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

Module	4G10	Title of report	Coursework Assignment 2
Date submitted: 13/12/2023		Assessment for this module is ☑ 100% / ☐ 25% coursework of which this assignment forms _50__ %	

UNDERGRADUATE and POST GRADUATE STUDENTS		
Candidate number:	5553C	☑ 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:			
PRE	**Attention to detail, typesetting and typographical errors** Is the report free of typographical errors? Are the figures/tables/references presented professionally?			

4G10: Predicting Hand Kinematics from Neural Data

5535C

December 13, 2023

Abstract

In order to use data from the brain to effectively move prosthetic limbs, accurate interpretation of neural activity is required. We attempt to predict hand velocities of monkeys reaching to a target with neural activity in their motor cortex. First, linear regression with Gaussian smoothing is used; this is the most simple model and as expected performs poorly, explaining 43.4% of the hand velocity variability. We then use an adjusted model, the Wiener filter, which is able to capture temporal inter-dependencies in the neural activity; this explained 70.1% of the hand velocity variability. Lastly the Kalman filter was able to explain 69.2% of the hand velocity variability. All the models struggle to perform exceptionally due to inherent linearity whilst working with the non-linear nature of the brain.

1 Linear regression with Gaussian smoothing

1.1 Method

We start by convolving the neural activity of each of the 162 neurons with a Gaussian kernel as given by Equation 1. This is done for each of the 400 trials, given by k (Note that convolution is over the 16 time bins, indicated by t). The data is now smoothed in the time domain, as shown by Figure 1.

$$\tilde{\mathbf{x}}_{k,t} = \int_{-\infty}^{\infty} f(\tau)\mathbf{x}_{k,t-\tau}d\tau$$
$$\text{where } f(\tau) \propto \exp\frac{-\tau^2}{2\sigma^2} \tag{1}$$

Now we move onto the instantaneous decoder, given by Equation 2. The predicted x and y velocities are stored in $\hat{\mathbf{v}}_{k,t} \in \mathcal{R}^2$; this is calculated from the learnt weight matrix ($\mathbf{W} \in \mathcal{R}^{2\times 162}$) and the smoothed neural activity $\tilde{\mathbf{x}}_{k,t} \in \mathcal{R}^{162}$.

$$\hat{\mathbf{v}}_{k,t} = \mathbf{W}\tilde{\mathbf{x}}_{k,t} \tag{2}$$

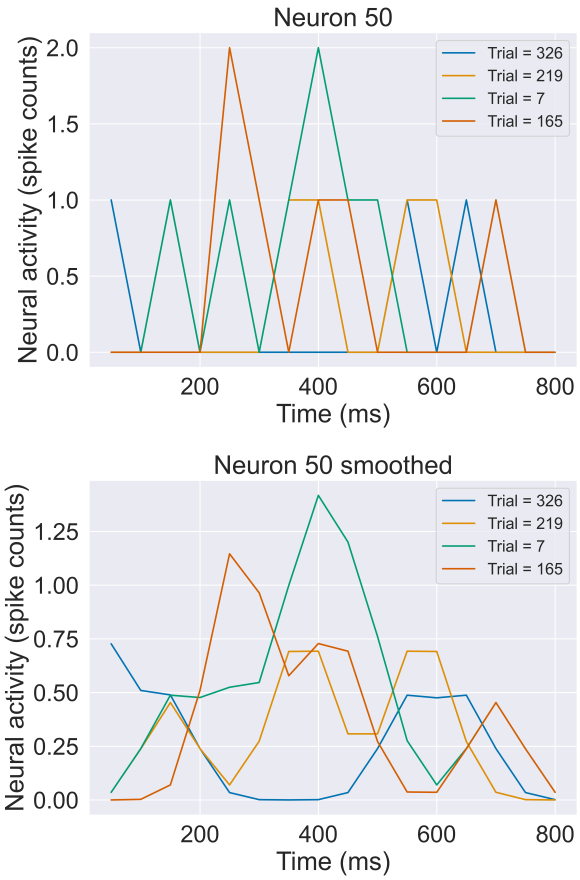

Figure 1: Comparison of the original (Top) vs the smoothed (Bottom) neural activity.

