CUED - Engineering Tripos Part IIB 2023-2024 **Module Coursework**

Module	4G10	Title of report	Coursework for Brain-Machine Interfaces Assignment 1
Date submitted: 18th November 2023		Assessment for this module is ☒ 100% / ☐ 25% coursework of which this assignment forms 50 %	

UNDERGRADUATE and POST GRADUATE STUDENTS		
Candidate number:	5704G	☒ Undergraduate ☐ Post graduate

Feedback to the student ☐ **See also comments in the text**		Very good	**Good**	Needs improvmt
CONTENT	**Completeness, quantity of content:** Has the report covered all aspects of the lab? Has the analysis been carried out thoroughly?			
	Correctness, quality of content Is the data correct? Is the analysis of the data correct? Are the conclusions correct?			
	Depth of understanding, quality of discussion Does the report show a good technical understanding? Have all the relevant conclusions been drawn?			
	Comments:			
PRESENTATION	**Attention to detail, typesetting and typographical errors** Is the report free of typographical errors? Are the figures/tables/references presented professionally?			
	Comments:			

Marker: Date:

Brain-Machine Interface Assignment 1

I. Introduction

We investigate applying a dynamical dimensionality reduction method to uncover rotational dynamics in the motor cortex. Theory suggests that the network in the motor cortex can autonomously generate dynamics and activity patterns, which causes body movements. These autonomous dynamical systems have the property that starting from the same state, the instantaneous change in the state is always the same. Since many animal movements are repetitive in nature, we hypothesise that the neural dynamics are also periodic. This is mathematically described by rotational trajectories in the latent state space of dimension M, which are governed by the antisymmetric matrix $A \in \mathbb{R}^{M\times M}$. The activity in this $\mathbb{R}^M$ space is hypothesised to be linear time-invariant and the rotations are small. The goal is to estimate this A matrix. In the end, we also check to see if our model hallucinates rotational dynamics by perturbing the measured neuron data to see if the rotational behaviour persists or not. We show that it does not, so there really are underlying rotational dynamics.

II. Plotting raw PSTHs

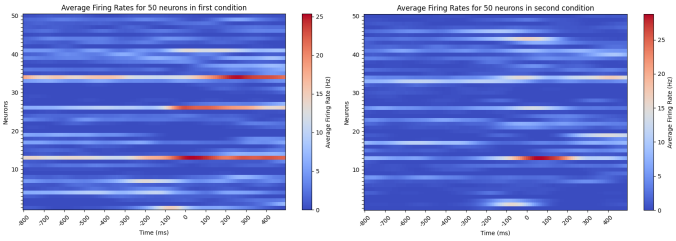
We see in the pre-movement period from $[-800ms, -300ms]$ the firing rates are roughly constant and low. Conversely, peaks occur just before (blue in figure 1(c)) or during the hand movement (orange in figure 1(c)).

The sequence goes as: nothing, target onset, monkey waits(neurons don't fire, since the neurons we measure are for hand movements, not visual stimulus), go cue, monkey has finite reaction time of 0.2sec, target acquisition starts at 0ms.

In figure 1(d) around $t = -200ms$ we see the mean firing rate starting to rise significantly above its baseline level. Assuming monkey has finite reaction time of $200ms$ between the go cue and the hand actually starting to move (at $0ms$), we can say that neurons start firing when the go cue appears.

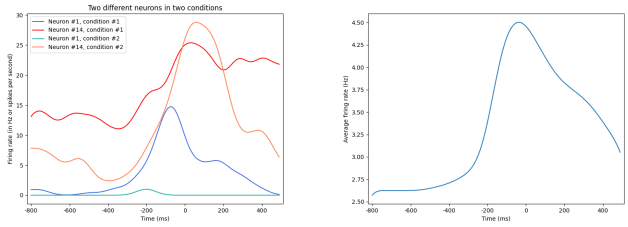
III. Preprocessing

First we normalised for each neuron its PSTH by: $psth = \frac{psth-b}{a-b+5}$, where a and b are, respectively, the maximum and minimum value of this neuron's PSTH



(a) PSTHs of 50 neurons in condition #1

(b) PSTHs of 50 neurons in condition #2



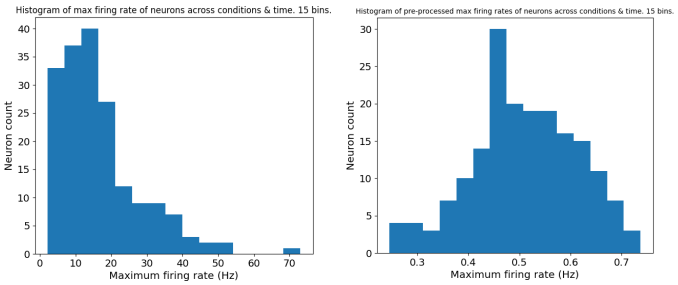
(c) Various neurons' PSTHs

(d) Population average firing rate

Fig. 1: Plots of neurons' PSTHs

across both times and conditions. The 5 is added in order to avoid division by zero errors.

The normalization step will be helpful because there are neurons which contribute to the interesting underlying behaviour, but which have small maximum firing rates compared to other neurons. Imagine if we used raw data from figure 1(a) without normalisation. The three red coloured neurons would dominate in all the subsequent analysis and would cover up anything interesting in the measurement.



(a) Before normalisation

(b) After normalising

Fig. 2: Histograms of maximum neurons' firing rates

IV. PC Space Trajectories

From now we work with PSTHs in the interval $[-150ms, 300ms]$. We use Principle component analysis (PCA) to project the measurements $\boldsymbol{x} \in \mathbb{R}^N$ to $\boldsymbol{z} \in \mathbb{R}^M$, where $\mathbb{R}^M$ is a subspace spanned by the first M principle components (PC) of the sample covariance matrix $\hat{S}$ (1).

In figure 3 we are visualising M dimensional trajectories in a 2D plane spanned by the first two PC's. For further interpretation please see section VI.

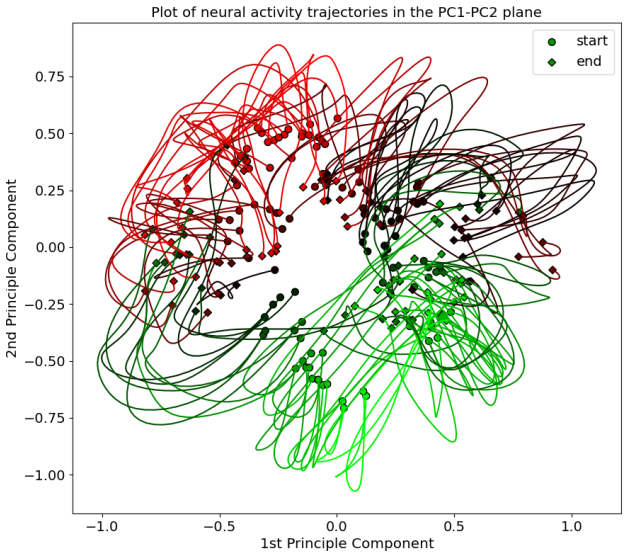


Fig. 3: Trajectories in the PC1-PC2 plane

V. Maximum-likelihood estimate (MLE) for A

The stochastic model with $\boldsymbol{\varepsilon}_t \sim \mathcal{N}(0, I_{M\times M})$ is:

$$\Delta \boldsymbol{z}_{t+1} = A\boldsymbol{z}_t + \sigma\boldsymbol{\varepsilon}_t \tag{1}$$

We treat the $\boldsymbol{z}_t$ as observed variables. Omitting the first measurement $\boldsymbol{z}_1$ in the PC space, because it is independent of the model dynamics, the likelihood for one condition is given by:

$$\begin{aligned} p(Z \mid \sigma, A) &= p\left(\{\boldsymbol{z}_t\}_{t=1}^T \mid \sigma, A\right) \\ &= p(\boldsymbol{z}_1 \mid \sigma, A) \prod_{t=1}^{T-1} p(\boldsymbol{z}_{t+1} \mid \boldsymbol{z}_t, \sigma, A) \end{aligned} \tag{2}$$

As for the log-likelihood, the first measurement $\log p(\boldsymbol{z}_1)$ in the PC space acts like a constant, because it is independent of the model dynamics. Setting $\sigma = 1$ for simplicity and neglecting terms independent of A the log-likelihood is:

$$\begin{aligned} \log p(Z \mid A) &= \sum_{t=1}^{T-1} \log p(\boldsymbol{z}_{t+1} \mid \boldsymbol{z}_t, A) + C \\ &= \sum_{t=1}^{T-1} \log p_{\varepsilon_{t+1}}(\boldsymbol{z}_{t+1} - (A+I)\boldsymbol{z}_t)) + C \\ &= \sum_{t=1}^{T-1} \log \mathcal{N}\left(\Delta\boldsymbol{z}_{t+1} - A\boldsymbol{z}_t; 0, \sigma^2 I_{M\times M}\right) + C \\ &= -\frac{1}{2}\sum_{t=1}^{T-1} (\Delta\boldsymbol{z}_{t+1} - A\boldsymbol{z}_t)^T (\Delta\boldsymbol{z}_{t+1} - A\boldsymbol{z}_t) + C \\ &= -\frac{1}{2}\sum_{t=1}^{T-1} \left(\boldsymbol{z}_t^T A^T A\boldsymbol{z}_t - 2\Delta\boldsymbol{z}_{t+1}^T A\boldsymbol{z}_t\right) + C \end{aligned} \tag{3}$$

