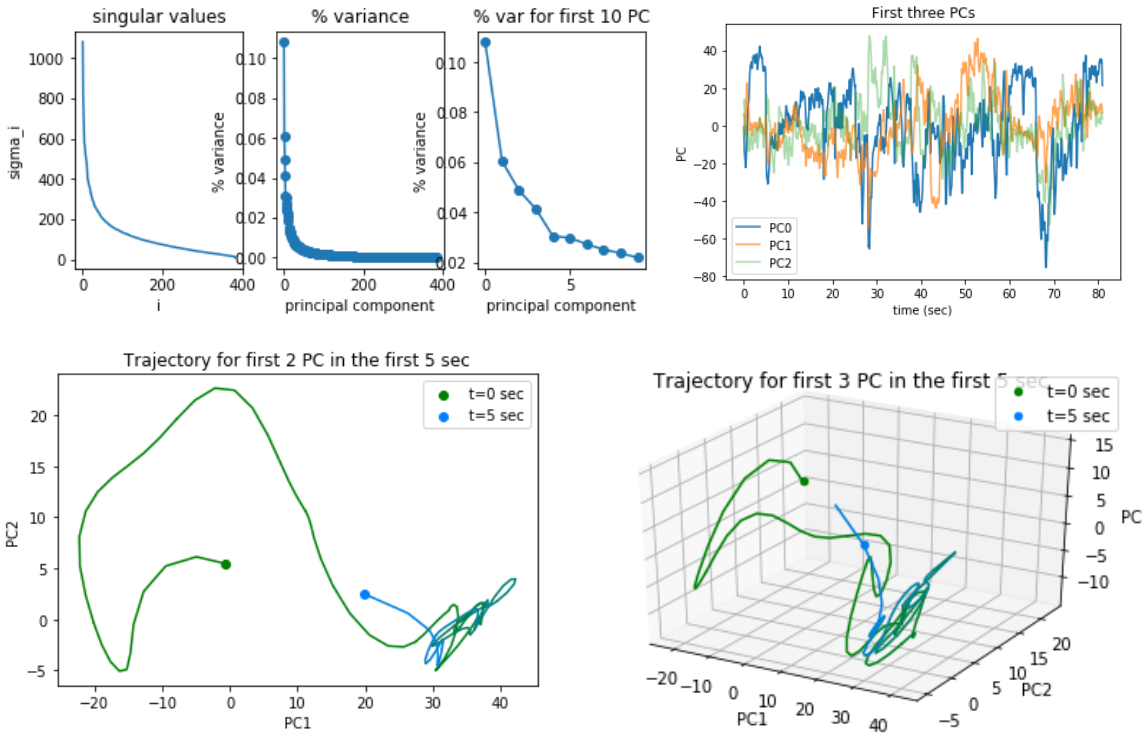


Figure 10: PCA results for data with Gaussian smoothing with a=0.1.

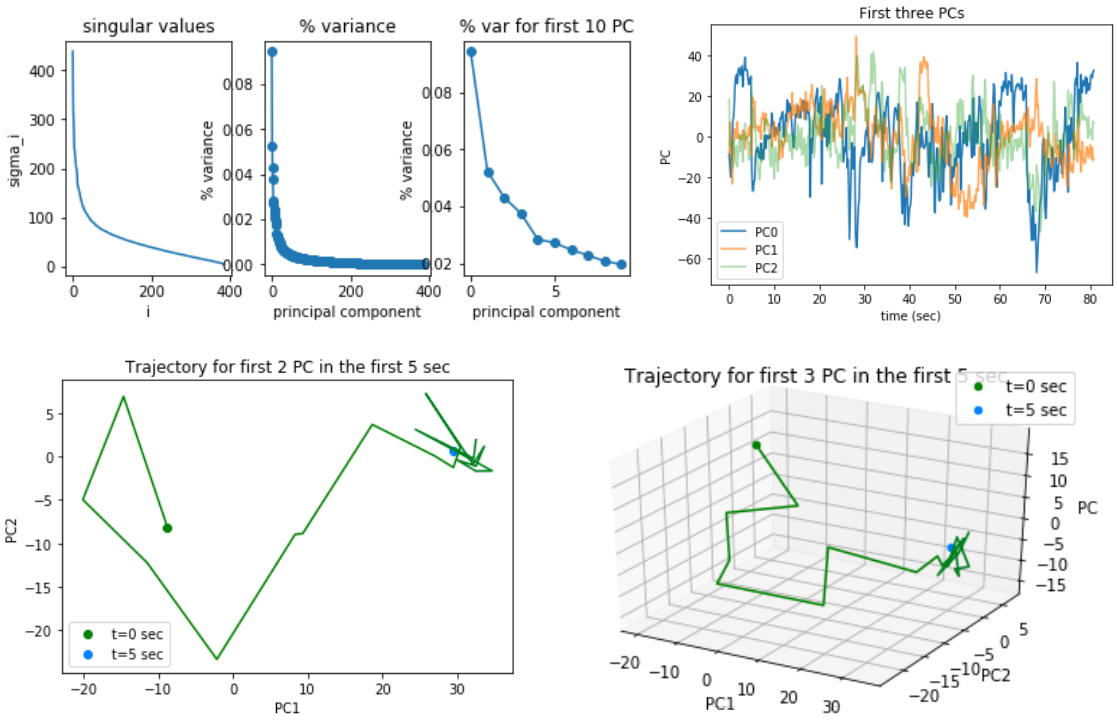


Figure 11: PCA results for data with binning of every 5 time points.

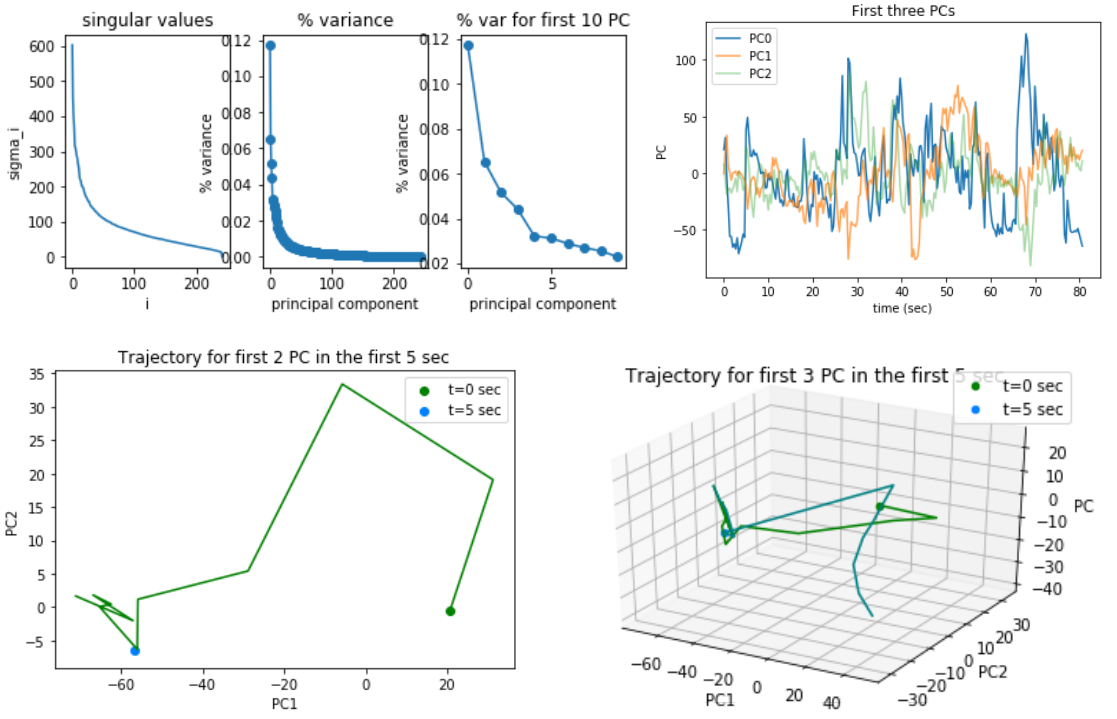


Figure 12: PCA results for data with binning of every 10 time points.

4.3 DMD

To perform DMD, first the data for the first (M-1) time point is extracted: $X_1^{m-1} = [x_1, x_2, ..., x_{m-1}]$ and SVD is performed on X_1^{m-1}. In order to determine the dimensionality of the lower-rank approximation $X_1^{m-1} = U\Sigma V^*$, we plot the explained variance from the SVD and chose l such that 90% of the variance is explained. For the data, l is chosen to be 13. The explained variance for all modes are shown in Figure 13

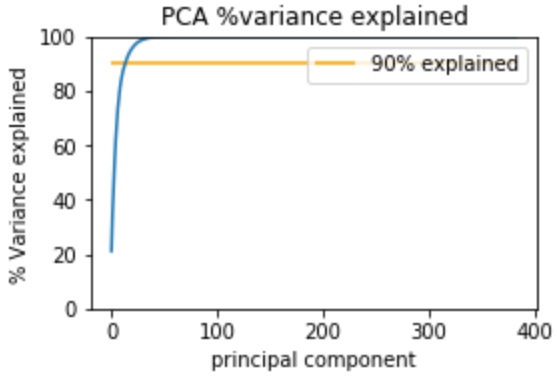


Figure 13: Explained variance of SVD for X_1^{m-1}.

Then, given the low-rank approximation $X_1^{m-1} = U\Sigma V^*$, we calculate the lower-rank matrix S' $= U^* X_2^{\,m} V\Sigma^{-1}$. From the similarity relation $AU \approx US'$, we can determine the eigenvalues of A from the lower-rank matrix S'. The DMD reconstruction of the data x_{DMD} at time t for any time is given by $x_{DMD}(t) = A^t x_1 = \sum_{j=1}^{l} b_j \varphi_j e^{w_j t} = \Phi\Omega^t b$. The Fourier mode w_j is shown in Figure 14.

Due to Fourier modes with larger imaginary parts capturing the larger oscillations in the system, we reconstruct the data using the top three Fourier modes with the largest imaginary parts. The original data and the reconstructed data (for the first 5 seconds) is shown in Figure 14. Note that the reconstructed data does capture some of the dynamics in the original data even though only 3 of the Fourier modes are used for reconstruction.

We also observe that the real part for all the Fourier modes are near zero. This suggests that there is no long-term dampening of the activity. However, the wide spread of the Fourier mode along the imaginary axis suggests that there are different frequencies of oscillations in the population dynamics.

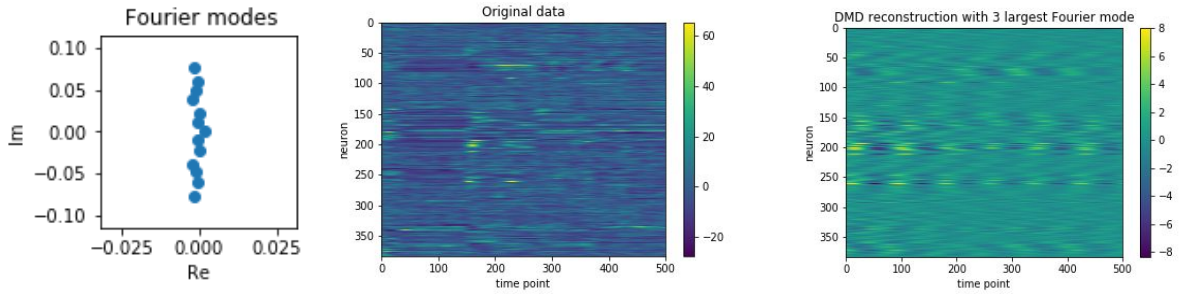


Figure 14: Left: Fourier modes for the operator A. Middle: Original data for the first 5 seconds. Right: DMD reconstructed data with the top three Fourier modes with largest imaginary parts.

We also looked at the whether there is any area-specificity in the eigenvector of A extracted for different Fourier modes. However, we did not find any area-specific clustering for the eigenvector of A that correspond to both the Fourier mode with largest and smallest norm. The plots for the entries for both eigenvectors are shown in Figure 15.

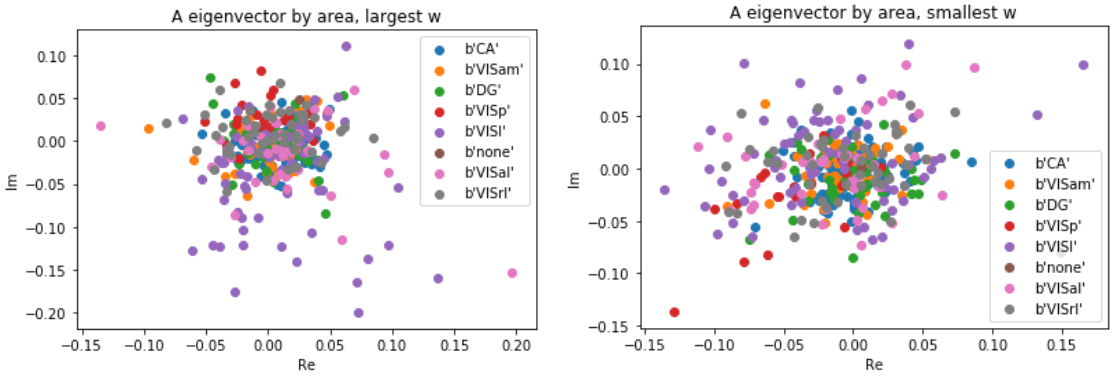


Figure 15: Entries of eigenvectors of A that correspond to the largest (left) and smallest (right) normed Fourier mode, plotted in the complex plane.

Section V: Summary and Conclusions

We see that there may be correlation with neuron activity and its location. So in future studies, we can explore the existence of location-based activity more. It is also an interesting future direction to explore why the supervised classification, K-neighbors, has significantly higher accuracy than the rest of the methods.

For the second part of the paper, we used PCA to extract low-dimensional dynamics of the high-dimensional neural data. We observe that there is no significantly dominating principle components. However, the explained variance becomes less widespread when the data is temporally smoothed. Specifically, a wide Gaussian kernel smoothing with a=0.01 shows the smoothest projection trajectory for the first few two/three principle components. In the future, it will be interesting to measure some observable condition dynamics such as pupil dilation (as a

metric for attention or arousal) and running speed, and test whether any of the identified low-dimensional dynamics has any correspondence with the observable measurements.

For the last part of the paper, we showed that the DMD method can be applied to the data to separate slow and fast changing dynamics of the population activity. Moreover, we showed that the DMD reconstruction with the Fourier modes that correspond to the first three fastest oscillations can already capture some of the dynamics in the original data. However, we did not find any area-specific information in the eigenvectors of the approximated operator A. Note that DMD performance is dependent on the sampling rate and may not give close approximations for the system at time points too far away from the initial time point. Therefore, future work can explore the extraction of low-dimensional dynamics with other methods.