We minimise this via Equation 3 (the hand training data is also smoothed). Note the addition of the regularising term on the weight matrix $\mathbf{W}$ to protect against overfitting.

$$min_{\mathbf{W}}||\mathbf{W}\tilde{\mathbf{x}}_{k,t} - \tilde{\mathbf{v}}_{k,t}||_2^2 + \lambda||\mathbf{W}||_2^2 \quad (3)$$

The maximum a posteriori solution to the minimisation problem above is given by Equation 4, where $\mathbf{V} \in \mathcal{R}^{2\times(400\times16)}$ and $\tilde{\mathbf{X}} \in \mathcal{R}^{162\times(400\times16)}$.

$$\mathbf{W_{MAP}} = \mathbf{V}\tilde{\mathbf{X}}^T(\tilde{\mathbf{X}}\tilde{\mathbf{X}}^T + \lambda\mathbf{I_N})^{-1} \quad (4)$$

Note that the hand velocities in the training and test sets have been shifted backwards $120ms$ relative to the neural activity in order to account for the fact that neural activity for a given movement increases rapidly as the subject plans their upcoming trajectory. The peak of neural activity is often at the onset of the movement.

1.2 Hyper-parameter selection

The first hyper-parameter to tune was the width of the Gaussian kernel σ, used for smoothing the neural activity. Intuitively, this would ideally allow a given time to be influenced by the adjacent times, to reverse the harsh boundary enforced by the discrete time in $50ms$ bins (e.g. activity at $75ms$ only contributes to the time at $100ms$ bin despite being just as close to the $50ms$ bin). We found the optimal value was $\sigma = 0.88$ (width of $44ms$), which as shown in Figure 2, yields the aforementioned shape; neighbouring samples have some contribution to a given time, with the other times all close to zero.

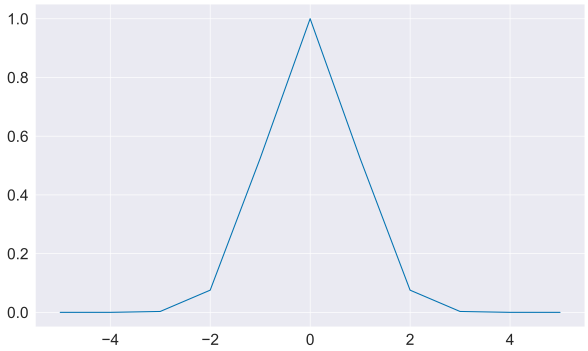

Figure 2: plot of the Gaussian smoothing kernel with $\sigma = 0.88$ (width of $44ms$).

The second hyper-parameter to tune is the regularising constant. This is $\lambda = \frac{\phi^2}{\nu^2}$, where $\phi^2\mathbf{I_2}$ is the covariance of the likelihood $P(\hat{\mathbf{v}}_{k,t}|\tilde{\mathbf{x}}_{k,t})$ and $\nu^2\mathbf{I_2}$ is the covariance of our prior on $\mathbf{W}$ (which has a zero mean); the prior is introduced to keep the weights small, preventing the model from overfitting to the training data. Although if λ is too large Equation 3 will find it more valuable to minimise the regularisation term and hence under-fit the data.

We proceeded with $\lambda = 1$ as this would not be large enough for overfitting and we have no further information indicating that the variance in the likelihood should be smaller or larger than that in the prior. Further testing proved that this choice was near optimal, although if we had a validation set, an 'L-curve' could be used to better choose λ.

1.3 Online suitability

Computationally, this is a plausible method as it only requires Equation 2, which is a relatively small matrix multiplication, to be calculated after each time step. Using NumPy this only takes $0.662ms$ (3 s.f.).

With the hyper-parameters $\lambda = 1$ and $\sigma = 0.88$ our model achieved an R^2 coefficient of 0.434. That means our model can describe 43.4% of the variability in the velocity. This level of inaccuracy will make accurate movement in an online BMI context extremely difficult; with path integration, the inaccurate velocities will lead to ever increasingly inaccurate locations. Furthermore, if used in closed loop, the subject

will struggle as the decoder's poor performance will now be part of the loop as visual feedback.