To take the gradient with respect to A of equation (3), we note the following known second order derivatives from (2), equations (70) and (77) on pages 10 and 11, respectively:

$$\frac{\partial \boldsymbol{x}^T A \boldsymbol{y}}{\partial A} = \boldsymbol{x}\boldsymbol{y}^T \tag{4}$$

$$\frac{\partial \boldsymbol{x}^T A^T A \boldsymbol{y}}{\partial A} = A\left(\boldsymbol{x}\boldsymbol{y}^T + \boldsymbol{y}\boldsymbol{x}^T\right) \tag{5}$$

In equation (4) set $\boldsymbol{x} = \boldsymbol{z}_t$, $\boldsymbol{y} = \Delta\boldsymbol{z}_{t+1}$ and in equation (5) set $\boldsymbol{x} = \boldsymbol{y} = \boldsymbol{z}_t$. The gradient of the log-likelihood from equation (3) with respect to A is now:

$$\begin{aligned} \nabla_A \log p(Z \mid A) = \\ -\frac{1}{2}\sum_{t=1}^{T-1} \left[2A\left(\boldsymbol{z}_t\boldsymbol{z}_t^T\right) - 2\Delta\boldsymbol{z}_{t+1}\boldsymbol{z}_t^T\right] \end{aligned} \tag{6}$$

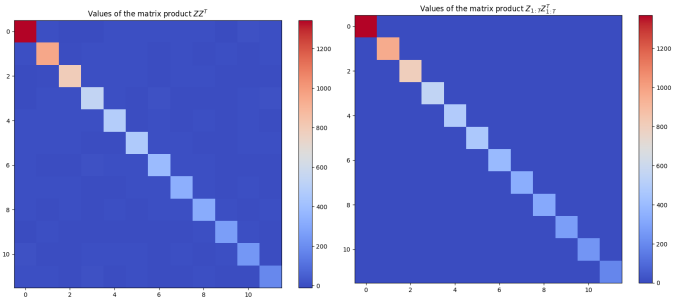
Since we are looking at one condition at the moment, let Z be a matrix of shape $M \times (T-1)$ with column vectors from $\boldsymbol{z}_1$ to $\boldsymbol{z}_{T-1}$ and ΔZ be a matrix with the same shape $M \times (T-1)$ formed with column vectors from $\Delta\boldsymbol{z}_2$ to $\Delta\boldsymbol{z}_T$. That is, $Z = [\boldsymbol{z}_1, ..., \boldsymbol{z}_{T-1}]$ and $\Delta Z = [\Delta\boldsymbol{z}_2, ..., \Delta\boldsymbol{z}_T]$. We can rewrite equation (6) as

$$\nabla_A \log p(Z \mid A) = -AZZ^T + \Delta ZZ^T \tag{7}$$

To get the Maximum likelihood estimate (MLE) of unconstrained A we set equation (7) equal to zero. Hence

$$A_{MLE}^{uncon.} = \Delta ZZ^T \left(ZZ^T\right)^{-1} \tag{8}$$

The previous analysis easily extends to multiple conditions because one of the key assumptions was that autonomous dynamics is the same in all conditions. Thus $\boldsymbol{z}_{c,t}$ for all c are governed by equation (1). Hence from now on we can treat matrices Z and ΔZ like they



(a) ZZ^T. It is a *nearly* diagonal matrix (b) $Z_{1:T}Z_{1:T}^T$. It is a diagonal matrix

Fig. 4: Entries of the two different matrix products

have shape $M \times C(T-1)$. With this extension, we calculated and plotted the unconstrained solution $A_{MLE}^{uncon.}$ from equation (8) in figure 5(a). We see that the entries in figure are nearly antisymmetric, but not exactly. We shall address this issue.

Notice in equation (8) the term $\left(ZZ^T\right)^{-1}$. I noticed the following on a sidenote. The matrix ZZ^T shown in figure 4(a) is *nearly* diagonal matrix. In the figure you can see shades of lightblue in the off-diagonal matrix elements. So we could do a first order approximation of $\left(ZZ^T\right)^{-1}$ by neglecting the off diagonal elements. This could be useful for much bigger datasets.

If we had not discarded the one column in Z and used instead $Z_{1:T} = [\boldsymbol{z}_1, ..., \boldsymbol{z}_{T-1}, \boldsymbol{z}_T]$, then $Z_{1:T}Z_{1:T}^T$ would be a diagonal matrix, as shown in figure 4(b). This is because the rows of such a $Z_{1:T}$ are all orthogonal to eachother. This is due to the way we constructed $Z_{1:T}$ in equation (18) in (1) as $Z_{1:T} = V_M^T X$. V_M has orthogonal column eigenvectors of $\hat{S}$.

A. *Parametrising an antisymmetric A*

Since A is a $M \times M$ antisymmetric matrix, it means the only free parameters are above the main diagonal. A has M zeros on the main diagonal. The number of free parameters K is given by solving $M^2 = M + 2K$, hence we get $K = \frac{M(M-1)}{2}$. For $M = 12$ we have $K = 66$.

Let $\boldsymbol{\beta}$ be a row vector whose entries β_a, $a \in (1, ..., K)$ correspond to $A_{i,j} (j > i)$ in row-major order. There exists a 3D array H of shape $K \times M \times M$ that linearly maps entries in $\boldsymbol{\beta}$ to entries in A.

$$A_{i,j} = \sum_{a=1}^{K} \beta_a H_{a,i,j} \tag{9}$$

All entries $H_{a,i,j}$ take values in $\{-1, 0, 1\}$. Please take a look at page 14 in the appendix for the `index_to_pair()` function.

Let us also define a 3D array W of shape $K \times M \times C(T-1)$, whose entries are defined by:

$$W_{a,i,n} = \sum_{j=1}^{M} H_{a,i,j} Z_{j,n} \tag{10}$$

Using equations (9) and (10) we can write the entries in the matrix product AZ as:

$$\begin{aligned}(AZ)_{i,n} &= \sum_{j=1}^{M} A_{i,j} Z_{j,n} = \sum_{j=1}^{M} \left(\sum_{a=1}^{K} \beta_a H_{a,i,j} \right) Z_{j,n} \\ &= \sum_{a=1}^{K} \beta_a \sum_{j=1}^{M} H_{a,i,j} Z_{j,n} = \sum_{a=1}^{K} \beta_a W_{a,i,n}\end{aligned} \tag{11}$$

Now contracting along repeated index a we have

$$(AZ)_{i,n} = \boldsymbol{\beta} W_{i,n} \tag{12}$$

where $W_{i,n}$ is a column vector with entries $W_{a,i,n}$ for $a \in (1, ..., K)$. Note, this tensor contraction along one repeated index is also called a tensor dot product.

It seems as if we could remove indices i and n. We shall abuse notation by writing $\boldsymbol{\beta} W$, by which we mean a multiplication of $\boldsymbol{\beta}$ of shape $1 \times K$ and W of shape $K \times M \times C(T-1)$ resulting in a matrix of shape $M \times C(T-1)$. Matrix product AZ also has shape $M \times C(T-1)$. Hence we get

$$AZ = \boldsymbol{\beta} W \tag{13}$$

Before continuing further, let us define what is a contraction along two indices (or so called *double tensor contraction*). Let us have some matrix X with shape $P \times Q$. Contraction along two indices, denoted with:

$$X : X \tag{14}$$

is an operation which multiplies every entry of X with itself and sums those entries across the P axis and across the Q axis, resulting in a scalar. Example 1: Let $X = \begin{pmatrix} 1 & 0 & 3 \\ 1 & 0 & 0 \end{pmatrix}$, $X : X = 1 \cdot 1 + 1 \cdot 1 + 3 \cdot 3 = 11$. If we also had $Y = \begin{pmatrix} 3 & 1 & 0 \\ 2 & 0 & 0 \end{pmatrix}$, then $X : Y = Y : X = 1 \cdot 3 + 1 \cdot 2 + \text{zeros} = 5$.

Example 2: if X is a 3D array of shape $P \times Q \times R$ then $X :_{Q,R} X$ would yield a 2D array of shape $P \times P$.