In summary, we explored various analyses of the large-scale neural recordings available by the recent technological development of Neuropixel. We show that the population activity while the animal is passively viewing natural movie clips contains brain-area specific information and can be extracted using both supervised and unsupervised classification methods. We also extracted some low-dimensional dynamics from the data using both PCA and DMD. In the future, we will extend the work to other methods to understand the neural functions from the population perspective. This work takes advantage of the newly available population data from the neural system and contribute to the understanding of the neural system using population data.

Reference

Jun, James J., and Steinmetz, Nicholas A., et al. “Fully integrated silicon probes for high-density recording of neural activity”. *Nature* 551 (2017): 232-236. *Nature International Journal of Science.* Web. 22 Mar. 2019.

00_Classification

March 22, 2019

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import pandas as pd

from sklearn import svm
from sklearn.svm import LinearSVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
```

In [ ]:
```
data = np.load('data.npy')
```

In [ ]:
```
data = np.mean(data, axis=1)
```

In [ ]:
```
data.shape
```

In [ ]:
```
np.save('classification_data', data)
```

In [ ]:
```
key = pd.read_csv('key.csv')
```

In [ ]:
```
area_list = key['ccf']
```

In [ ]:
```
len(area_list)
```

In [ ]:
```
area_list_unique = []
for area in area_list:
    if area not in area_list_unique:
        area_list_unique.append(area)
```

In [ ]:
```
area_list_unique
```

In [ ]:
```
area_ind_list = []
for key in area_list_unique:
    area_ind= [i for i, area in enumerate(area_list) if area==key]
    area_ind_list.append(area_ind)
```

In [ ]:
```
area_ind_len = []
for i in range(8):
    area_ind_len.append(len(area_ind_list[i]))
```

```
In [ ]: area_ind_list[5]

In [ ]: data = np.delete(data, 222, axis=0)

In [ ]: data.shape

In [ ]: np.save('classification_data', data)

In [ ]: area_list = np.asarray(area_list)

In [ ]: area_list = np.delete(area_list, 222)

In [ ]: area_list.shape

In [ ]: np.save('classification_area', area_list)

In [ ]: area_list_unique = np.asarray(area_list_unique)
        area_list_unique = np.delete(area_list_unique, 5)

In [ ]: np.save('classification_area_list_unique', area_list_unique)

In [ ]: area_list_unique = area_list_unique.tolist()

In [ ]: area_ind_list = []
        for area in area_list:
            area_ind_list.append(area_list_unique.index(area))

In [ ]: np.save('classification_area_ind_list', area_ind_list)

In [ ]: len(area_ind_list)
```

0.1 Unsupervised Classification

0.2 k-means

```
In [ ]: from sklearn.cluster import KMeans

In [ ]: kmeans = KMeans(n_clusters=7, random_state=0).fit(data)

In [ ]: prediction = kmeans.labels_

In [ ]: def plot(prediction, title):
            plt.figure(figsize=(14,6))
            for i_label in range(7):
                pred_ind = [i for i, pred in enumerate(prediction) if pred==i_label]
                area_num = np.zeros((7,))
                for i_pred in pred_ind:
                    area = area_ind_list[i_pred]
                    area_num[area]+=1

                labels = area_list_unique
```

```
                sizes = area_num

                plt.subplot(2,4,i_label+1)
                plt.pie(sizes, labels=labels, autopct='%1.1f%%')
                plt.axis('equal')
                plt.title('Cluster '+str(i_label+1))
                plt.savefig(title+'.png', dpi=200)
```

```
In [ ]: plot(prediction, 'k-means')
```

0.2.1 GMM

```
In [ ]: from sklearn.mixture import GaussianMixture
```

```
In [ ]: gmm = GaussianMixture(n_components=7)
```

```
In [ ]: data.shape
```

```
In [ ]: gmm.fit(data)
```

```
In [ ]: labels = gmm.predict(data)
```

```
In [ ]: plot(labels, 'GMM')
```

0.3 Supervised Classification

```
In [ ]: data.shape
```

```
In [ ]: area_ind_list = np.asarray(area_ind_list)
        area_ind_list.shape
        label = area_ind_list
```

```
In [ ]: def get_data(data, label, num_train=250, num_test=100):
            train_ind = np.random.choice(data.shape[0], num_train, replace=False)
            left_ind = [i for i in np.arange(data.shape[0]) if i not in train_ind]
            test_i= np.random.choice(data.shape[0]-num_train, num_test, replace=False)
            test_ind = [left_ind[i] for i in test_i]

            train_data = [data[i,:] for i in train_ind]
            train_label = [label[i] for i in train_ind]
            test_data = [data[i,:] for i in test_ind]
            test_label = [label[i] for i in test_ind]

            train_data = np.asarray(train_data)
            train_label = np.asarray(train_label)
            test_data = np.asarray(test_data)
            test_label = np.asarray(test_label)
            return train_data, train_label, test_data, test_label
```

```
In [ ]: def test_KNeigh(data, label, num_trial=100):
            acc_list = np.zeros((num_trial,))
            for i_trial in range(num_trial):
                train_data, train_label, test_data, test_label = get_data(data, label)
                neigh = KNeighborsClassifier(n_neighbors=3)
                neigh.fit(train_data, train_label)
                pred = neigh.predict(test_data)
                acc = 100*np.sum((pred-test_label)==0)/50
                acc_list[i_trial] = acc
            return acc_list

In [ ]: def test_NaiveB(data, label, num_trial=100):
            acc_list = np.zeros((num_trial,))
            for i_trial in range(num_trial):
                train_data, train_label, test_data, test_label = get_data(data, label)
                gnb = GaussianNB()
                pred = gnb.fit(train_data, train_label).predict(test_data)
                acc = 100*np.sum((pred-test_label)==0)/50
                acc_list[i_trial] = acc
            return acc_list

In [ ]: from sklearn.svm import SVC
        def test_SVC(data, label, num_trial=100):
            acc_list = np.zeros((num_trial,))
            for i_trial in range(num_trial):
                train_data, train_label, test_data, test_label = get_data(data, label)
                clf = SVC(gamma='auto')
                clf.fit(train_data, train_label)
                score_SVM = clf.score(test_data, test_label)
                acc_list[i_trial] = 100*score_SVM
            return acc_list

In [ ]: from sklearn.tree import DecisionTreeClassifier
        def test_Tree(data, label, num_trial=100):
            acc_list = np.zeros((num_trial,))
            for i_trial in range(num_trial):
                train_data, train_label, test_data, test_label = get_data(data, label)
                clf = DecisionTreeClassifier()
                clf.fit(train_data, train_label)
                score_RF = clf.score(test_data, test_label)
                acc_list[i_trial] = 100*score_RF
            return acc_list

In [ ]: acc_list = [[] for i in range(4)]
        acc_list[0] = test_KNeigh(data, label)
        acc_list[1] = test_NaiveB(data, label)
        acc_list[2] = test_SVC(data, label)
        acc_list[3] = test_Tree(data, label)
```

In [ ]:
```
acc_mean = np.zeros((4,))
acc_std = np.zeros((4,))
for i in range(4):
    acc_mean[i] = np.mean(acc_list[i])
    acc_std[i] = np.std(acc_list[i])
```

In [ ]:
```
plt.figure(figsize=(3,3))
plt.bar(np.arange(4), acc_mean, 0.5, yerr=acc_std)
plt.xlabel('Classification Methods')
plt.ylabel('Accuracy')
plt.ylim([0,100])
plt.xticks(np.arange(4), ('KNeigh', 'NaiveB', 'SVC', 'Tree'))
```

01_PCA_data_z-score

March 22, 2019

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from scipy import stats
from mpl_toolkits.axes_grid1 import make_axes_locatable
```

In [ ]:
```
# load the original data
data = np.load('data.npy')
data.shape,
```

In [26]:
```
# take the trial average
data_ave = np.mean(data, axis=1)
```

In [28]:
```
data_ave.shape
```

Out[28]:
```
(399, 2430)
```

In [36]:
```
neuron_delete_ind = []
for i_neuron in range(399):
    s = np.sum(data[i_neuron,:])
    if s < 81:
        neuron_delete_ind.append(i_neuron)
```

In [37]:
```
len(neuron_delete_ind)
```

Out[37]:
```
15
```

In [40]:
```
np.save('neuron_delete_ind_2415', neuron_delete_ind)
```

In [42]:
```
data_ave_cleared = np.delete(data_ave, neuron_delete_ind, 0)
```

In [43]:
```
data_ave_cleared.shape
```

Out[43]:
```
(384, 2430)
```

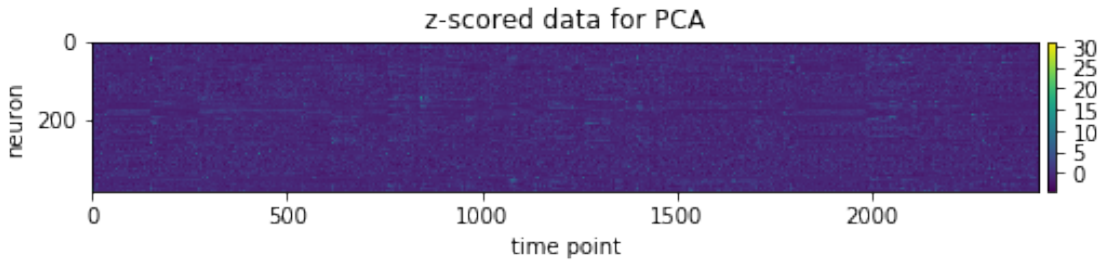
In [44]:
```
# z-score the data
data_zscored = 0*data_ave_cleared
for i_neuron in range(data_ave_cleared.shape[0]):
    data_zscored[i_neuron,:] = stats.zscore(data_ave_cleared[i_neuron,:])
```

In [45]:
```
data_zscored.shape
```

Out[45]:
```
(384, 2430)
```

In [62]:
```
plt.figure(figsize=(8,5))
ax = plt.gca()
im = ax.imshow(data_zscored)
plt.xlabel('time point')
plt.ylabel('neuron')
plt.title('z-scored data for PCA')

divider = make_axes_locatable(ax)
cax = divider.append_axes("right", size="1%", pad=0.05)
plt.colorbar(im, cax=cax)
```

Out[62]:
```
<matplotlib.colorbar.Colorbar at 0x7f6e956f2710>
```



In [63]:
```
np.save('data_zscored', data_zscored)
```

02_PCA_temporal_smoothing

March 22, 2019

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
```

```
In [3]: # load the z-scored data
        data = np.load('data_zscored.npy')
```

```
In [4]: data.shape
```

```
Out[4]: (384, 2430)
```

0.0.1 takeing binned spike count

```
In [5]: def bin_data(data, bin_size):
            data = data.reshape(data.shape[0], -1, bin_size)
            data = np.sum(data, axis=2)
            return data
```

```
In [6]: data_bin_10 = bin_data(data, 10)
        data_bin_10.shape
```

```
Out[6]: (384, 243)
```

```
In [7]: np.save('data_bin_10', data_bin_10)
```

```
In [8]: data_bin_5 = bin_data(data, 5)
        data_bin_5.shape
```

```
Out[8]: (384, 486)
```

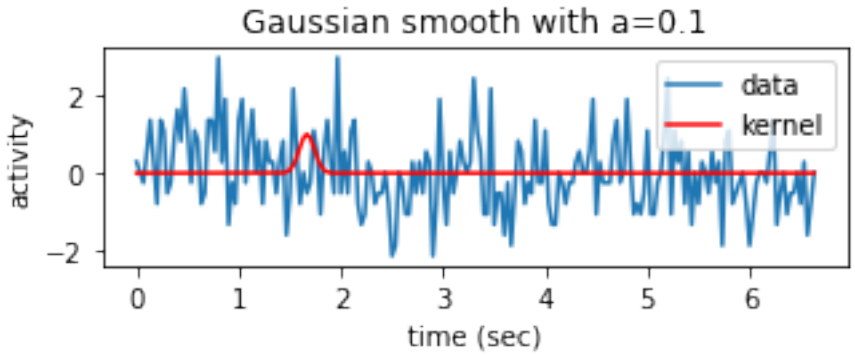
```
In [9]: np.save('data_bin_5', data_bin_5)
```

0.0.2 Gaussian smoothing

```
In [13]: def Gaussian_smooth_trial(trial, a, tau):
             t_list = np.arange(len(trial))
             g = np.exp(-a*(t_list-tau)**2)
             Sg = np.multiply(trial, g)
             return g, Sg
```

In [44]:
```
plt.figure(figsize=(5,1.5))

trial = data[0,:]
time = np.arange(0,81,1/30)[:200]
g, Sg = Gaussian_smooth_trial(trial, 0.1, 50)
plt.plot(time[:200], trial[:200], label='data')
plt.plot(time[:200], g[:200], color='red',label='kernel')
plt.xlabel('time (sec)')
plt.ylabel('activity')
plt.legend()
plt.title('Gaussian smooth with a=0.1')
```

Out[44]: Text(0.5,1,'Gaussian smooth with a=0.1')



In [36]:
```
def Gaussian_smooth(data, a):
    Gau_trial = 0*data
    t_list = np.arange(data.shape[1])

    for i_neuron in range(data.shape[0]):
        for tau in range(data.shape[1]):
            g = np.exp(-a*(t_list-tau)**2)
            Sg = np.multiply(data[i_neuron,:], g)
            Gau_trial[i_neuron, tau] = np.sum(Sg)

    return Gau_trial
```

In [48]:
```
Gau_trial = Gaussian_smooth(data, a=0.05)
```

In [49]:
```
Gau_trial.shape
```

Out[49]: (384, 2430)

In [50]:
```
np.save('Gau_smoothed_005', Gau_trial)
```

03_PCA

March 22, 2019

In [99]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from scipy import stats
from scipy.stats.stats import pearsonr
from mpl_toolkits.mplot3d import Axes3D
```