Lastly, this model was trained on data that was smoothed using a Gaussian kernel that smoothes data at a given time using adjacent times; this won't be possible in real-time as we won't have access to future times. To remedy this we should train the model on a data that has been smoothed by a kernel which only uses current and past times, so that when we decode in 'on the fly' we can continue to use the same kernel that the model was trained on.

2 Wiener filter

An alternate method to smoothing current and past times using simple kernel (such as an exponential), could be to use a kernel which is learnt and directly outputs the predicted velocities.

2.1 Method

The above can be described by Equation 5, Where lag is a hyper-parameter which determines the number of different times used to calculate the velocity at a given time.

$$\hat{\mathbf{v}}_{k,t} = \sum_{l=0}^{lag-1} \mathbf{W}_l \mathbf{x}_{k,t-l} \tag{5}$$

Equation 5 is now reorganised into Equation 6, where $\tilde{\mathbf{x}}_{k,t} \in \mathcal{R}^{162*lag}$; $\tilde{\mathbf{x}}_{k,t}$ now stores not only the data of the neurons at the given time and trial, but also the data of the neurons before it. If $lag = 5$ it will contain the current time data, as well as the data of the 4 preceding times.

$$\hat{\mathbf{v}}_{k,t} = \mathbf{W}\tilde{\mathbf{x}}_{k,t} \tag{6}$$

The solution of Equation 6 is now simply the least-squares solution given by Equation 7.

$$\mathbf{W_{ML}} = \mathbf{V}\tilde{\mathbf{X}}^T(\tilde{\mathbf{X}}\tilde{\mathbf{X}}^T)^{-1} \tag{7}$$

2.2 Online suitability

Computationally this model also feasible, as 'on the fly' each step only takes $7.98ms$ to compute which is still far less than the bin size; this is longer than the previous model due to the need to concatenate the data of the current time to a vector, whilst popping the data no longer needed.

With $lag = 8$ this model was able to obtain an R^2 coefficient of 0.701, so this model is able to explain 70.1% of the variability in the velocities (Note that with a lag of 8 only the velocities of the last 9 times could be calculated, hence to calculate the R^2 coefficient, the previous times steps were taken from the previous model; this means performance is even higher). This is far more accurate than the previous model, which achieved an R^2 coefficient of 0.434, and hence will perform movement far more accurately to a level which is likely manageable for a subject in closed loop.

This increase of performance is expected, as this model uses the neural activity of previous times and hence is able to capture temporal interdependence, whereas the previous model virtually atemporal (although not fully atemporal due to the Gaussian smoother obtaining information from other times).

3 Kalman filter

In this model, we capture the temporal interdependence, by using a auto-regressive prior for the temporal dynamics of the signals, which is then used in the generative LDS model of neural data.

3.1 Method

We begin with a pre-trained 10-dimensional linear latent dynamical system on the hand velocities of the training data. The generative model is as follows:

$$\begin{aligned} \mathbf{z}_{k,0} &\sim \mathcal{N}(\mu_0, \mathbf{\Sigma}_0) \\ \mathbf{z}_{k,t+1} &= \mathbf{A}\mathbf{z}_{k,t} + \mathcal{N}(\mathbf{0}, \mathbf{Q}) \\ \mathbf{v}_{k,t} &= \mathbf{C}\mathbf{z}_{k,t} + \mathcal{N}(\mathbf{0}, \mathbf{R}) \end{aligned} \tag{8}$$

Using the above model, we find the filtering distribution $P(\mathbf{z}_{k,t}|\mathbf{x}_{0:t})$, given by Equation 9).

$$\begin{aligned} \mathbf{P}_{k,t} &= \mathbf{A}\mathbf{\Sigma}_{k,t}\mathbf{A}^T + \mathbf{Q} \\ \mathbf{\Sigma}_{k,t+1}^{-1} &= \mathbf{C}^T\mathbf{R}^{-1}\mathbf{C} + (\mathbf{P}_{k,t})^{-1} \\ \mu_{k,t+1} &= \mathbf{\Sigma}_{k,t+1}((\mathbf{P}_{k,t})^{-1} + \mathbf{A}\mu_{k,t} \\ &\quad + \mathbf{C}^T\mathbf{R}^{-1}\mathbf{v}_{k,t+1}) \end{aligned} \tag{9}$$