Using this notation, we can rewrite equation (3) as

$$\log p\left(Z \mid A\right) = -\frac{1}{2}\left[AZ : AZ - 2\left(\Delta Z : AZ\right)\right] + C \tag{15}$$

Substituting equation (13) into equation (15) we get the log-likelihood in terms of $\boldsymbol{\beta}$:

$$\log p\left(Z \mid \boldsymbol{\beta}\right) = -\frac{1}{2}\left[\boldsymbol{\beta}W : \boldsymbol{\beta}W - 2\left(\Delta Z : \boldsymbol{\beta}W\right)\right] + C \tag{16}$$

Question is, how do we find the gradient with respect to $\boldsymbol{\beta}$? To do this, let us interpret 3D array W as K matrices, each of shape $M \times C(T-1)$, denoted with $W_1, W_2, ..., W_K$. Now we have:

$$\begin{aligned}
(\boldsymbol{\beta}W) : (\boldsymbol{\beta}W) &= (\sum_{a=1}^{K} \beta_a W_a) : (\sum_{b=1}^{K} \beta_b W_b) \\
&= (\beta_1 W_1 + ... + \beta_K W_K) : (\beta_1 W_1 + ... + \beta_K W_K) \\
&= \beta_1^2 W_1 : W_1 + ... + \beta_1\beta_K W_1 : W_K \\
&+ ... + \beta_K\beta_1 W_K : W_1 + ... + \beta_K^2 W_K : W_K \\
&= \sum_{a=1}^{K}\sum_{b=1}^{K} \beta_a\beta_b \left(W_a : W_b + W_b : W_a\right) \\
&= 2\sum_{a=1}^{K}\sum_{b=1}^{K} \beta_a\beta_b \left(W_a : W_b\right)
\end{aligned}$$

Taking gradient with respect to some β_c, for some $c \in \{1, ..., K\}$ we get

$$\begin{aligned}
\nabla_{\beta_c}\left(\boldsymbol{\beta}W\right) : \left(\boldsymbol{\beta}W\right) &= 2\sum_{a=1}^{K} \beta_a \left(W_a : W_c\right) \\
&= 2\left(\boldsymbol{\beta}W\right) : W_c
\end{aligned}$$

Note that both $\boldsymbol{\beta}W$ and W_c have shapes $M \times C(T-1)$. As a sanity check, this double tensor contraction gives a scalar, which is correct. Vectorising this gradient gives:

$$\nabla_{\boldsymbol{\beta}}\left(\boldsymbol{\beta}W\right) : \left(\boldsymbol{\beta}W\right) = 2\boldsymbol{\beta}\left(W : W\right) \tag{17}$$

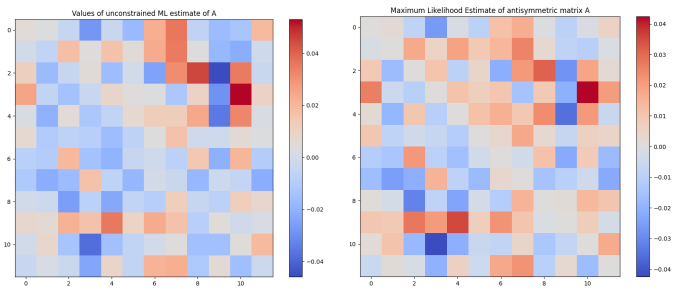
We define matrix $Q = W : W$ analogous to example 2 above. Matrix Q shape $K \times K$.

Hence the gradient with respect to $\boldsymbol{\beta}$ of the log-likelihood in equation (15) is:

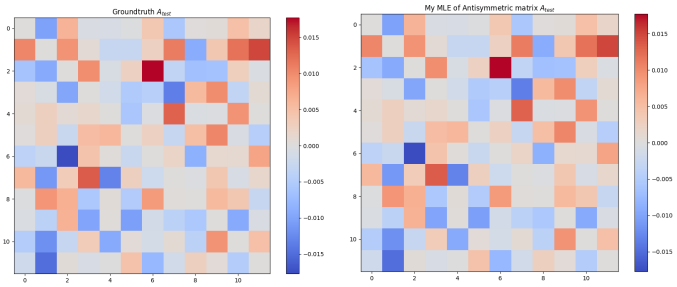
$$\begin{aligned}
\nabla_{\boldsymbol{\beta}} \log p\left(Z \mid \beta\right) &= -\frac{1}{2}\left[2\boldsymbol{\beta}\left(W : W\right) - 2\Delta Z : W\right] \\
&= -\boldsymbol{\beta}\underbrace{W : W}_{Q} + \underbrace{\Delta Z : W}_{\boldsymbol{b}}
\end{aligned} \tag{18}$$

The row vector $\boldsymbol{b}$ is K dimensional because we are performing tensor double contraction of ΔZ with shape $M \times C(T-1)$ and W of shape $K \times M \times C(T-1)$.

If we set the gradient in equation (18) equal to zero and solve, we get the MLE estimate of $\boldsymbol{\beta}$. To obtain the entries in matrix A, we can use equation (9), which is a contraction along one repeated index. The colour plot of the estimated A is shown in figure 5(b).



(a) Unconstrained MLE of A (b) MLE of antisymmetric A



(c) Groundtruth A_{test} (d) MLE of A_{test}

Fig. 5: Colour plots of various A matrices

$\lambda_{1,2}$	$\lambda_{3,4}$	$\lambda_{5,6}$
$\pm j9.18 \cdot 10^{-2}$	$\pm j6.68 \cdot 10^{-2}$	$\pm j4.49 \cdot 10^{-2}$
$\lambda_{7,8}$	$\lambda_{9,10}$	$\lambda_{11,12}$
$\pm j1.61 \cdot 10^{-2}$	$\pm j1.10 \cdot 10^{-2}$	$\pm j2.81 \cdot 10^{-3}$

TABLE I: Eigenvalues of estimated matrix A

To test if our function for estimating A is correct, we pass it a test dataset Z_{test}, which gives a MLE of A_{test} in figure 5(d) and compare it to the groundtruth A_{test} in figure 5(c). As we can see, the two figures are the same and we achieved an accuracy of 10^{-12}.

VI. 2D Projections with Rotational Dynamics

The eigenvalues of the estimated A are shown in table I. The units of the eigenvalues are $\frac{rad}{timestep}$, where $timestep = 10ms$ is the discrete time difference between each two measurements. The eigenvectors will be $M = 12$ dimensional and they have been computed.

Now we will explain the difference between figures 3 and 6.

In figure 3 we see M dimensional trajectories in a 2D plane spanned by PC1 and PC2. These PC1 and PC2 basis vectors can be understood as directions in which the data varies mostly, i.e. has highest variance.

Before moving on, we note the following. First, figures 3 and 6 have the same origin $(0, 0)$. Second, matrix A

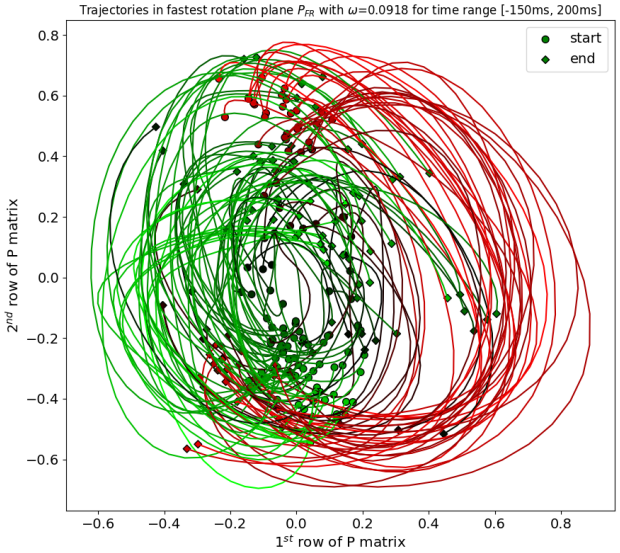


Fig. 6: 2D Trajectories in plane of fastest rotation

represents and infinitesimal rotation in the M dimensional subspace $\mathbb{R}^M$. We decouple this M dimensional rotation into independent 2D rotations in $\frac{M}{2}$ orthogonal planes (which intersect at the origin). These planes are related to the eigenvectors of A as described in the handout.

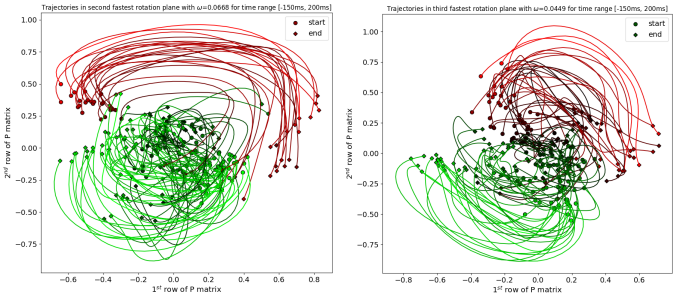
In figure 6 we are visualising one of those 2D rotation planes (in fact the plane with the fastest rotation P_{FR}). Hence we see circular motion around the origin. The basis vectors of that rotation plane have nothing to do with the basis vectors PC1 and PC2 from above.