In [3]:
```
# load original data
data = np.load('data_zscored.npy')
```

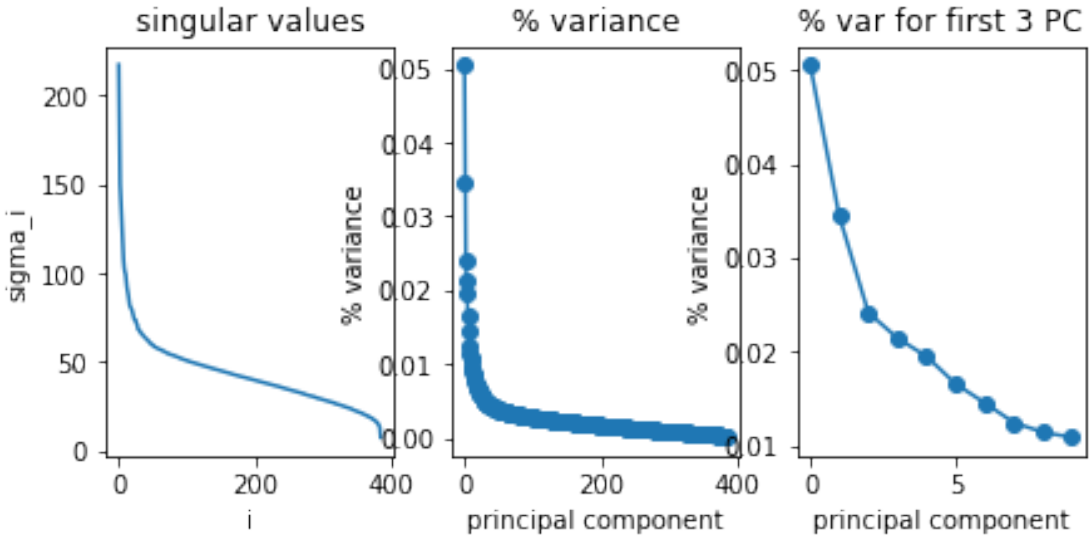
In [4]:
```
data.shape
```

Out[4]:
```
(384, 2430)
```

In [6]:
```
# SVD the data matrix
u, s, vh = np.linalg.svd(data, full_matrices=False)
```

In [46]:
```
def plot_s(s):
    f, (ax1, ax2, ax3) = plt.subplots(1,3, figsize=(7,3))

    ax1.plot(s)
    ax1.set_xlabel('i')
    ax1.set_ylabel('sigma_i')
    ax1.set_title('singular values')

    # get variance
    variance_list = []
    for i in range(len(s)):
        variance_list.append(s[i]**2/np.sum(s**2))

    ax2.plot(variance_list, 'o-')
    ax2.set_xlabel('principal component')
    ax2.set_ylabel('% variance')
    ax2.set_title('% variance')

    ax3.plot(variance_list[:10], 'o-')
    ax3.set_xlabel('principal component')
    ax3.set_ylabel('% variance')
    ax3.set_title('% var for first 3 PC')
```

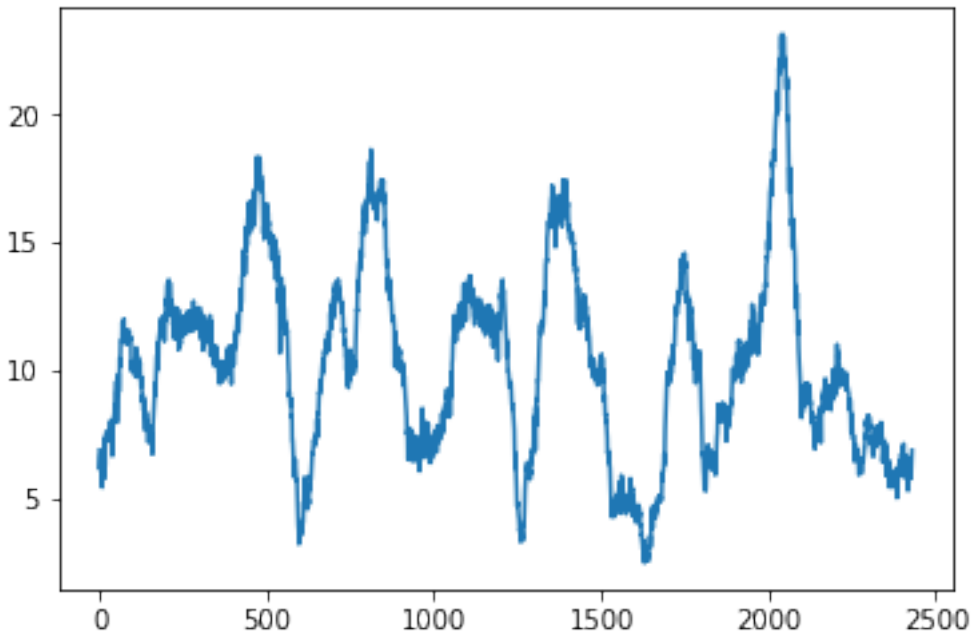
```
In [47]: plot_s(s)
```



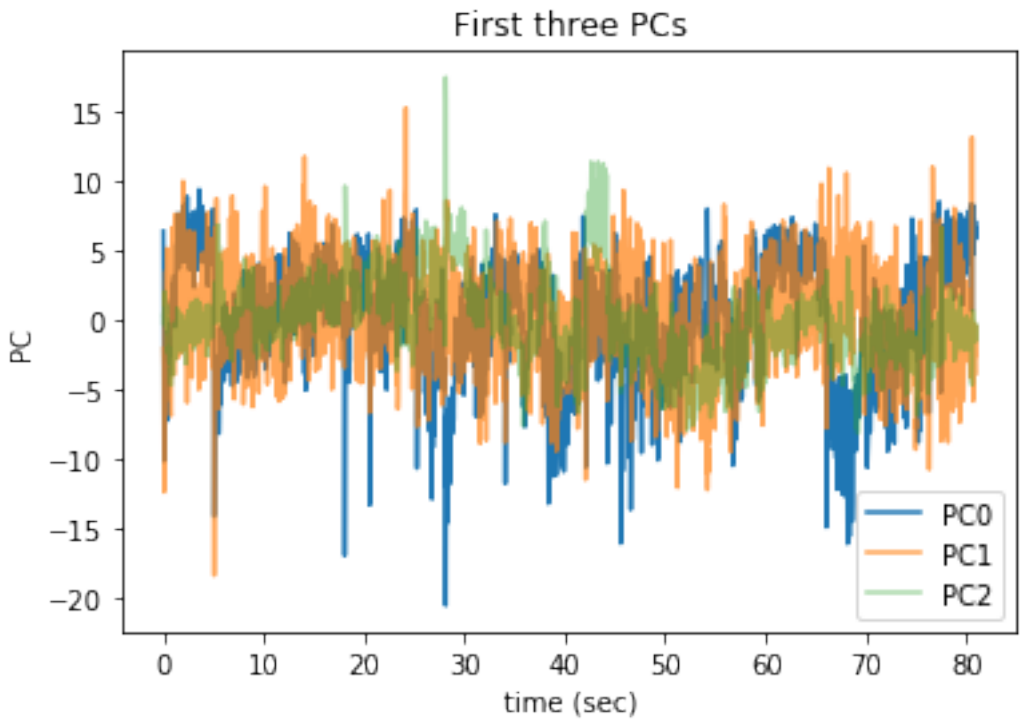

```
In [48]: data.shape

Out[48]: (384, 2430)

In [53]: PC3 = np.zeros((3, data.shape[1]))
         for i in range(3):
             PC3[i,:] = vh[i,:]*s[i]

In [56]: ls

01_PCA_data_z-score.ipynb        data.npy
01.png                           data_zscored.npy
02_PCA_temporal_smoothing.ipynb  finalp2.py
02.png                           finalp.py
03_PCA.ipynb                     Gau_smoothed_001.npy
03.png                           Gau_smoothed_01.npy
04_by_area.ipynb                 key.csv*
04.png                           neuron_delete_ind.npy
0_get_trial.ipynb                runing_speed.npy*
```

```
1_bin_trial.ipynb                trial_10.npy
2_all_data.ipynb                 trial_5.npy
activity.npy*                    trial.npy
area_list_384.npy                try_0_visualize_data.ipynb
data_bin_10.npy                  unused_code .ipynb
data_bin_5.npy
```

```
In [57]: run = np.load('runing_speed.npy')

In [58]: run.shape

Out[58]: (1, 98, 2430)

In [59]: run = run.reshape((98, 2430))

In [66]: num_nan = []
         for i_repeat in range(98):
             n = np.sum(np.isnan(run[i_repeat,:]))
             num_nan.append(n)

In [71]: run_ave = np.nanmean(run, axis=0)

In [72]: run_ave.shape

Out[72]: (2430,)

In [73]: plt.plot(run_ave)

Out[73]: [<matplotlib.lines.Line2D at 0x7f0d2f87af60>]
```



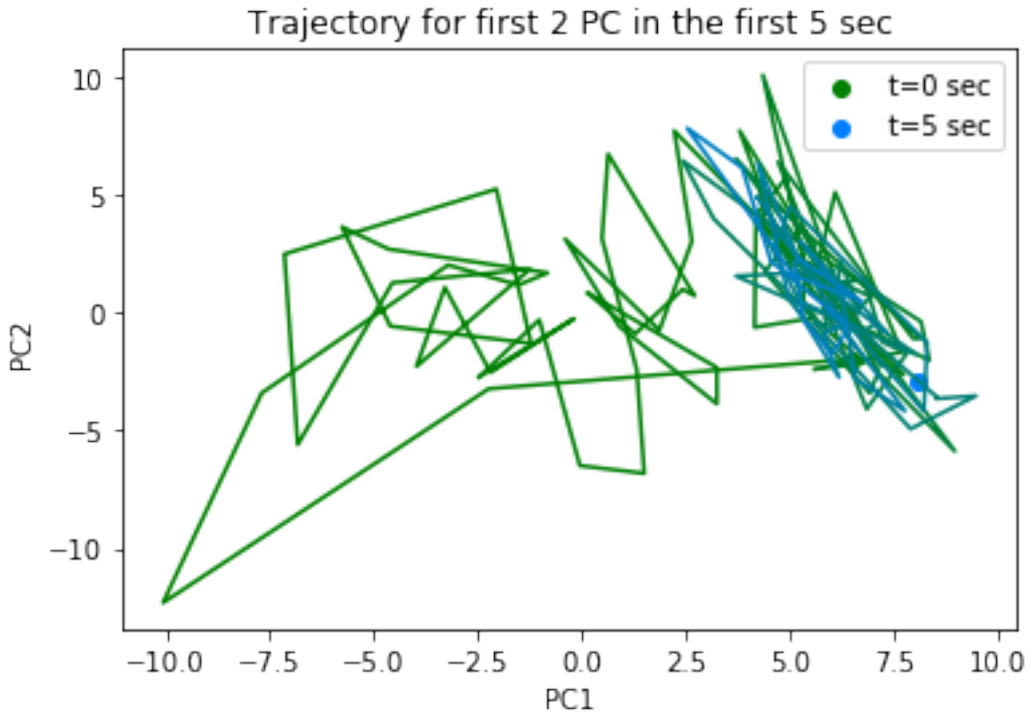

```
In [74]: np.save('run_ave', run_ave)

In [78]: run_zed = stats.zscore(run_ave)

In [79]: np.save('run_zscored', run_zed)

In [75]: PC3.shape

Out[75]: (3, 2430)

In [140]: plt.figure()
          for i in range(3):
              plt.plot(time, PC3[i, :], alpha=1.0-0.3*i, label='PC'+str(i))
          plt.legend()
          plt.xlabel('time (sec)')
          plt.ylabel('PC')
          plt.title('First three PCs')

Out[140]: Text(0.5,1,'First three PCs')
```




```
In [138]: time = np.arange(0,81,1/30)

In [139]: len(time)

Out[139]: 2430

In [130]: t_start = 0
          t_end = 150

          x = PC3[0,t_start:t_end]
          y = PC3[1,t_start:t_end]
          n = len(x)

          # Your colouring array
          T=np.linspace(0,1,np.size(x))**2
          fig = plt.figure()
          ax = fig.add_subplot(111)

          # Segement plot and colour depending on T
          s = 10 # Segment length
          for i in range(0,n-s,s):
              ax.plot(x[i:i+s+1],y[i:i+s+1],color=(0.0,0.5,T[i]))
          ax.scatter(x[0], y[0], label='t=0 sec', color=(0.0,0.5,T[0]))
          ax.scatter(x[-1], y[-1], label='t=5 sec', color=(0.0,0.5,T[-1]))
```

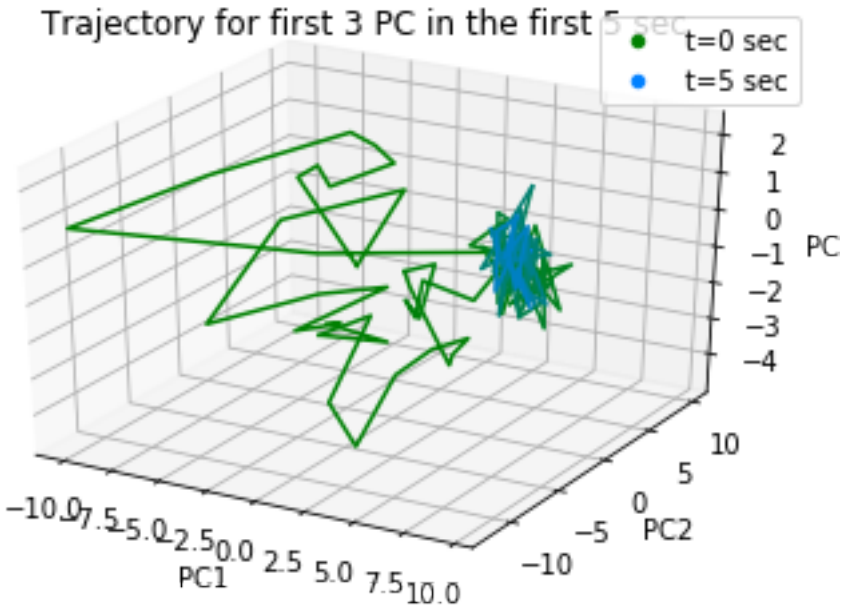
```
plt.title('Trajectory for first 2 PC in the first 5 sec')
plt.xlabel('PC1')
plt.ylabel('PC2')
plt.legend()
```

Out[130]: <matplotlib.legend.Legend at 0x7f0d1793e5c0>



In [134]:
```
t_start = 0
t_end = 150
fig = plt.figure()
ax = fig.gca(projection='3d')

x = PC3[0,t_start:t_end]
T=np.linspace(0,1,np.size(x))**2
s = 10 # Segment length
for i in range(0,n-s,s):
    ax.plot(PC3[0,i:i+s+1], PC3[1,i:i+s+1], PC3[2,i:i+s+1], color=(0.0,0.5,T[i]))
ax.scatter(PC3[0,t_start], PC3[1,t_start], PC3[2,t_start], label='t=0 sec', color=(0
ax.scatter(PC3[0,t_end-t_start-1], PC3[1,t_end-t_start-1], PC3[2,t_end-t_start-1], la
plt.legend()

ax.set_xlabel('PC1')
ax.set_ylabel('PC2')
ax.set_zlabel('PC3')
ax.set_title('Trajectory for first 3 PC in the first 5 sec')
```

Out[134]: Text(0.5,0.92,'Trajectory for first 3 PC in the first 5 sec')



In [94]:
```
for i in range(3):
    cov = pearsonr(run_zed, PC3[i,:])[0]
    print('PC'+str(i)+': '+str(cov))
```

```
PC0: -0.3427161317559347
PC1: 0.27136326267553856
PC2: 0.17318044057513313
```