Now with the entire series we are able to find the smoothing distribution $P(\mathbf{z}_{k,t}|\mathbf{x}_{0:T})$, given by backwards update steps in Equation 10).

$$\begin{aligned} \mathbf{G}_{k,t} &= \mathbf{\Sigma}_{k,t}\mathbf{A}^T(\mathbf{P}_{k,t})^{-1} \\ \tilde{\mu}_{k,t} &= \mu_{k,t} + \mathbf{G}_{k,t}(\tilde{\mu}_{k,t+1} - \mathbf{A}\mu_{k,t}) \\ \tilde{\mathbf{\Sigma}}_{k,t} &= \mathbf{\Sigma}_{k,t} + \mathbf{G}_{k,t}(\tilde{\mathbf{\Sigma}}_{k,t+1} - \mathbf{P}_{k,t})\mathbf{G}_{k,t}^T \end{aligned} \tag{10}$$

We then use supervised learning to model the neural activity with Equation 11, where $\hat{\mathbf{z}}_{k,t}$ is the mean of the latent variables from the smoothing distribution.

$$\mathbf{x}_{k,t} = \mathbf{D}\hat{\mathbf{z}}_{k,t} + \mathcal{N}(\mathbf{0}, \mathbf{S}) \tag{11}$$

We find the optimal solutions by differentiating the log probability over all trials and times and differentiating with respect to D and S.

$$\begin{aligned} P(\mathbf{x}_{1:K,1:T}|\mathbf{z_{1:K,1:T}}) &= \prod_{k,t}\mathcal{N}(\mathbf{D}\hat{\mathbf{z}}_{k,t}, \mathbf{S}) \\ log(P) &= log(\prod_{k,t}\mathcal{N}(\mathbf{D}\hat{\mathbf{z}}_{k,t}, \mathbf{S})) \\ &= \sum_{k,t} log(\mathcal{N}(\mathbf{D}\hat{\mathbf{z}}_{k,t}, \mathbf{S})) \end{aligned} \tag{12}$$

Expanding this results in Equation 13. To save space we will use $\mathbf{x}$ and $\hat{\mathbf{z}}$ in stead of $\mathbf{x}_{k,t}$ and $\hat{\mathbf{z}}_{k,t}$.

$$\begin{aligned} log(P) &= -\frac{1}{2}\sum_{k,t}(\mathbf{x}^T\mathbf{S}^{-1}\mathbf{x} - 2\hat{\mathbf{z}}^T\mathbf{D}^T\mathbf{S}^{-1}\mathbf{x} + \\ &\quad \hat{\mathbf{z}}^T\mathbf{D}^T\mathbf{S}^{-1}\mathbf{D}\hat{\mathbf{z}} - log((2\pi)^D|\mathbf{S}|)) \\ log(P) &= -\frac{1}{2}\sum_{k,t}(\mathbf{x}^T\mathbf{S}^{-1}\mathbf{x} - 2\hat{\mathbf{z}}^T\mathbf{D}^T\mathbf{S}^{-1}\mathbf{x} \\ &\quad + \hat{\mathbf{z}}^T\mathbf{D}^T\mathbf{S}^{-1}\mathbf{D}\hat{\mathbf{z}}) - KTlog((2\pi)^D|\mathbf{S}|) \end{aligned} \tag{13}$$

The optimal solution is given by Equations 14 and 15.

$$\begin{aligned} &\frac{\partial\mathcal{L}}{\partial\mathbf{D}} = -\frac{1}{2}\sum_{k,t}(2(\mathbf{x}^T\mathbf{S}^{-1})^T\hat{\mathbf{z}}^T - 2\mathbf{S}^{-T}\mathbf{D}\hat{\mathbf{z}}\hat{\mathbf{z}}^T) \\ &\frac{\partial\mathcal{L}}{\partial\mathbf{D}} = 0 \\ &\Rightarrow \sum_{k,t}\mathbf{S}^{-T}\mathbf{x}\hat{\mathbf{z}}^T = \sum_{k,t}\mathbf{S}^{-T}\mathbf{D}^*\hat{\mathbf{z}}\hat{\mathbf{z}}^T \\ &\Rightarrow \mathbf{D}^* = \left(\sum_{k,t}\mathbf{x}_{k,t}\hat{\mathbf{z}}_{k,t}^T\right)\left(\sum_{k,t}\hat{\mathbf{z}}_{k,t}\hat{\mathbf{z}}_{k,t}^T\right)^{-1} \end{aligned} \tag{14}$$