With this new understanding, we can interpret figure 3 as visualising a M dimensional rotation around the origin $(0, 0)$ in the PC1-PC2 plane. That is why the trajectories look curvy and weird.

We also plotted two more 2D rotation planes in figure 7. Figure 6 corresponds to trajectories the fastest rotation plane and has the greatest eigenvalue from table I. We see that in the same amount of time, the trajectories exhibit much more rotation in this plane that those in figure 7. Likewise, there is more rotation in the second fastest rotation plane in figure 7(a) with eigenvalue λ_3 than in third fastest rotation plane in figure 7(b) with eigenvalue λ_5.

VII. Pre-movement period

Now we apply the projections obtained in the interval $[-150ms, 300ms]$ to the pre-movement period $[-800ms, -150ms]$. This is shown in figure 8 in colours blue to pink. We are visualising both pre-movement period and movement period trajectories in the P_{FR} plane. Notice how the pre-movement pediod trajectories do not have any rotations in them. Rather at the very



(a) 2D Trajectories in 2^{nd} fastest plane of rotation

(b) 2D Trajectories in 3^{rd} fastest plane of rotation

Fig. 7: 2D Trajectories

end of the pre-movement period there is an *out of the centre* movement. This is because on target onset but before the go cue, the monkey has a sense of where it will move its hand, hence some of the neurons start to fire. If the system were truly autonomous, you could predict future motion from pre-movement neural firings because of the assumption that the time derivative is a deterministic function of the state vector.

VIII. Control Analysis

To test whether our model is inherently hallucinating rotational dynamics or not, we distort the data and see what trajectories we get in the fastest plane of rotation P_{FR}.

We distort half of the PSTHs in a manner shown in orange in figure 9(a). At time bin $t = -150ms$ we invert the signal in the movement-period.

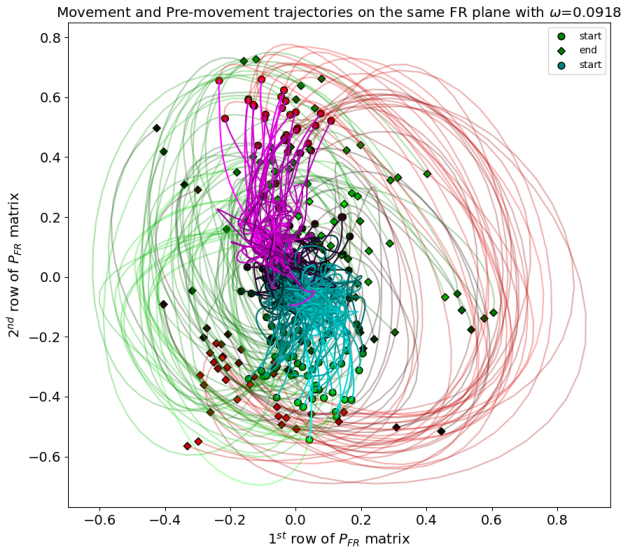


Fig. 8: Pre-movement and movement trajectories in plane of fastest rotation

The distorted signal is still continuous, hence the distortion preserves continuity of the trajectories in the newly fitted plane of fastest 2D rotation, as in figure 9(b).

In the following discussion, everything is viewed in an average sense, i.e. across all neurons and conditions.

The dataset now has a different average rate of change in the signal. In the pre-movement period, before $t = -150ms$, the rate of change goes from zero to some positive value. This corresponds to the initial *out of the centre* trajectories in figure 9(b), very much like those in blue and pink figure 8.

But now the rate of change in the movement-period segment after $t = -150ms$ is roughly zero on average in the whole dataset X, since we inverted exactly half of the signals. Hence, the newly fitted A matrix should not encode any rate of change in the movement-period (if it did, we would deed our model as hallucinatory).

Hence, we can roughly expect in our model $\Delta \boldsymbol{z}_{t+1} \approx A\boldsymbol{z}_t$ that all $\Delta \boldsymbol{z}_{t+1}$ terms in the movement-period have the same magnitude and direction as $\Delta \boldsymbol{z}_{-150ms}$. This would correspond to the trajectories in figure 9(b) just continuing drifting out of the centre with the same velocity they had at $t = -150ms$. A good example of this the trajectory indicated with the arrow in figure 9(b).

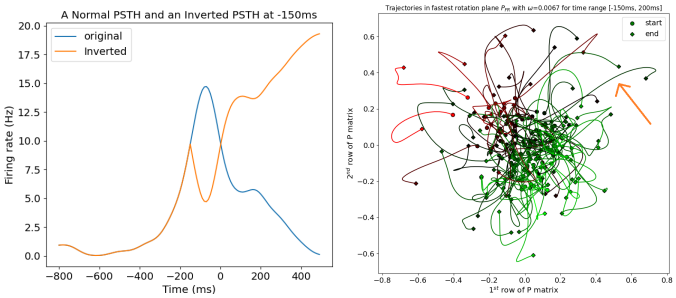
The small curvy rotations at the ends of the trajectories in figure 9(b) could be because we are displaying the plane of fastest rotation and so we could be overfitting the rotations just a little bit, but this is still negligible when compared to figure 6.

We also calculated the largest eigenvalue for this plane is $\lambda_{1,2} = \pm j6.7 \cdot 10^{-3}$, which is a whole magnitude lower than $\lambda_{1,2} = 9.18 \cdot 10^{-2}$ in table I. This is good since the magnitude of the eigenvalue determines the angular velocity of trajectory rotations. Lower magnitude will mean less trajectory rotations, if rotations are even present as in 6 or not present as in 9(b).

Since there are no significant trajectory rotations present in figure 9(b), we conclude that the model is not hallucinating rotations and in fact, rotational dynamics really exist.

IX. Conclusion

In this report we analysed the monkey's motor cortex PSTH measurements in the delayed centre-out reach task. We hypothesised and found out rotational dynamics in the motor cortex by applying a dynamical dimensionality reduction method. First we utilised PCA to get lower M-dimensional latents and treated those as observed variables. Then we fitted the A matrix in which governs the rotational dynamics. Ultimately, we analysed various properties of the neural behaviour in the pre-movement and movement periods with multiple plot visualisations. We also developed good coding practices throughout the project.



(a) Normal and inverted movement-period PSTH

(b) 2D Trajectories of distorted data in fastest plane of rotation

Fig. 9: Control Analysis

References

[1] Yashar Ahmadian. *4G10 BMI: notes for lecture 2 linear atemporal dimensionality reduction*. University of Cambridge. Version: November 8, 2023.

[2] Kaare Brandt Petersen and Michael Syskind Pedersen. *The Matrix Cookbook*. Technical University of Denmark, November 2012. Version: November 15, 2012.