04_PCA_plot

March 22, 2019

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from scipy import stats
from scipy.stats.stats import pearsonr
from mpl_toolkits.mplot3d import Axes3D
```

In [6]:
```
def plot_s(s):
    f, (ax1, ax2, ax3) = plt.subplots(1,3, figsize=(7,3))

    ax1.plot(s)
    ax1.set_xlabel('i')
    ax1.set_ylabel('sigma_i')
    ax1.set_title('singular values')

    # get variance
    variance_list = []
    for i in range(len(s)):
        variance_list.append(s[i]**2/np.sum(s**2))

    ax2.plot(variance_list, 'o-')
    ax2.set_xlabel('principal component')
    ax2.set_ylabel('% variance')
    ax2.set_title('% variance')

    ax3.plot(variance_list[:10], 'o-')
    ax3.set_xlabel('principal component')
    ax3.set_ylabel('% variance')
    ax3.set_title('% var for first 10 PC')
```

In [7]:
```
# load original data
data = np.load('data_zscored.npy')
```

In [8]:
```
def data_plot(data):
    # SVD the data matrix
    u, s, vh = np.linalg.svd(data, full_matrices=False)

    plot_s(s)
```

```
            PC3 = np.zeros((3, data.shape[1]))
            for i in range(3):
                PC3[i,:] = vh[i,:]*s[i]

            return PC3
```

```
In [42]: def plot_PC3(PC3):
             plt.figure()
             l = PC3.shape[1]
             time = np.arange(0,81,81/l)
             for i in range(3):
                 plt.plot(time, PC3[i, :], alpha=1.0-0.3*i, label='PC'+str(i))
             plt.legend()
             plt.xlabel('time (sec)')
             plt.ylabel('PC')
             plt.title('First three PCs')
```

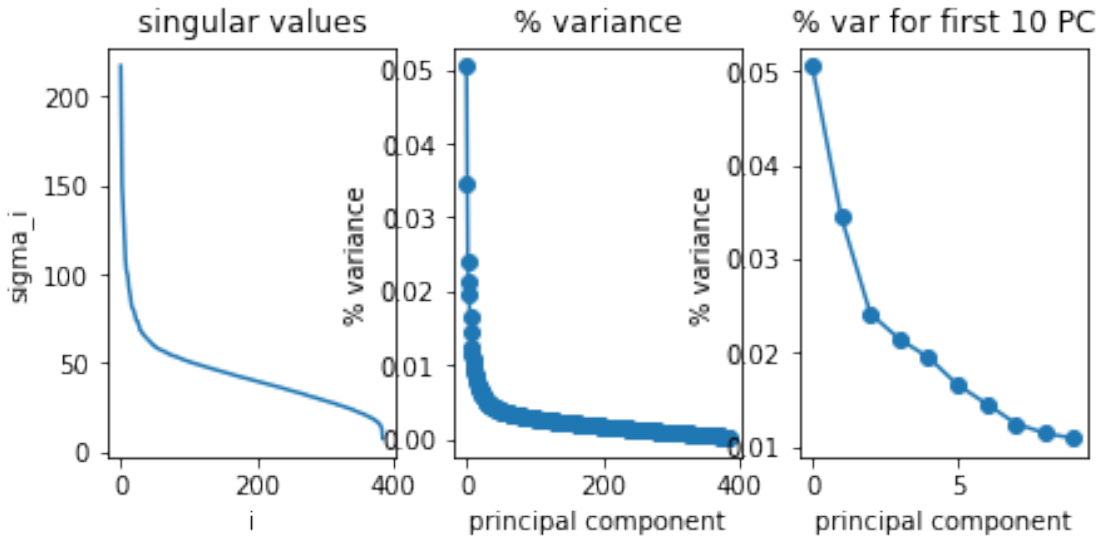
```
In [81]: def plot_PC_1(PC3, t_end=150):
             t_start = 0

             x = PC3[0,t_start:t_end]
             y = PC3[1,t_start:t_end]
             n = len(x)

             # Your colouring array
             T=np.linspace(0,1,np.size(x))**2
             fig = plt.figure()
             ax = fig.add_subplot(111)

             # Segement plot and colour depending on T
             s = 10 # Segment length
             for i in range(0,n,s):
                 ax.plot(x[i:i+s+1],y[i:i+s+1],color=(0.0,0.5,T[i]))
             ax.scatter(x[0], y[0], label='t=0 sec', color=(0.0,0.5,T[0]))
             ax.scatter(x[-1], y[-1], label='t=5 sec', color=(0.0,0.5,T[-1]))
             plt.title('Trajectory for first 2 PC in the first 5 sec')
             plt.xlabel('PC1')
             plt.ylabel('PC2')
             plt.legend()
```

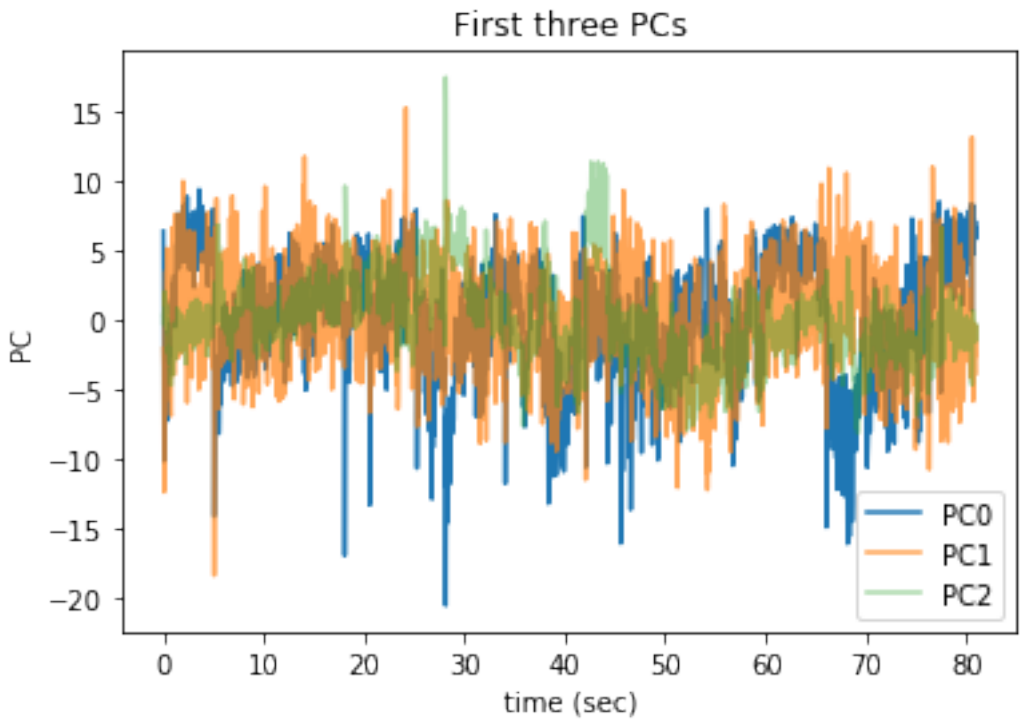
```
In [82]: def plot_PC_02(PC3, t_end=150):
             t_start = 0
             fig = plt.figure()
             ax = fig.gca(projection='3d')

             x = PC3[0,t_start:t_end]
             n = len(x)
```

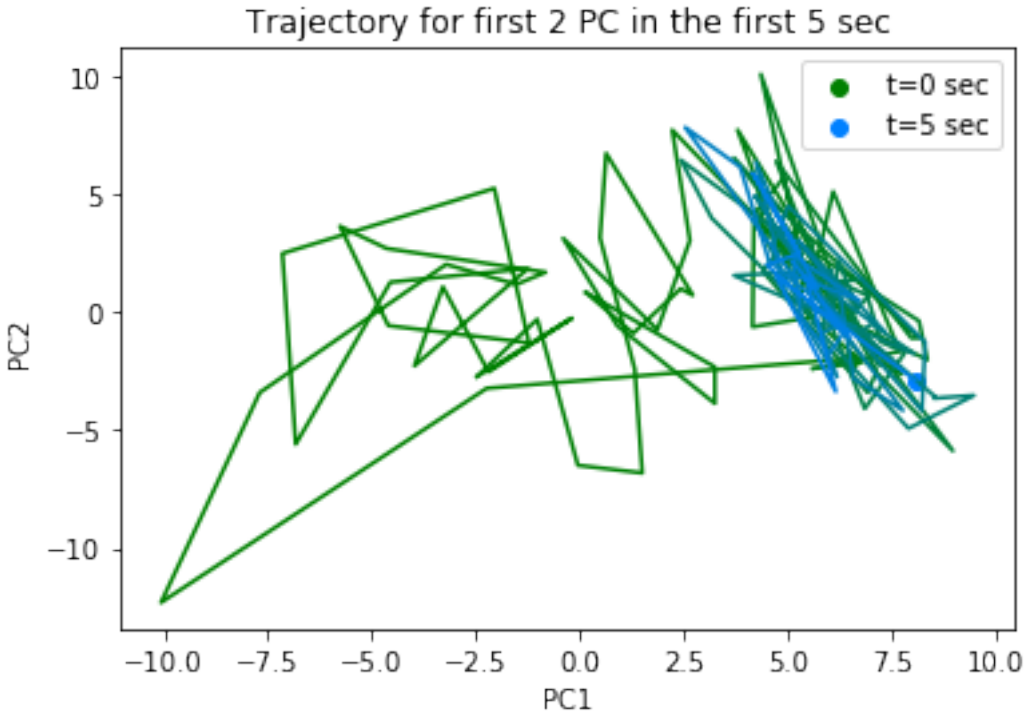
```
            T=np.linspace(0,1,np.size(x))**2
            s = 10 # Segment length
            for i in range(0,n,s):
                ax.plot(PC3[0,i:i+s+1], PC3[1,i:i+s+1], PC3[2,i:i+s+1], color=(0.0,0.5,T[i]))
            ax.scatter(PC3[0,t_start], PC3[1,t_start], PC3[2,t_start], label='t=0 sec', color=
            ax.scatter(PC3[0,t_end-t_start-1], PC3[1,t_end-t_start-1], PC3[2,t_end-t_start-1]
            plt.legend()

            ax.set_xlabel('PC1')
            ax.set_ylabel('PC2')
            ax.set_zlabel('PC3')
            ax.set_title('Trajectory for first 3 PC in the first 5 sec')

In [83]: def plot_all_PC(data_file, t_end, run_zed = run):
            data = np.load(data_file)
            PC3 = data_plot(data)
            plot_PC3(PC3)
            plot_PC_1(PC3, t_end)
            plot_PC_02(PC3, t_end)

            run_zed = run_zed.reshape((len(PC3[0,:]), -1))
            run_zed = np.mean(run_zed, 1)
            for i in range(3):
                cov = pearsonr(run_zed, PC3[i,:])[0]
                print('PC'+str(i)+': '+str(cov))

In [84]: run = np.load('run_zscored.npy')

In [85]: ls

01_PCA_data_z-score.ipynb       11.png            finalp2.py

01.png                          12.png            finalp.py

02_PCA_temporal_smoothing.ipynb 13.png            Gau_smoothed_001.npy

02.png                          14.png            Gau_smoothed_01.npy

03_PCA.ipynb                    15.png            key.csv*

03.png                          16.png            neuron_delete_ind.npy

04_by_area.ipynb                17.png            run_ave.npy

04.png                          18.png            runing_speed.npy*

05_PCA_plot.ipynb               1_bin_trial.ipynb run_zscored.npy

05.png                          2_all_data.ipynb  trial_10.npy
```

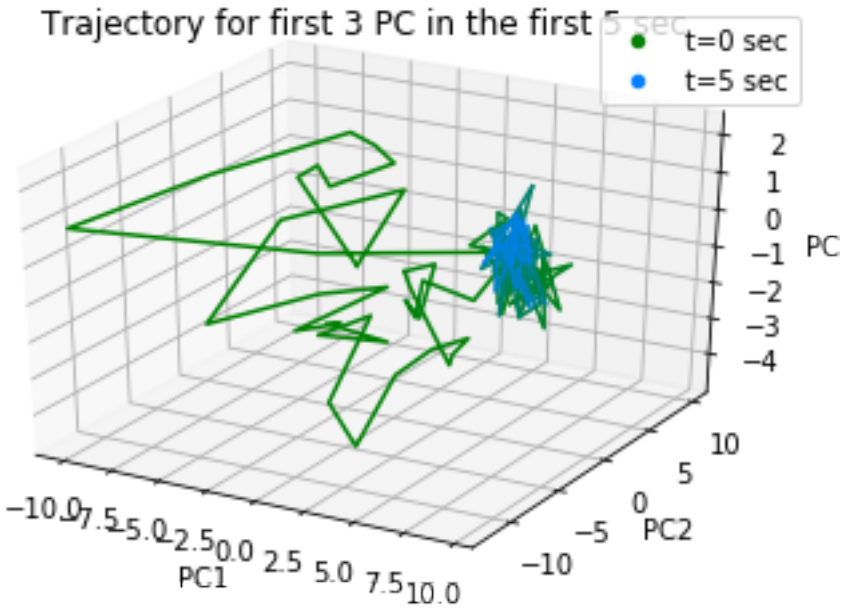
```
06.png                          activity.npy*       trial_5.npy

07.png                          area_list_384.npy   trial.npy

08.png                          data_bin_10.npy     try_0_visualize_data.ipynb

09.png                          data_bin_5.npy      unused_code .ipynb

0_get_trial.ipynb               data.npy

10.png                          data_zscored.npy
```

```
In [86]: plot_all_PC('data_zscored.npy', t_end=150)