$$\begin{aligned} &\frac{\partial\mathcal{L}}{\partial\mathbf{S}} = -\frac{1}{2}\sum_{k,t}(-\mathbf{S}^{-T}\mathbf{x}\mathbf{x}^T\mathbf{S}^{-T} + 2\mathbf{S}^{-T}\mathbf{D}\hat{\mathbf{z}}\mathbf{x}^T\mathbf{S}^{-T} \\ &- \mathbf{S}^{-T}\mathbf{D}\hat{\mathbf{z}}\hat{\mathbf{z}}^T\mathbf{D}^T\mathbf{S}^{-T}) - \frac{KT}{2}\mathbf{S}^{-1} \\ &\frac{\partial\mathcal{L}}{\partial\mathbf{S}} = 0 \\ &\Rightarrow KT\mathbf{S}^* = -\mathbf{x}\mathbf{x}^T + 2\mathbf{D}\hat{\mathbf{z}}\mathbf{x}^T - \mathbf{D}\hat{\mathbf{z}}\hat{\mathbf{z}}^T\mathbf{D}^T \\ &\Rightarrow \mathbf{S}^* = \frac{1}{KT}\left(\sum_{k,t}\mathbf{x}_{k,t}\mathbf{x}_{k,t}^T - \mathbf{D}^*\sum_{k,t}\hat{\mathbf{z}}_{k,t}\mathbf{x}_{k,t}^T\right) \end{aligned} \tag{15}$$

Using the above solution, we are now able to use the Kalman filter to predict hand velocity with the neural activity test set.

3.2 Performance

This model resulted in a R^2 coefficient of 0.692. This is a large increase from the linear regression with smoothing model and is on par with the Wiener filter. The increase is due to the auto-regressive prior on temporal latent

dynamics; this allows information regarding a given neural observation to propagate downstream and affect latent variables in the next time step, which was learnt when the hand velocity was treated as the observed variable. As a result this model is not atemporal and hence is able to understand the temporal interdependence in the neural activity.

3.3 Online suitability

This model is computationally feasible, as the calculation for the next hand velocity takes approximately $0.566ms$, which is very fast. This is this advantage of the model over the wiener filter, which is required to concatenate and pop data from a vector at each time step and therefore takes far more time than matrix algebra on this scale.

As this model is able to explain 69.2% of the variability of the hand velocities, it will perform far better than linear regression with smoothing model, although accuracy will not be high enough for a subject to use effectively day-to-day.

3.4 Improvements

One possible way of increasing the performance of the Kalman filter, would be to increase the dimension of the latent variable; this would allow it to capture more information on the latent dynamics when being trained, and hence utilise more information from the neural activity when predicting hand velocities, which will result in more accurate predictions. The drawback of this is that the computation time will start to rise rather quickly due to the number of times the latent variable is used when calculating hand velocity.

Furthermore if collecting more data is an option, one could collect more to be more accurate, as well as collect more variables, for example the position and acceleration of the hand; now we provide more information which will further assist when training of the auto-regressive prior. The resulting model will better understand the latent dynamics and more effectively predict the hand velocity. This will of course lead to a greater amount of time required and a more complex system to collect training data.

4 Alternate models

A more dramatic change would be to use non-linear latent dynamics; The current Kalman filter is linear which is unable to effectively understand the non-linearity of the brain. An LFAD would be able to capture the non-linearity of features and hence provide a more accurate way to infer hand velocities. A major drawback of this is that computationally this will take longer, although not to long that this method is not feasible. Furthermore training of the model will be more computationally intensive as the system is no longer linear and Gaussian.

An alternate method to capture non-linearity would be the use of an LSTM recurrent neural network; this won't be as complicated to optimise as an LFAD due to open-source libraries such as Keras, although it still is very computationally intensive, even with the use of GRUs, and will require expensive GPUs to speed up.