X. Appendix A
My Python Code

```
% modules.py
"""Module providing a function to load a .npz file and to plot a figure."""
from pathlib import Path
import datetime
from numpy import load


def load_data(filename='./data/psths.npz'):
    """Function returns:

    a 3D Numpy array, X, with shape N  C  T,
    containing the so-called PSTHs of N = 182 neurons in T = 130 time-bins
    and C = 108 task conditions. PSTH stands for "peristimulus time histogram".
    and refers to the sequence of average spike counts or firing rates of a neuron
    in different time bins in a time interval around the onset of a stimulus or a movement.
    In our case, the interval goes from -800 ms (milliseconds) to +500 ms relative to
    the onset of hand movement; the interval was divided into 130 time bins of 10 ms width.
    As is commonly done, the trial-averaged spike counts have been divided
    by the bin width (in units of seconds), such that
    X[i, c, t] is the average firing rate of neuron i in
    the t-th time bin in condition c (in units of Hz or spikes per second).

    A 1D array, times, with the start time (in milliseconds) of the different PSTH bins
    relative to movement onset (see Fig. 1A).
    """

    data = load(filename)
    X, times = data['X'], data['times']
    times = times.reshape(len(times), 1)
    return X, times


def load_test_data(filename='./data/test.npz'):
    """
    Returns Z_test 3D array of shape M x C x T and
            A_test 2D array of shape K x K, where K = M(M-1)/2.

            NOTE: A_test CORRESPONDS TO A COMPLETELY DIFFERENT
                  DATASET WHICH HAS NO CONNECTION TO REAL DATASET A
    """

    data = load(filename)
    return data['Z_test'], data['A_test']


def save_fig(fig, filename):
    """Save a figure with a timestamp at end as ID."""

    if filename is None:
        raise ValueError('Filename cannot be None.')

    # Generate a timestamp to make a unique filename
    timestamp = datetime.datetime.now().strftime('%Y%m%d%H%M%S')

    fig.savefig(Path.cwd() / 'plots' / f'{filename}_{timestamp}')

    return None


% q1.py

from pathlib import Path
```

```
import datetime
import numpy as np
import matplotlib.pyplot as plt
from modules import load_data, save_fig

# Load data
X, times = load_data()

# Generate a timestamp to make a unique filename
timestamp = datetime.datetime.now().strftime('%Y%m%d%H%M%S')

# Define the color map (from dark blue to red)
cmap = plt.get_cmap('coolwarm')

# Create first figure and axis
fig1, ax1 = plt.subplots(figsize=(10, 6))

# Choose how many neurons you want to plot and under which condition
n = 51
c = 0

# Create the heatmap
heatmap = ax1.imshow(X[:n, c, :], cmap=cmap, aspect='auto', origin='lower')

# Set the x and y-axis labels
ax1.set_xlabel('Time (ms)')
ax1.set_ylabel('Neurons')

# Set the x-axis and y-axis tick labels using the times list
ax1.set_xticks(np.arange(0, len(times), 10))
ax1.set_xticklabels(np.squeeze(times[::10]), rotation=45)

neuron_labels1 = [str(i) if (i % 10 == 0 and i != 0) else '' for i in range(n)]
ax1.set_yticks(range(n))
ax1.set_yticklabels(neuron_labels1)

# Set title
ax1.set_title(f'Average Firing Rates for 50 neurons in condition #{c+1}')

# Add a color bar to indicate the values
cbar = plt.colorbar(heatmap, ax=ax1)
cbar.set_label('Average Firing Rate (Hz)')

# Save the first plot to an image file
# save_fig(fig1, 'Q1_50_neurons_second_cond')

# Create second figure and axis
fig2, ax2 = plt.subplots(figsize=(8, 6))

# Plot the first neuron activity
i = 0
ax2.plot(times, X[0, 0, :], label='Neuron #1, condition #1',
         color='royalblue')
ax2.plot(times, X[13, 0, :],
         label='Neuron #14, condition #1', color='red')
ax2.plot(times, X[0, 1, :], label='Neuron #1, condition #2',
         color='lightseagreen')
ax2.plot(times, X[13, 1, :],
         label='Neuron #14, condition #2', color='coral')

# Set the x and y axis labels and title
ax2.set_xlabel('Time (ms)')
ax2.set_ylabel('Firing rate (in Hz or spikes per second)')
ax2.set_title(f'Two different neurons in two conditions')

# Set the x axis labels
```

```
ax2.set_xlim(times[0] - 20, times[-1] + 20)
ax2.set_xticks(np.squeeze(times[::20]))
ax2.set_xticklabels(np.squeeze(times[::20]))

ax2.legend()

# save_fig(fig2, f'Q1_two_neurons_in_two_cond')


# Task: Plot the population average firing rate as a function of time,
# obtained by taking the average of the PSTHs across neurons and conditions.

# Create third figure and axis
fig3, ax3 = plt.subplots(figsize=(8, 6))

# Plot the population average firing rate
ax3.plot(times, X.mean(axis=(0, 1)))

# Set the x-axis tick labels
ax3.set_xlim(times[0] - 20, times[-1] + 20)
ax3.set_xticks(np.squeeze(times[::20]))
ax3.set_xticklabels(np.squeeze(times[::20]))

# Label the x-axis and y-axis
ax3.set_xlabel('Time (ms)')
ax3.set_ylabel('Average firing rate (Hz)')

# save_fig(fig3, 'Q1_avg_FR_across_neurons_and_conds')

# Show the plot
plt.show()
```

```
% q2.py


import numpy as np
import matplotlib.pyplot as plt
from modules import load_data, save_fig

X, times = load_data()


def compute_max(X):
    """Compute the maximum of each neuron across conditions and time."""
    return np.max(X, axis=(1, 2))


def plot_max_scatter(X):
    """Plot a scatter plot of the neurons maximum
    (across conditions and time) firing rates."""

    fig1, ax1 = plt.subplots(figsize=(8, 6))

    max_X = compute_max(X)

    ax1.scatter(range(X.shape[0]), max_X)
    ax1.set_title(
        'Plot of maximum firing rate (in Hz) of each neuron across conditions and time')
    ax1.set_xlabel('Neuron')

    plt.show()


def plot_histogram(X, n_bins=15):
```

```
    """Plot a histogram of the neurons' maximum
    (across conditions and time) firing rates.
    """

    # Figure and axis
    fig2, ax2 = plt.subplots(figsize=(7, 6))

    max_X = compute_max(X)

    # Plot histogram
    ax2.hist(max_X, bins=n_bins)

    # Change the tick labels size
    ax2.tick_params(axis='x', labelsize=14)
    ax2.tick_params(axis='y', labelsize=14)

    # Set x-axis and y-axis labels and title
    ax2.set_xlabel('Maximum firing rate (Hz)', fontsize='x-large')
    ax2.set_ylabel('Neuron count', fontsize='x-large')
    ax2.set_title(
        'Histogram of max firing rate of neurons across conditions & time. 15 bins.', fontsize='large')

    save_fig(fig2, filename='Q2_Histogram_of_max_FR_of_each_neuron')

    plt.show()
    plt.clf()


plot_histogram(X)

def normalise(X):
    """
    Separately for each neuron normalize its PSTH according to:
    psth = (psth - b) / (a - b + 5)
    where psth is the PSTH of the neuron in all conditions, and a and b are, respectively, the
    maximum and minimum value of this neuron's PSTH across both times and conditions. This step
    ensures that the normalized activities of different neurons have the same scale and approximate
    range of variations. Henceforth (unless otherwise stated) we will work with this mean-centered
    and normalized PSTH array, which we keep denoting by X.

    input:
    X -> numpy array of dimensions (N x C x T), where
    N = 182 neurons, C = 108 various conditions and T = 130 time bins
    """

    a = np.max(X, axis=(1, 2)).reshape(X.shape[0], 1, 1)
    b = np.min(X, axis=(1, 2)).reshape(X.shape[0], 1, 1)
    return 1.0 * (X - b) / (a - b + 5)


def mean_centering(X):
    """
    Remove from X its cross-condition mean (calculated and subtracted
    separately for each time bin and neuron).
    """

    return X - np.mean(X, axis=1).reshape(X.shape[0], 1, X.shape[2])

# ---------------------- DIMENSIONALITY REDUCITON BY PCA ----------------------

# From this step until exercise 6 we will only work
# with the PSTHs limited to the interval from  150ms  to +300ms relative to movement onset (we
# will however keep using X to denote the corresponding slice of the normalized and mean-removed
# PSTH array, and use T to denote the number of time bins, now equal to 46 in this interval)
```

```
def limit_psth(X, times, lower=-150, upper=300):
    """Limits the psth to interval
    from -150ms to +300ms relative to movement onset.

    Input:
    X -> data matrix with shape (N x C x T), where T = 130.
    times -> ndarray with shape (T x 1), where T = 130.

    Output:
    X -> data matrix with shape (N x CT), where T = 46.
    times -> ndarray with shape (T x 1), where T = 46.
    T -> new number of time bins equal to 46.
    """

    X = mean_centering(normalise(X))

    # Create a boolean mask to select values within the specified range
    # mask shape is (130, 1)
    mask = (times >= lower) & (times <= upper)
    mask = mask[:, 0]  # mask shape is now (130,)

    times = times[mask]

    # number of time bins is now 46
    T = times.shape[0]

    X = X[:, :, mask]  # shape of X is now (N, C, 46)
    X = X.reshape(X.shape[0], -1)  # shape of X is now (N, Cx46)

    return X, times, T


X, times, T = limit_psth(X, times)

# PCA


def pca_proj_matrix(X, M=12):
    """
    Find the eigenvectors and eigenvalues of S_hat = 1/T * X @ X.T
    and take the top M=12 principle components in the neuron activity space.

    input:
    X -> matrix of shape (N x CT) (T should be 46)
    M -> # Number of principal components to select

    output:
    V_M -> matrix of shape (N x M)
    """

    S_hat = 1/T * X @ X.T
    _, evecs = np.linalg.eig(S_hat)

    # Select the top M eigenvectors
    V_M = evecs[:, :M]

    return V_M


def pca_dim_reduction(X, M=12):
    """
    Projecting onto the first M = 12 principle
    components in the neuron activity space.

    Output:
    Z -> matrix of projected neuron activity
```

```
        with shape M    CT = 12    4968. We denote with
        Z[i, n] the elements of Z.

    """

    V_M = pca_proj_matrix(X, M)
    Z = V_M.T @ X
    del V_M

    return Z


V_M = pca_proj_matrix(X)
Z = pca_dim_reduction(X)

print(V_M.shape)
print(Z.shape)

print(np.dot(Z[:, :-1], Z[:, :-1].T).shape)


% q3.py


import numpy as np
import matplotlib.pyplot as plt