PC0: -0.3427161317559347
PC1: 0.27136326267553856
PC2: 0.17318044057513313
```

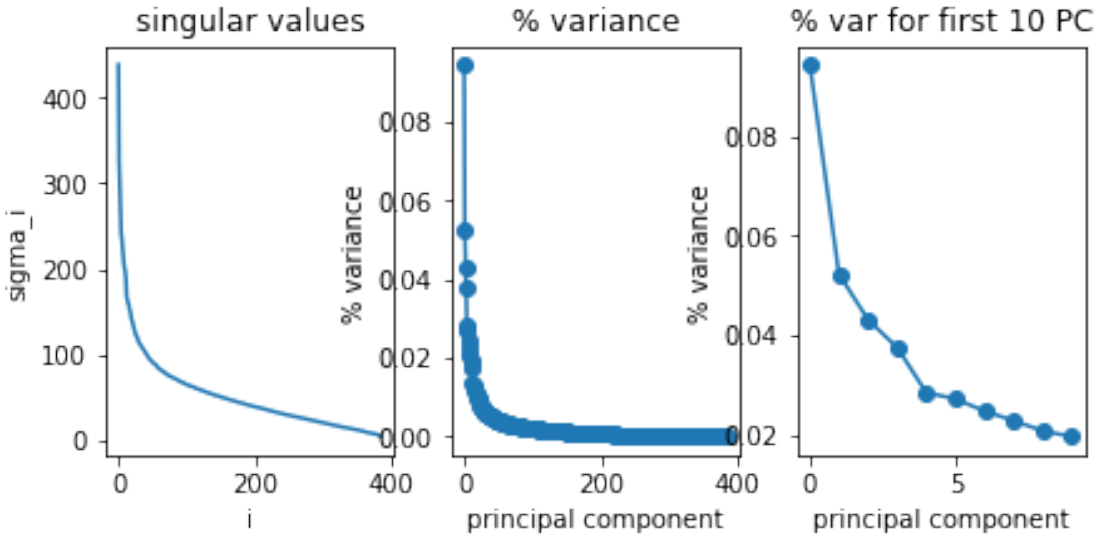


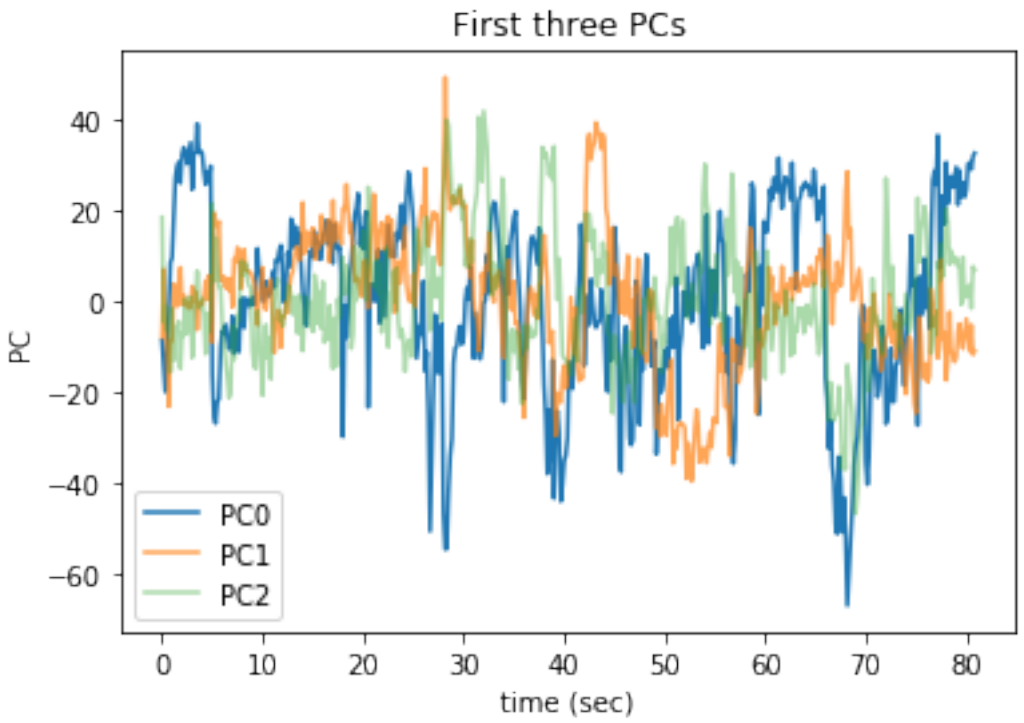

First three PCs
PC
time (sec)
PC0
PC1
PC2

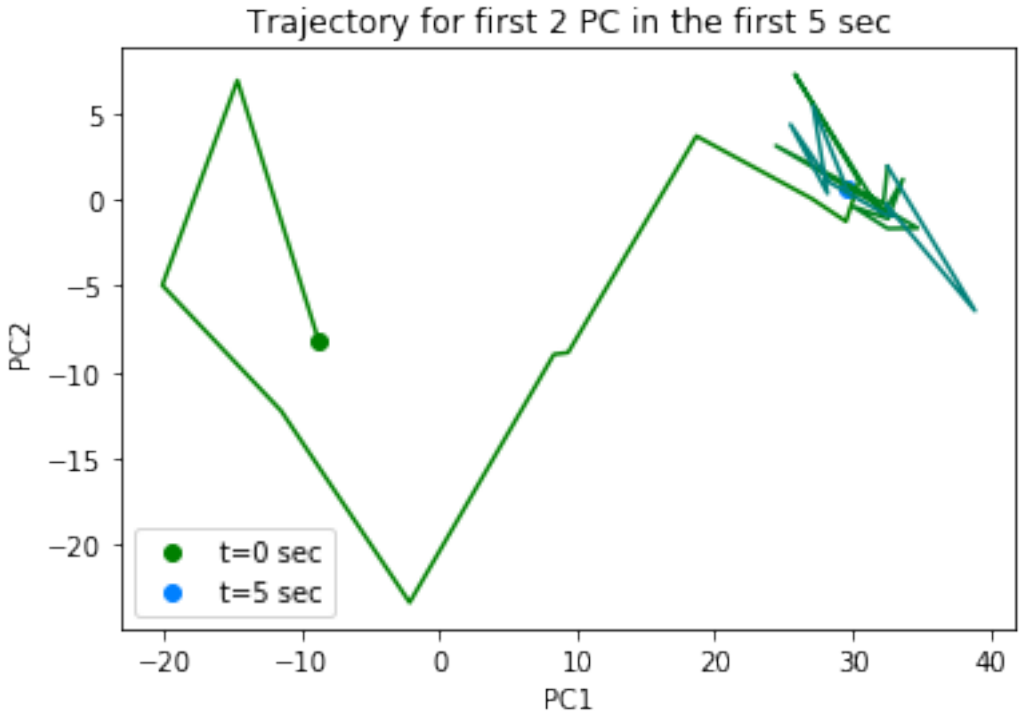

Trajectory for first 2 PC in the first 5 sec
PC2
PC1
t=0 sec
t=5 sec

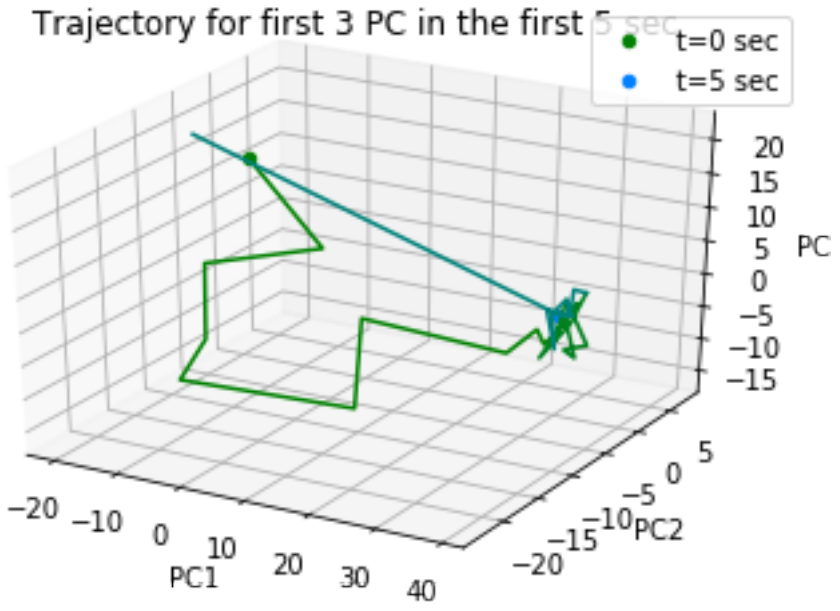



```
In [87]: plot_all_PC('data_bin_5.npy', t_end=30)

PC0: -0.38929219848841035
PC1: 0.44937223005373284
PC2: -0.3833019286776914
```

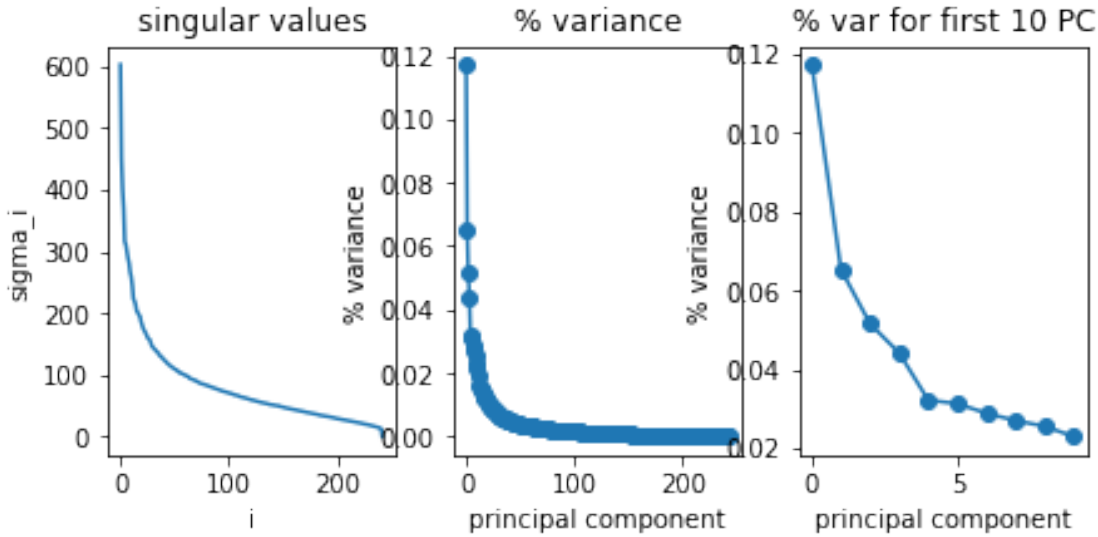


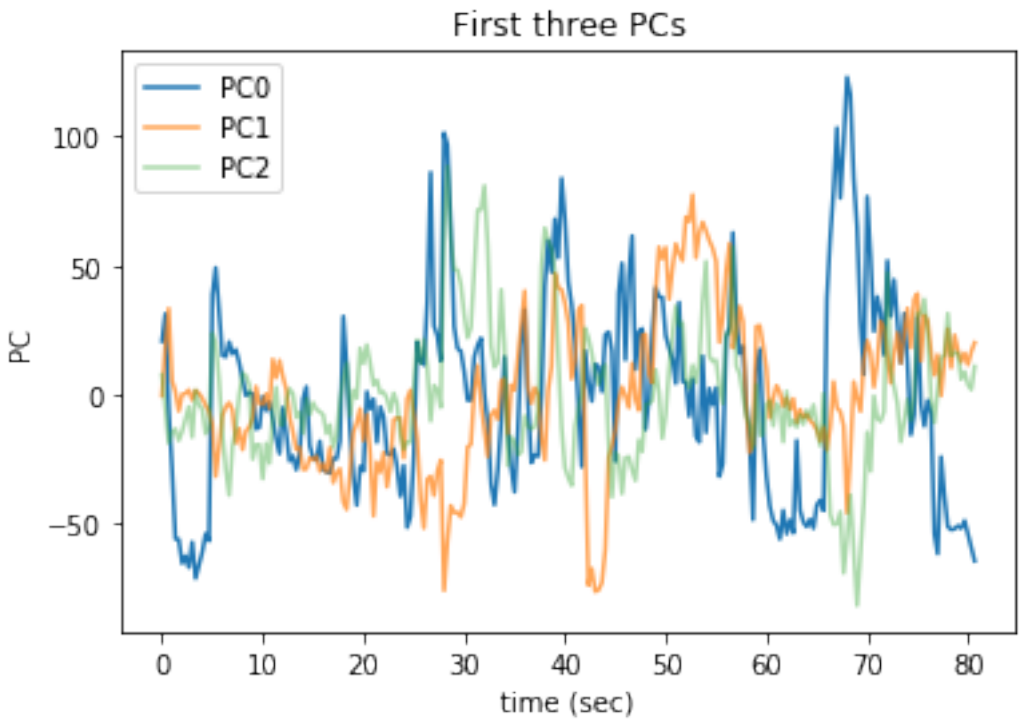

First three PCs
PC
time (sec)
PC0
PC1
PC2
40
20
0
−20
−40
−60
0
10
20
30
40
50
60
70
80

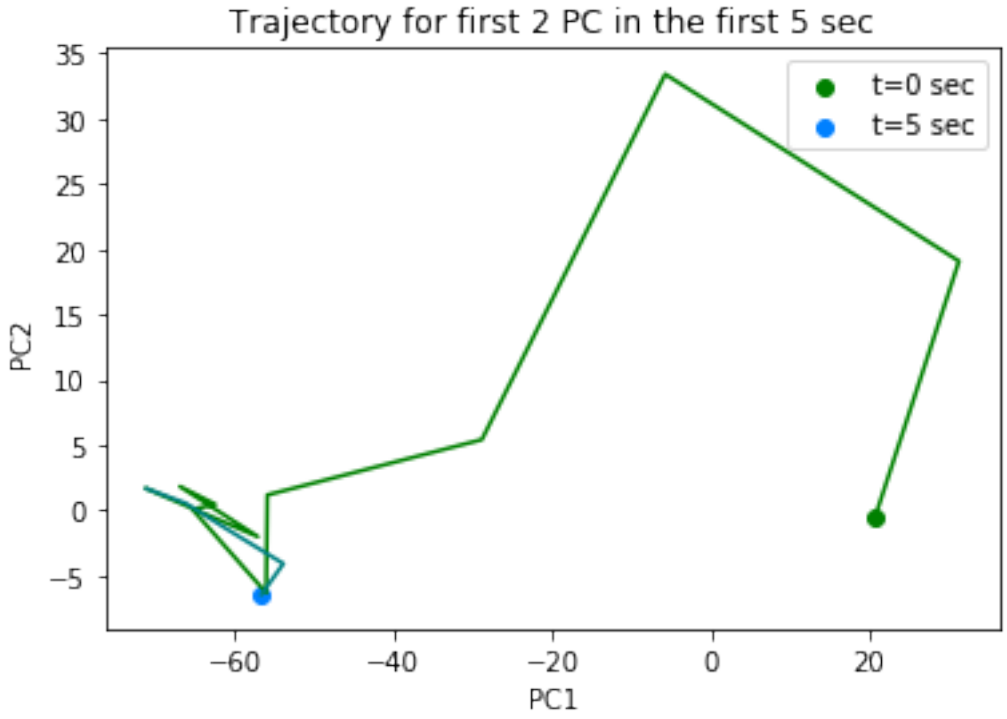

Trajectory for first 2 PC in the first 5 sec
PC2
PC1
t=0 sec
t=5 sec
5
0
−5
−10
−15
−20
−20
−10
0
10
20
30
40

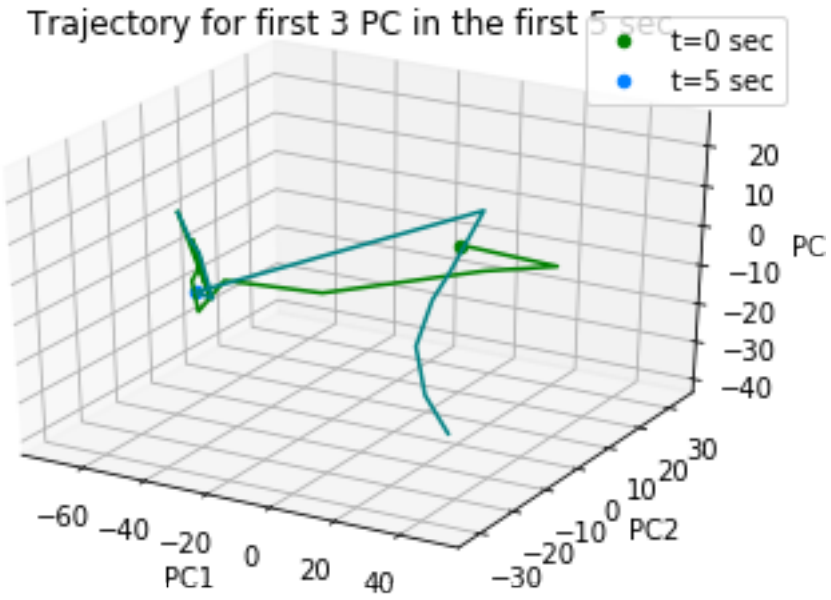



```
In [88]: plot_all_PC('data_bin_10.npy', t_end=15)

PC0: 0.41128599440265323
PC1: -0.43547007559009815
PC2: -0.42824477222493895
```

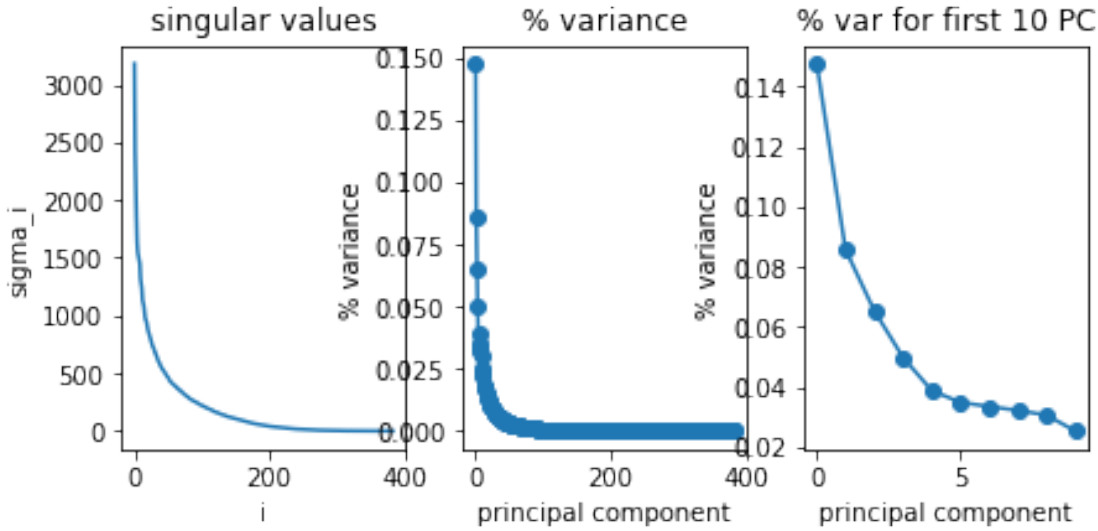


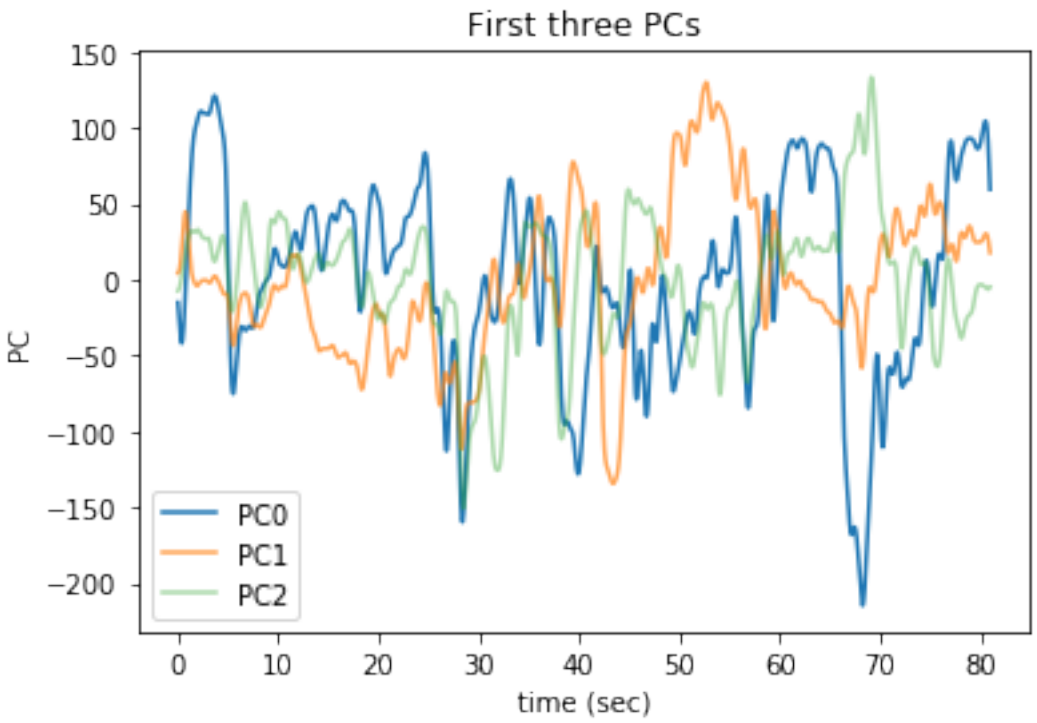

First three PCs
PC0
PC1
PC2
100
50
0
−50
PC
0
10
20
30
40
50
60
70
80
time (sec)

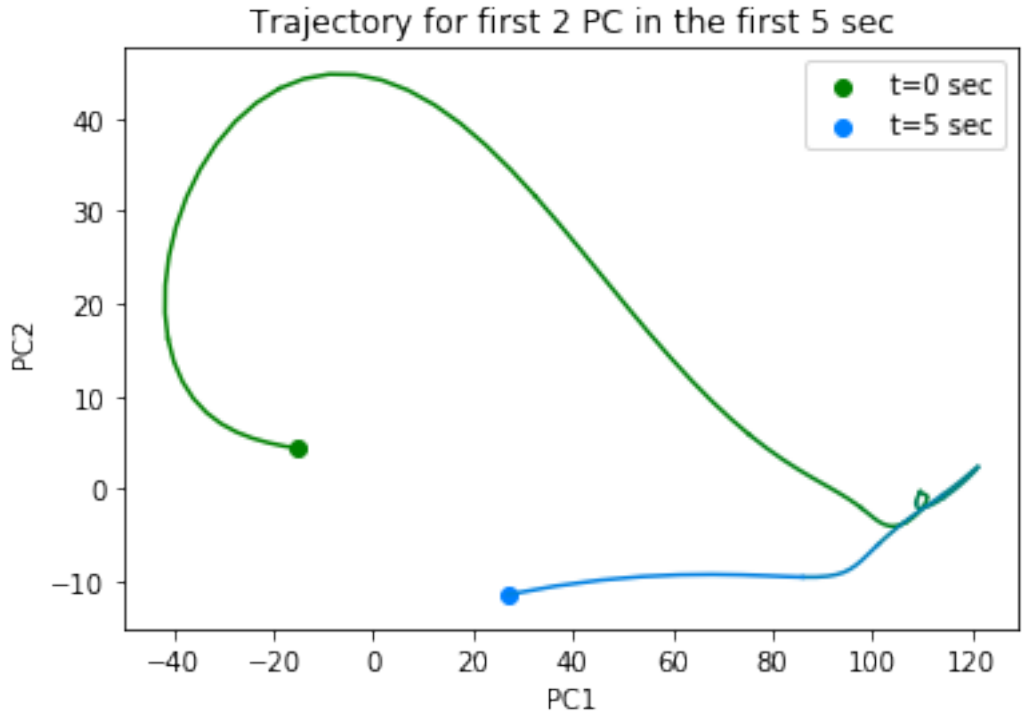

Trajectory for first 2 PC in the first 5 sec
t=0 sec
t=5 sec
35
30
25
20
15
10
5
0
−5
PC2
−60
−40
−20
0
20
PC1

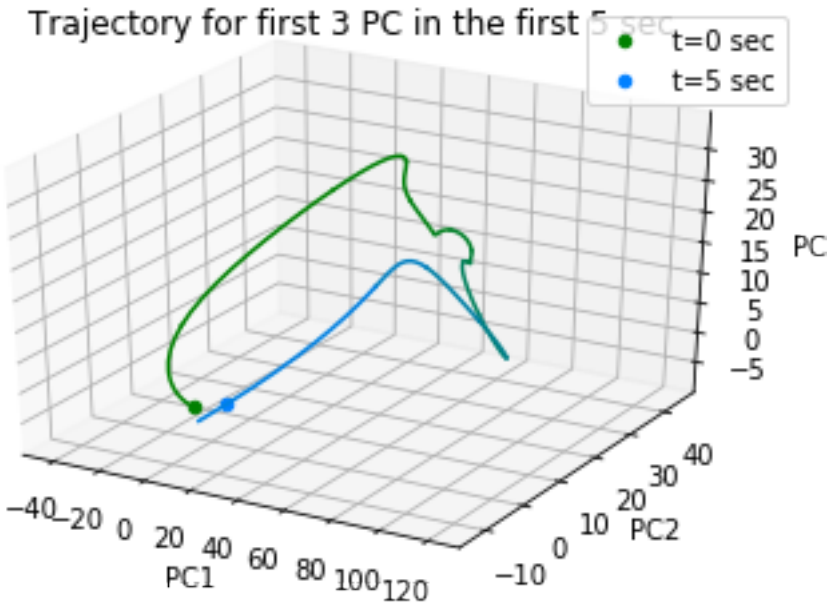



```
In [90]: plot_all_PC('Gau_smoothed_001.npy', t_end=150)

PC0: -0.436870635776971
PC1: -0.4594772858133892
PC2: 0.4202007392641546
```

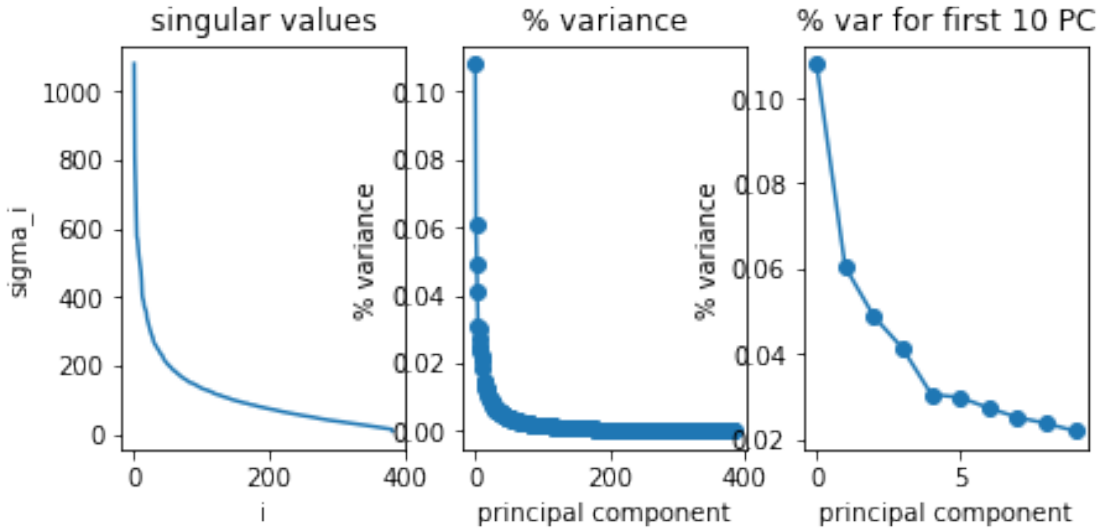


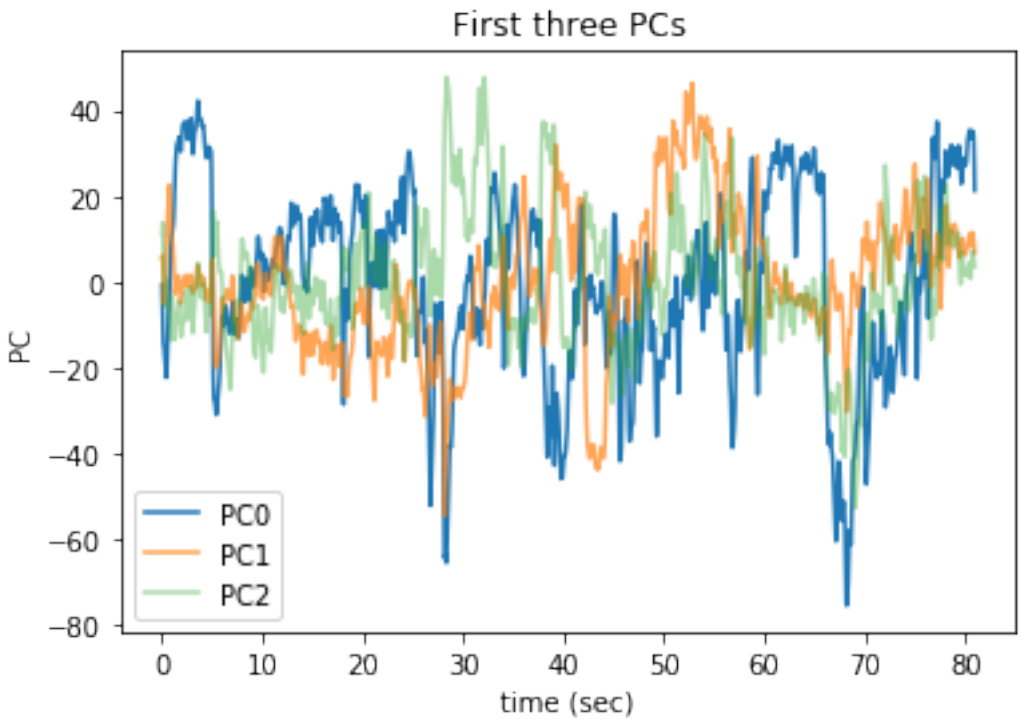

First three PCs
PC
time (sec)
PC0
PC1
PC2
150
100
50
0
−50
−100
−150
−200
0
10
20
30
40
50
60
70
80

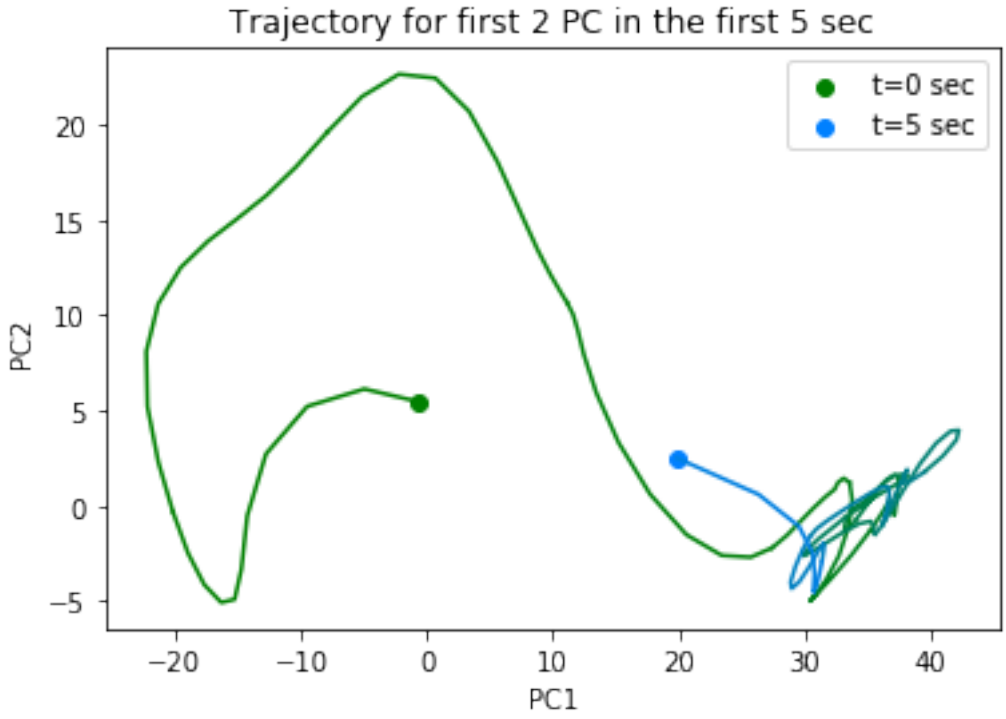

Trajectory for first 2 PC in the first 5 sec
PC2
PC1
t=0 sec
t=5 sec
40
30
20
10
0
−10
−40
−20
0
20
40
60
80
100
120

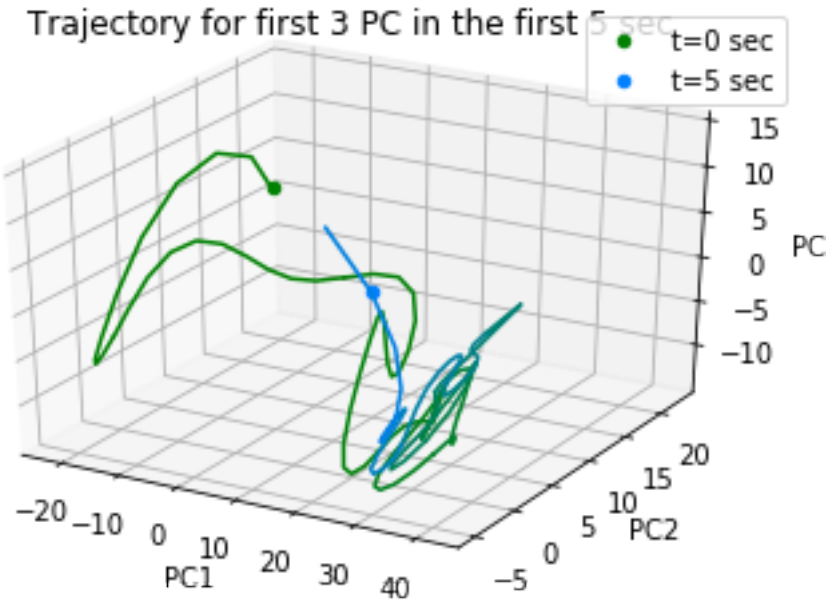