import cond_color
from modules import load_data, save_fig
from q2_preprocessing import limit_psth, pca_dim_reduction

# Load data
X, times = load_data()

# Limit the PSTH to time interval between -150ms and +300ms
# Shape of X is (N x CT) = (12 x 4968) and T = 46
X, times, T = limit_psth(X, times)

# Dimensionality reduction by PCA
Z = pca_dim_reduction(X)


def plot_pc1_pc2_plane(Z, plt_title=None, savefig=True, T=46, **kwargs):
    """
    Plot of trajectories in the PC1-PC2 plane (corresponding
    to 0 and 1 left-indices of Z, which for plotting you would
    reshape back into a 3D array). Superimpose
    the trajectories for all conditions in the same plot.

    Input:
    Z -> shape (M x CT)
    savefig (boolean) -> True if you want to save the plot
    T (int) -> Number of timebins used in the last dimension of Z
    Q (int) -> Indicates which question we are solving.
               Q can take values 3 or 5.
    """

    Q = kwargs.get('Q', None)

    if not isinstance(plt_title, str):
        raise ValueError('Plot title must be a string')

    if not (Q == 3 or Q == 5):
        raise ValueError(
            'You can plot only in questions 3 or 5. Please indicate which.')

    # Extract the principle components
```

```
    pc1, pc2 = Z[0, :], Z[1, :]

    # Create a figure
    fig, ax = plt.subplots(figsize=(10, 9))

    # Reshape pc1 and pc2 for ploting purposes
    pc1 = pc1.reshape(-1, T)
    pc2 = pc2.reshape(-1, T)

    # Get the colors in which we plot the trajectories of different conditions
    # xs and ys are coordinates of initial point of trajectories.
    xs, ys = pc1[:, 0], pc2[:, 0]
    colors = cond_color.get_colors(xs, ys, alt_colors=False)
    print(f'{len(colors)}')

    print(f'{pc1.shape=}')
    # Plot all trajectories from various conditions on same plot
    for c in range(pc1.shape[0]):
        ax.plot(pc1[c, :], pc2[c, :], color=colors[c])

    # Plot round markers on the starting point of trajectories
    cond_color.plot_start(xs, ys, colors, markersize=200)

    # Plot diamond-shaped markers on the ending point of trajectories
    xs, ys = pc1[:, -1], pc2[:, -1]
    cond_color.plot_end(xs, ys, colors, markersize=30)

    # Change the tick labels size
    ax.tick_params(axis='x', labelsize=14)
    ax.tick_params(axis='y', labelsize=14)

    # Add legend
    ax.legend(prop={'size': 14})

    if Q == 3 or Q is None:  # We are solving Q3
        # Add axis labels and title for Q3
        ax.set_xlabel('1st Principle Component', fontsize='x-large')
        ax.set_ylabel('2nd Principle Component', fontsize='x-large')
        ax.set_title(plt_title, fontsize='large')

    elif Q == 5:
        # Add axis labels and title for Q5
        ax.set_xlabel('$1^{st}$ row of P matrix', fontsize='x-large')
        ax.set_ylabel('$2^{nd}$ row of P matrix', fontsize='x-large')
        ax.set_title(plt_title, fontsize='large')

    # Save the figure
    if savefig:
        if Q == 3:
            save_fig(fig, 'Q3_Trajectories_in_PC1_PC2_plane')
        elif Q == 5:
            save_fig(fig, 'Q5_P_FR_Trajectories_3rd_fastest')

    plt.show()


plt_title = 'Plot of neural activity trajectories in the PC1-PC2 plane'

# plot_pc1_pc2_plane(Z, plt_title, savefig=False, Q=3)


% q4.py


import numpy as np
import matplotlib.pyplot as plt
```

```
from modules import load_data, save_fig
from q2_preprocessing import limit_psth, pca_dim_reduction


# Load data
X, times = load_data()

# Limit the PSTH to time interval between -150ms and +300ms
# Shape of X is (N x CT) = (12 x 4968) and T = 46
X, times, T = limit_psth(X, times)

# Dimensionality reduction by PCA
Z = pca_dim_reduction(X)  # shape is (M x CT)

M = Z.shape[0]  # M = 12
C = int(Z.shape[1] / T)  # C = 108


# ---------------- Q4(a) Log-likelihood and its (naive) gradient ----------------


# Reshape Z back into a tensor of shape (M x C x T)
Z = Z.reshape(M, -1, T)

# Take difference along time axis. The t+1-th column of dZ
# denoted as dz_{t+1} in the report is dz_{t+1} = z_{t+1} - z_t:
dZ = Z[:, :, 1:] - Z[:, :, :-1]  # shape is (M x C x (T-1))
# print(f'{dZ.shape=}')

# Redefine Z by discarding the last column
Z = Z[:, :, :-1]  # shape is (M x C x (T-1))
# print(f'{Z.shape=}')

# Log-likelihood, log p(Z | sigma=1, A)
# ll = - 0.5 * Z.T @ A.T @ A @ Z + dZ.T @ A @ Z + const

# Naive gradient of ll w.r.t. A:
# dll = -A @ Z @ Z.T + dZ @ Z.T

# Maximum likelihood estimate of an unconstrained A:
# set dll = 0 --> A = dZ @ Z.T @ (Z @ Z.T)^{-1}

# ---------------- Q4(b) Parametrising an antisymmetric A ----------------


def index_to_pair(a, M):
    """
    Given an index a in the range 0 to K-1 (where K = M(M-1)/2),
    corresponding to the a-th entry of vector beta, returns the pair
    of corresponding indices (i, j) in matrix A such that Aij = beta[a],
    where i < j.

    Parameters:
    - a (int): Index in the range 0 to K-1.
    - M (int): Size of the square matrix A.

    Returns:
    - tuple: A tuple (i, j) representing the indices in matrix A.

    Raises:
    - ValueError: If the provided index is out of range.
    """
    if not (0 <= a < M * (M - 1) / 2):
        raise ValueError("Index out of range.")

    # Find i, j such that Aij = beta[a]
```

```
    i = 0
    j = 1
    count = 0

    while count < a:
        j += 1
        if j == M:
            i += 1
            j = i + 1
        count += 1

    return i, j


def construct_H(M):
    """
    Constructs the H tensor. Since the elements of antisymmetric matrix A
    are linearly related to row vector beta of length K, we can write
    the elements of A as a linear combination of beta[a] acording to:

    A[i, j] = sum_{a=1}^{K} beta[a] * H[a, i, j]

    Elements H[a, i, j] take values in {-1, 0, +1}.

    Parameters:
    - M (int): Size of the square matrix A.

    Returns:
    - H (numpy.ndarray) of shape KxMxM: Linearly relates A and beta.
    """

    # K is number of unconstrained entries in A
    K = M * (M - 1) // 2

    H = np.zeros((K, M, M), dtype=int)

    for a in range(K):
        i, j = index_to_pair(a, M)

        H[a, i, j] = 1
        H[a, j, i] = -1

    return H


def reconstruct_A(beta, H):
    """
    Reconstruct antisymmetric matrix A. Basically does this:

    for a in range(K):
        A += beta[:, a] * H[a, :, :]

    Parameters:
    - beta: row vector of shape (1xK)
    - H: tensor of shape (KxMxM)

    Returns:
    - A: antisymmetric matrix with shape (MxM) filled with
         entries of beta in a row major order.
    """

    # Ovo se uopsteno naziva:
    # KONTRAKCIJA PO PONOVLJENOM INDEKSU

    # np.tensordot(beta, H, axes=1) je specijalan slucaj ovoga,
    # koji se jos zove i "tenzorski skalarni proizvod".
```

```
    _, M = beta.shape

    if M != H.shape[0]:
        raise ValueError("Dimensions don't align up.")

    K, M, M = H.shape

    if K != M*(M-1)/2:
        raise ValueError("Check the dimensions of H matrix, they are wrong.")

    A = np.tensordot(beta, H, axes=1)
    return np.squeeze(A, axis=0)


# print(f'{M=}')
H = construct_H(M)


# ---------------- Q4(c) Gradient w.r.t. beta ----------------


# Now that we can represent out A matrix using beta
# We can rewrite the Log-likelihood, log p(Z | sigma=1, A)
# in terms of beta log p(Z | sigma=1, beta)

def construct_W(H, Z):
    """
    Parameters:
    H -> array of shape KxMxM
    Z -> array of shape MxCx(T-1)

    Returns:
    W -> Array of shape KxMxC(T-1)"""

    K, M, M = H.shape

    if K != M*(M-1) / 2 or M != Z.shape[0]:
        raise ValueError(
            "Check the dimensions of H and Z matrix, they are wrong.")

    Z = Z.reshape(M, -1)  # shape is M x C(T-1)

    return np.tensordot(H, Z, axes=1)


W = construct_W(H, Z)
# print(f'{W.shape=}')

# Log-likelihood log p(Z | sigma=1, beta) is:
# ll = -0.5 * [np.tensordot(beta, W, axes=1).T @ np.tensordot(beta, W, axes=1)
#              - 2 * dZ.T @ np.tensordot(beta, W, axes=1)]

# gradient of ll w.r.t. beta is:
# dll = -beta @ np.tensordot()


def construct_Q(W):
    """
    Parameter:
    W -> 3D array of shape K x M x C(T-1)

    Returns:
    Q -> 2D array of shape K x K. We get Q by contracting W with
         itself along 1 and 2 axis.
    """
```

```
    return np.tensordot(W, W, axes=([1, 2], [1, 2]))


Q = construct_Q(W)
# print(f'{Q.shape=}')


def construct_b(dZ, W):
    """
    Parameter:
    dZ -> 3D array of shape M x C x (T-1)
    W ->  3D array of shape K x M x C(T-1)

    Returns:
    b -> Row vector of shape 1 x K. It is calculated by contracting
         reshaped version of dZ and W.
    """

    # Reshape dZ
    dZ = dZ.reshape(M, -1)  # shape is MxC(T-1)
    # print(f'{dZ.shape=}')

    # Compute the row vector b
    b = np.tensordot(dZ, W, axes=([0, 1], [1, 2]))
    b = b.reshape((1, 66))
    # print(f'{b.shape=}')
    return b


b = construct_b(dZ, W)

# The gradient of Log-likelihood w.r.t beta is:
# b - beta @ Q


# ---------(d) Maximum Likelihood Estimate for antisymetric A ---------


def MLE_A(b, Q):
    """Performs Maximum Likelihood Estimate of antisymmetric matrix A."""

    # For MLE set grad of ll w.r.t.b equal to 0, which gives:
    # b - beta @ Q = 0. Hence

    # print(f'{Q.T.shape=}')
    # print(f'{b.T.shape=}')
    beta_transpose = np.linalg.solve(Q.T, b.T)

    beta = beta_transpose.T

    # print(f'{beta.shape=}')

    A = reconstruct_A(beta, H)

    # print(f'{A.shape=}')
    # print(f'{A=}')

    return A


A = MLE_A(b, Q)


def plot_A_matrix(A, test=False):

    # Plot entries of anitysymmetric matrix A
```

```
    fig, ax = plt.subplots(figsize=(10, 8))
    pos = ax.imshow(A, cmap='coolwarm', interpolation='none')
    fig.colorbar(pos, ax=ax)

    if test:  # If we passed the actual solution matrix A
        # pass
        ax.set_title('Groundtruth $A_{test}$')
        # save_fig(fig, 'Q4e_groundtruth_A_test')
    else:
        ax.set_title(f'Maximum Likelihood Estimate of antisymmetric matrix A')
        # save_fig(fig, 'Q4e_MLE_A_colour_plot')

    plt.show()
```

% q4e.py

```
"""Function to estimate matrix A from data Z."""
import numpy as np
import matplotlib.pyplot as plt
from modules import load_data, load_test_data
from q2_preprocessing import limit_psth, pca_dim_reduction
from q4_MLE_for_A import *


def estimate_A(Z):
    """Function to estimate antisymmetric matrix A from data Z.

    Parameters:
    Z (ndarray): 3D data array of shape M x C x T.
                 Z is obtained by doing PCA dimensionality reduction
                 on data matrix X of PSTH recordings.

    Returns:
    A (ndarray): antisymmetric matrix of shape K x K,
                 where K = M(M-1)/2. It governs the autonomous dynamics of
                 the PSTH neural activity.
    """

    M, C, T = Z.shape

    # Take difference along time axis. The t+1-th column of dZ
    # denoted as dz_{t+1} in the report is dz_{t+1} = z_{t+1} - z_t:
    dZ = Z[:, :, 1:] - Z[:, :, :-1]  # shape is (M x C x (T-1))

    # Redefine Z by discarding the last column
    Z = Z[:, :, :-1]  # shape is (M x C x (T-1))

    # H is 3D array which linearly relates row vector beta and A
    H = construct_H(M)

    # W is 3D array of shape KxMxC(T-1). It linearly combines H and Z.
    W = construct_W(H, Z)

    Q = construct_Q(W)  # shape is KxK
    b = construct_b(dZ, W)  # shape is 1xK

    # Perfom Maximum likelihood estimate of A
    A = MLE_A(b, Q)

    return A
```

```
# Load data
X, times = load_data()
Z_test, A_test = load_test_data()

# Limit the PSTH to time interval between -150ms and +300ms
# Shape of X is (N x CT) = (12 x 4968) and T = 46
X, times, T = limit_psth(X, times)

# Dimensionality reduction by PCA
Z = pca_dim_reduction(X)  # shape is (M x CT)

# Reshape Z back into a 3D array of shape (M x C x T)
M = Z.shape[0]  # M = 12
Z = Z.reshape(M, -1, T)

# Estimate the antisymmetric matrix A of shape KxK
A_my_est = estimate_A(Z_test)

# plot_A_matrix(A_my_est, test=False)

print(f'Are the two matricies identical? Ans: \
      {np.allclose(A_my_est, A_test, atol=1e-8, rtol=1e-3)}')

print(f'Max abs difference is: {np.max(np.abs(A_my_est - A_test))}')



% q5.py


"""Compute eigenvalues and eigenvectors of A."""
import numpy as np
from modules import load_data
from q2_preprocessing import limit_psth, pca_dim_reduction
from q3_plot_pc_space_traj import plot_pc1_pc2_plane
from q4e_estimate_matrix_A import estimate_A


# Load data
X, times = load_data()

# Limit the PSTH to time interval between -150ms and +300ms
# Shape of X is (N x CT) = (182 x 4968) and T = 46
X, times, T = limit_psth(X, times)

# Dimensionality reduction by PCA
Z = pca_dim_reduction(X)  # shape is (M x CT)

M = Z.shape[0]
Z = Z.reshape(M, -1, T)

# Estimate the matrix A from actual data
A = estimate_A(Z)


def construct_P(A, rot_plane=1):
    """
    For even M, every rotation in M-dimensional space can be decomposed into independent 2D
    rotations in M/2 orthogonal 2D planes (that only meet at the origin). In the case of our matrix A,
    which represents an infinitesimal rotation, these special 2D planes are related to the eigenvectors
    of A, as follows. The eigenvalues of an antisymmetric matrix are all pure imaginary, and come in
    complex conjugate pairs; in other words, they are all of the form   j   for real and positive    (where
    j =   1   ). The pair of eigenvectors corresponding to a pair of complex conjugate eigenvalues are
    also complex conjugates of each other. It turns out that the real and imaginary parts of (either
    one) of these two eigenvectors constitute a pair of orthogonal real vectors that span one of the
    M/2 special 2D planes. The imaginary part of the corresponding eigenvalue (i.e.,    in the above
```

```
    notation) then provides the angular velocity of the rotation in that special 2D plane.

    Focus first on the eigenvalue with the largest imaginary part; this eigenvalue corresponds to the
    fastest special 2D rotation induced by A. Construct the 2    M matrix P with its two rows given
    by the normalized real and imaginary parts of the eigenvector corresponding to this eigenvalue.
    (Note that the real and imaginary part vectors need to be first normalized by you to have unit
    length.)

    Parameters:
    A -> 2D Square antisymmetric matrix of shape MxM (M is even).
         Obtained from function estimate_A(Z).

    rot_plane (int) -> indicates in which rotation plane
                       you want to visualise the trajectories. Takes value between 1 and M/2.

    Returns:
    P -> Matrix of shape 2 x M
    """

    M = A.shape[0]

    if M % 2 != 0:
        raise ValueError(
            'M must be even, where M is shape of antisymmetric matrix A.')

    if rot_plane not in range(1, M//2):
        raise ValueError(
            '2D rotation plane you select must be in range 1 to {M//2}')

    # Find eigenvalues and eigenvectors of A
    evals, evecs = np.linalg.eig(A)

    np.set_printoptions(formatter={'float': lambda x: "{0:0.2e}".format(x)})
    print(f'{evals.imag=}')
    print(f'{evecs=}')

    evalue = evals[2*(rot_plane-1)]  # Select eval corresponding to rot_plane
    omega = np.abs(evalue.imag)
    print(f'{omega=:.4f}')

    v = evecs[:, 2*(rot_plane-1)]
    real_v, imag_v = v.real, v.imag

    real_v_norm = real_v / np.linalg.norm(real_v)
    imag_v_norm = imag_v / np.linalg.norm(imag_v)

    P = np.zeros((2, M))
    P[0, :] = real_v_norm
    P[1, :] = imag_v_norm

    return P, omega


def project_movement_to_rotation_plane(P, Z):
    """
    By applying P to Z obtain the special 2D projection of the M-dimensional trajectories,
    in the special plane with the fastest rotation. We will call this special plane the plane of fastest
    rotation, or FR plane for short, and will call the corresponding projection matrix P_FR.
    """
    # Plot only from -150ms to +200ms
    Z = Z[:, :, :-10]

    # Compute P_rot
    P_rot = np.tensordot(P, Z, axes=1)  # shape is 2 x C x T

    return P_rot
```

```
def plt_2D_rotation_traj(P_rot, omega, rot_plane=1, savefig=False):
    """
    Plot the projected trajectories.

    rot_plane (int) -> indicates in which rotation plane
                       you want to visualise the trajectories. Takes value between 1 and M/2.

                       NOTE: must be the same value passed as in "construct_P" function.
    """

    if rot_plane == 1:
        plt_title = "Trajectories in fastest rotation plane $P_{FR}$ with "

    elif rot_plane == 2:
        plt_title = "Trajectories in second fastest rotation plane with "

    elif rot_plane == 3:
        plt_title = "Trajectories in third fastest rotation plane with "

    plt_title += f"$\omega$={omega:.4f} for time range [-150ms, 200ms]"

    plot_pc1_pc2_plane(P_rot, plt_title, savefig=savefig, T=36, Q=5)


# To answer Q5d just set rot_plane= 2 or 3
rot_plane = 3

P, omega = construct_P(A, rot_plane=rot_plane)
P_rot = project_movement_to_rotation_plane(P, Z)

plt_2D_rotation_traj(P_rot, omega, rot_plane=rot_plane, savefig=False)



% q6.py


"""Applies projections obtained from interval [-150ms, 300ms]