```
In [93]: plot_all_PC('Gau_smoothed_01.npy', t_end=150)

PC0: -0.3980136069639942
PC1: -0.43525027820214685
PC2: -0.41799301497502434
```

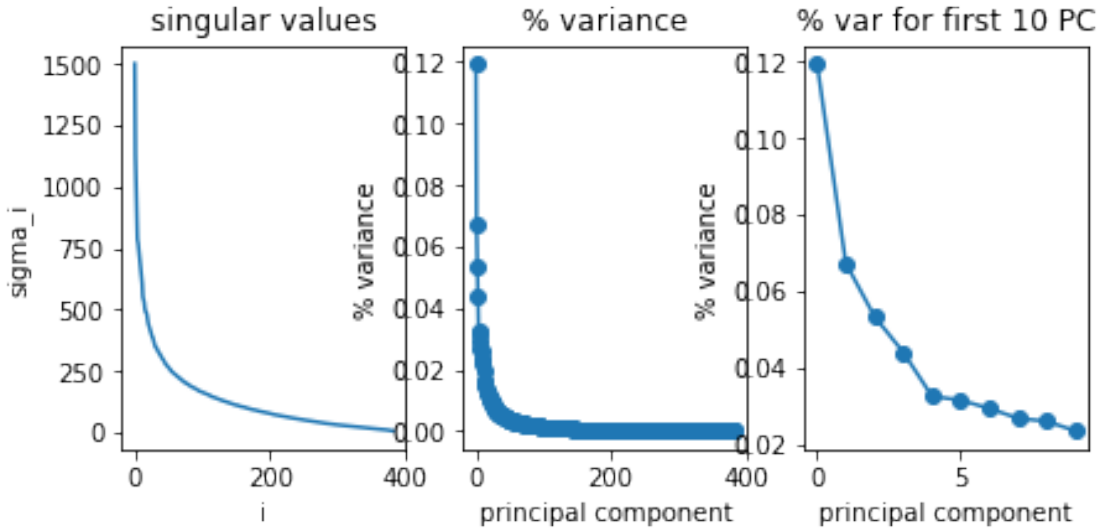


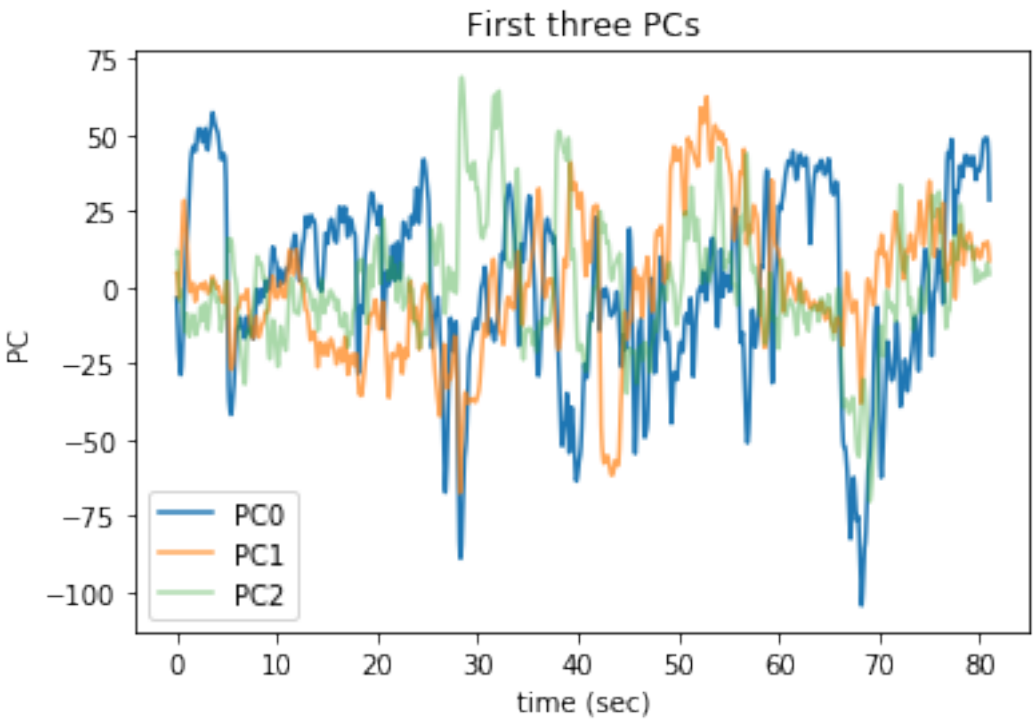

First three PCs
PC
time (sec)
PC0
PC1
PC2
40
20
0
−20
−40
−60
−80
0
10
20
30
40
50
60
70
80

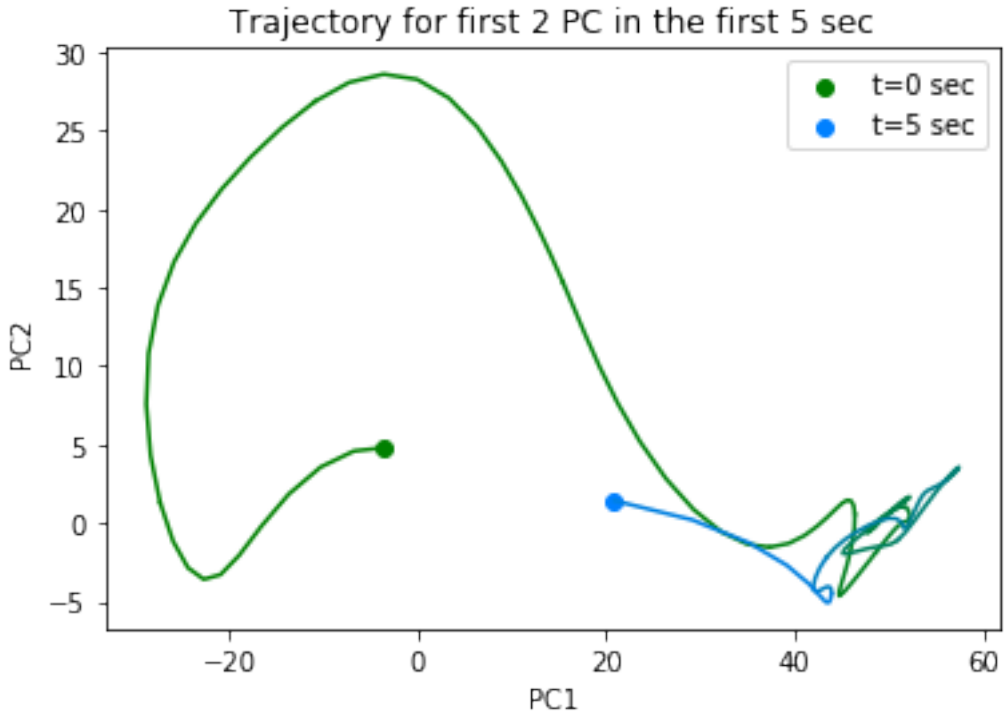

Trajectory for first 2 PC in the first 5 sec
PC2
PC1
t=0 sec
t=5 sec
20
15
10
5
0
−5
−20
−10
0
10
20
30
40

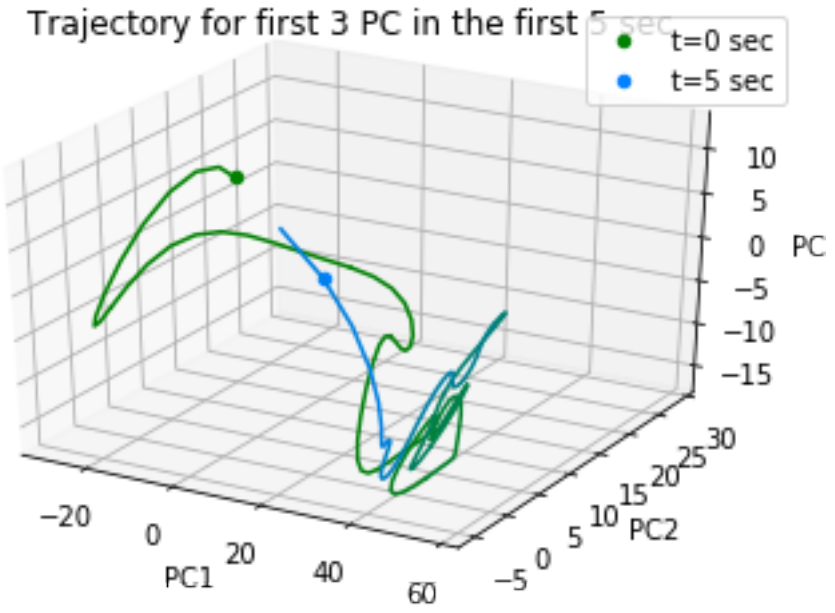



```
In [94]: plot_all_PC('Gau_smoothed_005.npy', t_end=150)

PC0: -0.4074086929579095
PC1: -0.4350121633013316
PC2: -0.4320325460706861
```




First three PCs
PC
time (sec)
PC0
PC1
PC2
75
50
25
0
−25
−50
−75
−100
0
10
20
30
40
50
60
70
80


Trajectory for first 2 PC in the first 5 sec
PC2
PC1
t=0 sec
t=5 sec
30
25
20
15
10
5
0
−5
−20
0
20
40
60


Trajectory for first 3 PC in the first 5 sec
t=0 sec
t=5 sec
PC1
PC2
PC
−20
0
20
40
60
−5
0
5
10
15
20
25
30
10
5
0
−5
−10
−15

05_DMD

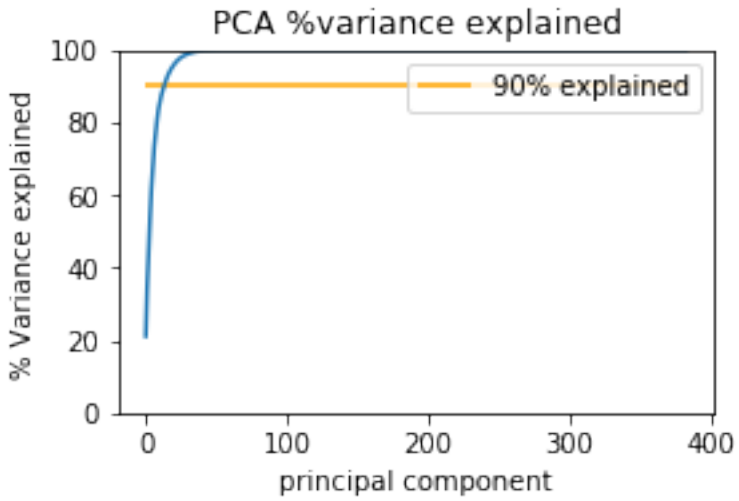
March 22, 2019

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
        from scipy import stats
        from scipy.stats.stats import pearsonr
        from mpl_toolkits.mplot3d import Axes3D

In [119]: data = np.load('Gau_smoothed_001.npy')

In [120]: data.shape

Out[120]: (384, 2430)

In [121]: # Due to DMD, only choose the first 500 time points
          data = data[:,:500]

In [122]: data.shape

Out[122]: (384, 500)

In [123]: # select X1 and X2
          X1 = data[:,:-1]
          X2 = data[:,1:]
          print(X1.shape)

(384, 499)


In [124]: # perform reduced SVD on X1
          U, s, V = np.linalg.svd(X1, full_matrices=False)

In [125]: def get_l(s):
              variance_list = []
              acc_var_list = []
              for i in range(len(s)):
                  variance_list.append(s[i]**2/np.sum(s**2))
                  acc_var_list.append(100*np.sum(variance_list))

              plt.figure(figsize=(4,2.5))
```

```
    plt.plot(acc_var_list)
    plt.xlabel('principal component')
    plt.ylabel('% Variance explained')
    plt.title('PCA %variance explained')
    plt.ylim([0,100])
    plt.hlines(90, 0, len(variance_list), color='orange', label='90% explained')
    plt.legend()

    l = next(x[0] for x in enumerate(acc_var_list) if x[1]>=90)
    return l
```