to the interval [-800ms, -150ms],
which is refered to as pre-movement period.
"""
import matplotlib.pyplot as plt
import cond_color
from modules import load_data, save_fig
from q2_preprocessing import limit_psth, pca_proj_matrix, pca_dim_reduction
from q4e_estimate_matrix_A import estimate_A
from q5 import *

# ------------------- Q5 REPEATED -----------------
# Lower you will find the start of Q6, but this Q5 REPEATED
# code is needed we can answer Q6

# Load data
X, times = load_data()

# Limit the PSTH to time interval [-150ms, +300ms]
# Shape of X is (N x CT) = (182 x 4968) and T = 46
X, times, T = limit_psth(X, times)

# Obtain the PCA Projection matrix V_M
V_M = pca_proj_matrix(X)

# Dimensionality reduction by PCA
Z = pca_dim_reduction(X)  # shape is (M x CT)
```

```
M = Z.shape[0]
Z = Z.reshape(M, -1, T)

# Estimate the matrix A from actual data
A = estimate_A(Z)

# Construct the projection matrix P for fastest rotation plane
P_FR, omega = construct_P(A, rot_plane=1)

# Trajectories are obtained by projecting
# data Z onto the fastest rotation plane
P_rot = project_movement_to_rotation_plane(P_FR, Z)


# ------------------ START OF Q6 ------------------


def combine_P_FR_and_V_M(P_FR, V_M):
    """
    Specifically, combine the exact same PC-projection matrix V_M
    obtained in exercise 2c with the projection onto the FR
    plane found in 5b to obtain a 2    N projection matrix.
    """

    return P_FR @ V_M.T


proj_matrix = combine_P_FR_and_V_M(P_FR, V_M)


# Now we apply the projections obtained for the interval [-150ms, 300ms] to the
# interval [-800ms, -150ms], which we will refer to as the pre-movement period

# Reload data
X, times = load_data()

# Limit the PSTH to time interval [-800ms, -150ms]
# Shape of X is (N x CT) = (182 x 7128) and T = 66
X, times, T = limit_psth(X, times, lower=-800, upper=-150)


# Directly project the N dimensional trajectories during pre-movement period
# onto the FR plane
P_rot_premov = proj_matrix @ X
print(f'{P_rot_premov.shape=}')
P_rot_premov = P_rot_premov.reshape(P_rot_premov.shape[0], -1, T)


def plt_premov_and_mov_rot_trajectories(P_rot, P_rot_premov, omega, savefig=False):
    """Plot superimposed trajectories from Q5c and Q6.

    Parameters:
    P_rot -> contains the monkey movement period trajectories from [-150ms, 200ms]
             projected on the fastest rotation (FR) plane.
             shape is (2 x C x 36)

    P_rot -> contains the monkey pre-movement period trajectories from [-800ms, -150ms]
             projected on the same FR plane as P_rot.
             shape is (2 x C x 66)
    """

    # Extract the first and second row of each matrix
    r1, r2 = P_rot[0], P_rot[1]
    r1_premov, r2_premov = P_rot_premov[0], P_rot_premov[1]
```

```
    # Create a figure
    fig, ax = plt.subplots(figsize=(10, 9))

    # --------- PLOT TRAJECTORIES FROM MONKEY MOVEMENT PERIOD -------------

    # Get the colors in which we plot the trajectories of different conditions
    # xs and ys are coordinates of initial point of trajectories.
    xs, ys = r1[:, 0], r2[:, 0]
    colors = cond_color.get_colors(xs, ys, alt_colors=False)

    # Plot all trajectories from various conditions on same plot
    for c in range(r1.shape[0]):
        ax.plot(r1[c, :], r2[c, :], color=colors[c], alpha=0.3)

    # Plot round markers on the starting point of trajectories
    cond_color.plot_start(xs, ys, colors, markersize=200)

    # Plot diamond-shaped markers on the ending point of trajectories
    xs, ys = r1[:, -1], r2[:, -1]
    cond_color.plot_end(xs, ys, colors, markersize=30)

    # ------- PLOT TRAJECTORIES FROM MONKEY PRE-MOVEMENT PERIOD -----------

    # Get the colors in which we plot the trajectories of different conditions
    # xs and ys are coordinates of FINAL point of trajectories.
    xs, ys = r1_premov[:, -1], r2_premov[:, -1]
    colors = cond_color.get_colors(xs, ys, alt_colors=True)

    # Plot all the trajectories on the same plot
    for c in range(r1_premov.shape[0]):
        ax.plot(r1_premov[c, :], r2_premov[c, :], color=colors[c])

    # Plot round markers on the starting point of trajectories
    cond_color.plot_start(
        r1_premov[:, 0], r2_premov[:, 0], colors, markersize=200)

    # Add legend
    ax.legend()

    # Add axis labels
    ax.set_xlabel('$1^{st}$ row of $P_{FR}$ matrix', fontsize='x-large')
    ax.set_ylabel('$2^{nd}$ row of $P_{FR}$ matrix', fontsize='x-large')

    # Change the tick labels size
    ax.tick_params(axis='x', labelsize=14)
    ax.tick_params(axis='y', labelsize=14)

    # Add title
    ax.set_title(
        f'Movement and Pre-movement trajectories on the same FR plane with $\omega$={omega:.4f}', fontsize='x-

    # Save the figure
    if savefig:
        save_fig(fig, 'Q6')

    plt.show()


plt_premov_and_mov_rot_trajectories(P_rot, P_rot_premov, omega, savefig=True)
```

```
% q7.py
```

```
"""Control Analysis"""
import numpy as np
import matplotlib.pyplot as plt
from modules import load_data, save_fig
from q2_preprocessing import limit_psth, pca_proj_matrix, pca_dim_reduction
from q4e_estimate_matrix_A import estimate_A
from q5 import *


# Load data
X, times = load_data()

# Number of neurons, contitions and timebins
N, C, _ = X.shape

# Create figure
fig, ax = plt.subplots(figsize=(7, 6))

# Add title
ax.set_title('A Normal PSTH and an Inverted PSTH at -150ms',
             fontsize='x-large')
ax.set_xlabel('Time (ms)', fontsize='x-large')
ax.set_ylabel('Firing rate (Hz)', fontsize='x-large')

# Change the tick labels size
ax.tick_params(axis='x', labelsize=14)
ax.tick_params(axis='y', labelsize=14)

# Plot
ax.plot(times, X[0, 0, :], label='original')

# Pick the bin corresponding to time -150ms
t0 = 65
print(f'{times[t0]=}')

# Distort the raw PHTS
for n in range(N):

    # Randomly pick conditions which to flip for each neuron
    c = np.random.choice(C, (C//2,), replace=False)

    # Inversion of subsequent PSTH values about X[n, c, t0]
    X[n, c, t0:] = 2 * np.expand_dims(X[n, c, t0], axis=1) - X[n, c, t0:]


ax.plot(times, X[0, 0, :], label='Inverted')

# Add legend
ax.legend(prop={'size': 14})

# Savefig
save_fig(fig, 'Q7_PSTH')


plt.show()


# --------- Rerun the computational steps from q2 to q5c
# --------- and plot fastest rotation plane

# Limit the PSTH to time interval between -150ms and +300ms
# Shape of X is (N x CT) = (182 x 4968) and T = 46
X, times, T = limit_psth(X, times)

# Obtain the PCA Projection matrix V_M
V_M = pca_proj_matrix(X)
```

```
# Dimensionality reduction by PCA
Z = pca_dim_reduction(X)  # shape is (M x CT)

M = Z.shape[0]
Z = Z.reshape(M, -1, T)

# Estimate the matrix A from actual data
A = estimate_A(Z)


# choose rotation plane
rot_plane = 2

# Construct the projection matrix P for fastest rotation plane
P_FR, omega = construct_P(A, rot_plane)

# Trajectories are obtained by projecting
# data Z onto the fastest rotation plane
P_rot = project_movement_to_rotation_plane(P_FR, Z)

# Plot the trajectories in the fastest rotation plane

plt_2D_rotation_traj(P_rot, omega, rot_plane, savefig=True)
```