In [126]:
```
l = get_l(s)
print(l)
```

```
13
```



In [127]:
```
print(U.shape)
print(s.shape)
print(V.shape)
```

```
(384, 384)
(384,)
(384, 499)
```

In [128]:
```
# take the low-rank approximation
U = U[:,:l]
s = s[:l]
V = V[:l, :]
```

In [129]:
```
# compute S'
V = V.conj().T
S_bar = U.T.conj().dot(X2).dot(V) * np.reciprocal(s)
```

In [130]:
```
# Find eigenvalues and eigenvectors of S_bar
w,v = np.linalg.eig(S_bar)
```

In [131]:
```
w.shape
```

Out[131]: (13,)

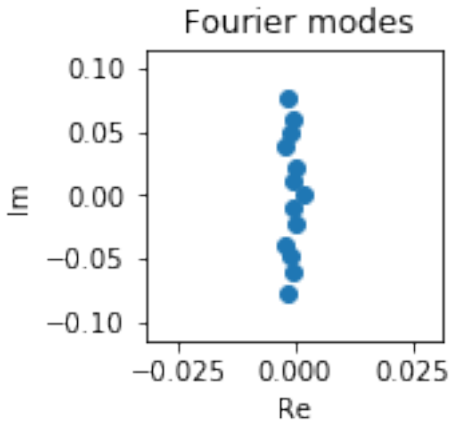
In [132]:
```
# Find eigenvalues and eigenvector for A
A_vec = U.dot(v)
A_eigs = w
```

In [133]:
```
# calculate w, the Fourier mode
# in this case, dt=1
w_Fourier = np.log(A_eigs)
```

In [134]:
```
# compute the amplitude coefficients b
b, res, r, sing = np.linalg.lstsq(A_vec, data.T[0], rcond=None)
```

In [135]:
```
def plot_w(w_Fourier):
    plt.figure(figsize=(2,2))
    plt.scatter([x.real for x in w_Fourier], [x.imag for x in w_Fourier])
    plt.xlabel('Re')
    plt.ylabel('Im')
    plt.title(' Fourier modes')
```

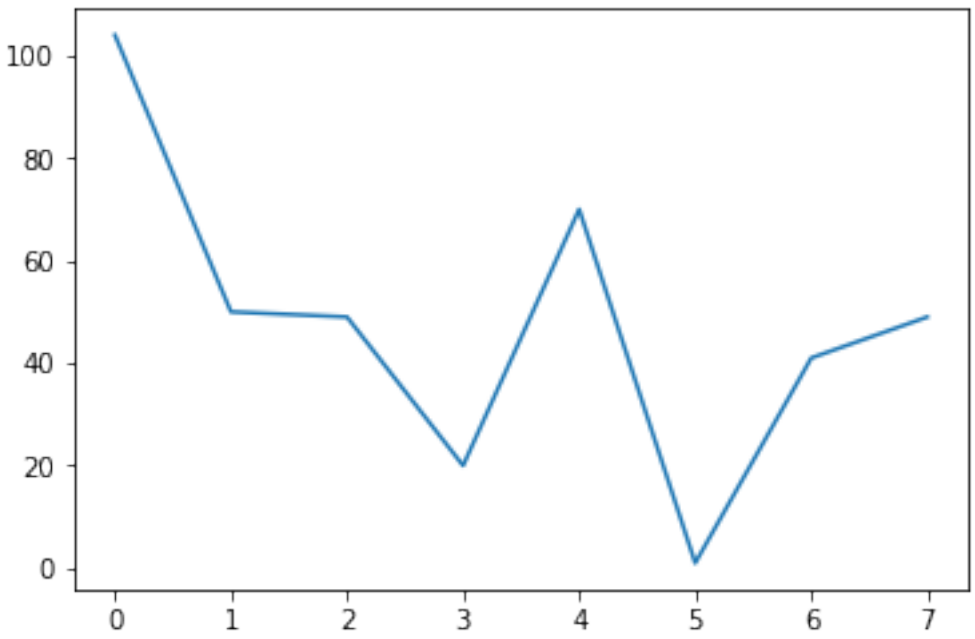
In [136]:
```
plot_w(w_Fourier)
```



In [137]:
```
# choose w_j with larger norms
w_norm = np.abs(np.imag(w_Fourier))
w_ind = np.argmax(w_norm)
```

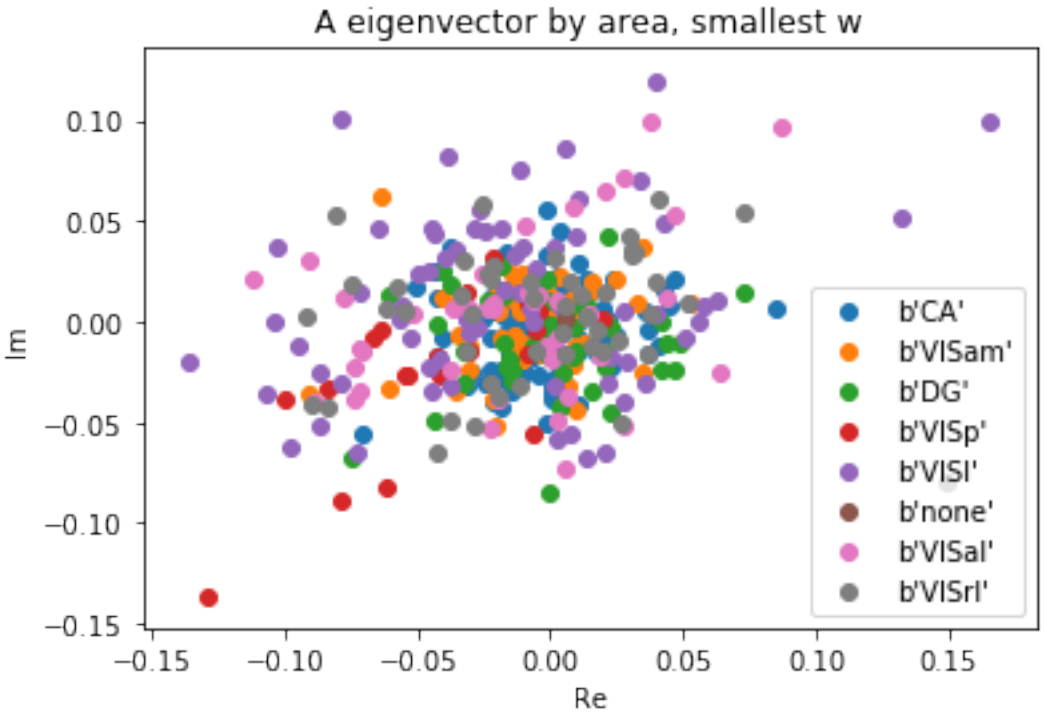
```
In [138]: w_ind_list = []
          for i in range(3):
              w_ind = np.argmax(w_norm)
              w_ind_list.append(w_ind)
              w_norm[w_ind] = 0

In [139]: w_ind_list

Out[139]: [0, 1, 2]

In [140]: A_vec.shape

Out[140]: (384, 13)

In [141]: data.shape

Out[141]: (384, 500)

In [142]: t_list = np.arange(data.shape[1])

In [143]: back_list = np.zeros((data.shape), dtype=complex)
          for i_t, t in enumerate(t_list):
              for w_ind in w_ind_list:
                  back = b[w_ind]*A_vec[:,w_ind]*np.exp(w_Fourier[w_ind]*t)
                  back_list[:,i_t] += back

In [149]: len(A_vec[:,0])

Out[149]: 384

In [162]: area_list = np.load('area_list_384.npy')

In [165]: area_list_unique = []
          for area in area_list:
              if area not in area_list_unique:
                  area_list_unique.append(area)

In [166]: len(area_list_unique)

Out[166]: 8

In [167]: area_ind_list = []
          for key in area_list_unique:
              area_ind= [i for i, area in enumerate(area_list) if area==key]
              area_ind_list.append(area_ind)

In [186]: area_list_unique
```

```
Out[186]: ["b'CA'",
           "b'VISam'",
           "b'DG'",
           "b'VISp'",
           "b'VISl'",
           "b'none'",
           "b'VISal'",
           "b'VISrl'"]

In [187]: np.save('area_list_unique', area_list_unique)

In [183]: area_ind_len

Out[183]: [104, 50, 49, 20, 70, 1, 41, 49]

In [188]: np.save('area_ind_list', area_ind_list)

In [169]: len(area_ind_list)

Out[169]: 8

In [181]: area_ind_len = []
          for i in range(8):
              area_ind_len.append(len(area_ind_list[i]))

In [182]: plt.plot(area_ind_len)

Out[182]: [<matplotlib.lines.Line2D at 0x7f62de610278>]
```



In [180]:
```
A_ind=np.argmin(w_Fourier)
plt.figure()
for i in range(8):
    key = area_list_unique[i]
    key_ind = area_ind_list[i]
    plt.scatter(np.real(A_vec[key_ind,A_ind]), np.imag(A_vec[key_ind,A_ind]), label=
plt.legend()
plt.xlabel('Re')
plt.ylabel('Im')
plt.title('A eigenvector by area, smallest w')
```

Out[180]:
```
Text(0.5,1,'A eigenvector by area, smallest w')
```

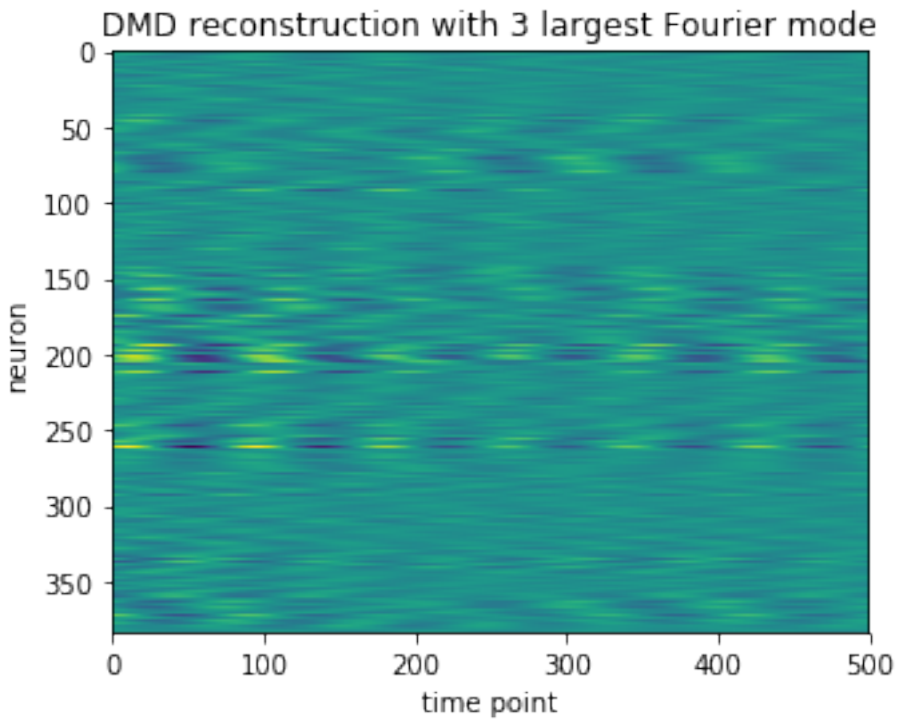


In [144]:
```
back_list = np.asarray(back_list)
back_list.shape
```

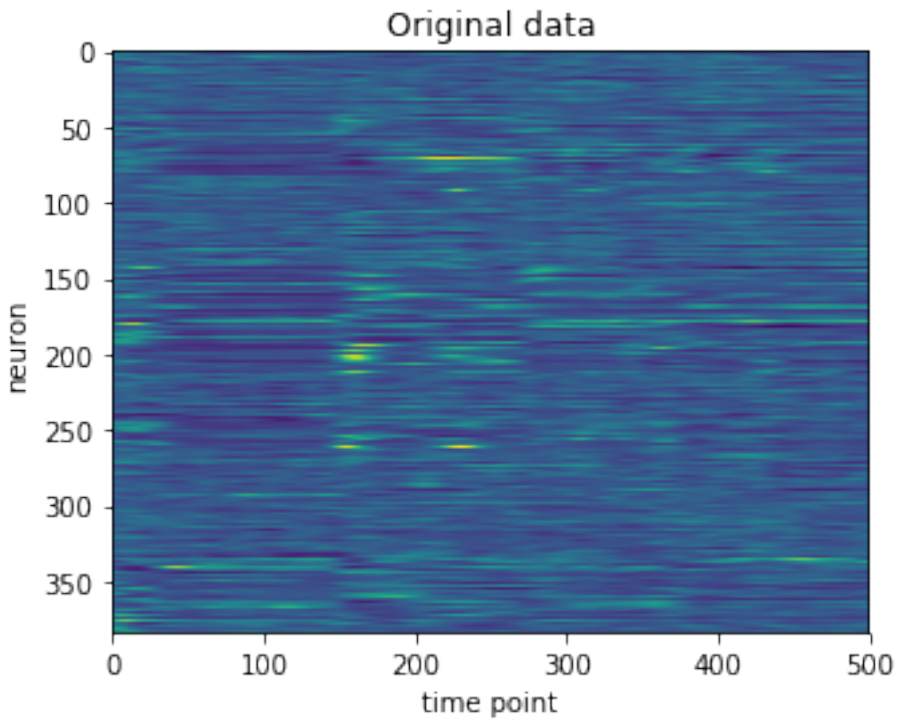
Out[144]:
```
(384, 500)
```

In [158]:
```
plt.imshow(np.real(back_list))
plt.xlabel('time point')
plt.ylabel('neuron')
plt.title('DMD reconstruction with 3 largest Fourier mode')
```

Out[158]:
```
Text(0.5,1,'DMD reconstruction with 3 largest Fourier mode')
```




```
In [159]: plt.imshow(data)
          plt.xlabel('time point')
          plt.ylabel('neuron')
          plt.title('Original data')

Out[159]: Text(0.5,1,'Original data')
```


Original data
0
50
100
150
200
250
300
350
neuron
0
100
200
300
400
500
time point