5 Code

5.1 Linear regression with Gaussian smoothing

```
1 from io import BytesIO
2 import numpy as np
3 import requests
4 from scipy.ndimage import gaussian_filter
5 from scipy.linalg import inv
```

```

# grab the data from the server
r = requests.get('https://4G10.cbl-cambridge.org/data.npz', stream=True)
data = np.load(BytesIO(r.raw.read()))

Neural_test = data['neural_test']
Neural_train = data['neural_train']
Hand_train = data['hand_train']


# Gaussian Smoothing
def smoother(Neural_train, std):
    return gaussian_filter(Neural_train, sigma=std, axes=2)


# Decoder
def W_MAP_calc(V, X_til, lam):
    lam_I_n = lam * np.identity(X_til.shape[0])
    brackets = inv(np.tensordot(X_til, X_til, axes=([1, 2], [1, 2])) + lam_I_n)
    return np.tensordot(V, X_til, axes=([1, 2], [1, 2])) @ brackets


# Predict
for std in [0.88]:
    Neural_train_filt = smoother(Neural_train, std)
    Neural_test_filt = smoother(Neural_test, std)
    for lam in [0.01, 0.1, 1, 2.5, 5]:
        W_MAP = W_MAP_calc(Hand_train, Neural_train, lam)
        Predicted_vel = np.tensordot(W_MAP, Neural_test_filt, axes=([1], [0]))
```

5.2 Wiener filter

```
from io import BytesIO
import numpy as np
import requests
from scipy.linalg import inv

# grab the data from the server
r = requests.get('https://4G10.cbl-cambridge.org/data.npz', stream=True)
data = np.load(BytesIO(r.raw.read()))

Neural_test = data['neural_test']
Neural_train = data['neural_train']
Hand_train = data['hand_train']

lag = 8

X_train_lags = np.zeros((lag * 162, 400, 16 + 1 - lag))
for k in range(400):
    for t in range(16 + 1 - lag):
        for l in range(lag):
            X_train_lags[(l*162):((l+1)*162), k, t] = Neural_train[:, k, t + lag
                - l - 1]

Hand_train_wien = Hand_train[:, :, (lag - 1):16]

X_lagsTX_lags = np.tensordot(X_train_lags, X_train_lags, axes=([1, 2], [1, 2]))
W_wien = inv(X_lagsTX_lags) @ np.tensordot(X_train_lags, Hand_train_wien, axes
    =([1, 2], [1, 2]))

```

```
X_test_lags = np.zeros((lag * 162, 100, 16 + 1 - lag))
for k in range(100):
    for t in range(16 + 1 - lag):
        for l in range(lag):
            X_test_lags[(l*162):((l+1)*162), k, t] = Neural_test[:, k, t + lag - l - 1]

V_predict = np.tensordot(W_wien, X_test_lags, axes=([0], [0]))
```

5.3 Kalman filter

```
from io import BytesIO
import numpy as np
import requests
from scipy.linalg import inv

# grab the data from the server
r = requests.get('https://4G10.cbl-cambridge.org/data.npz', stream=True)
data = np.load(BytesIO(r.raw.read()))


# 3.1 An autoregressive prior for hand kinematics
z_0 = np.random.multivariate_normal(data['hand_KF_mu0'].reshape((-1,)), data['hand_KF_Sigma0'], size=(100,))

epsilon = np.random.multivariate_normal(np.zeros(10), data['hand_KF_Q'], size=(100, 16))

A = data['hand_KF_A']
C = data['hand_KF_C']
R = data['hand_KF_R']
Q = data['hand_KF_Q']
Sigma0 = data['hand_KF_Sigma0']
mu0 = data['hand_KF_mu0'].reshape((10,))
v_train = data['hand_train']

Sigma_inv = np.zeros((10, 10, 400, 16))
mu = np.zeros((10, 400, 16))

CTRC = C.T @ inv(R) @ C
for i in range(Sigma_inv.shape[2]):
    P = inv(A @ Sigma0 @ A.T + Q)
    Sigma_inv[:, :, i, 0] = CTRC + P
    mu[:, i, 0] = inv(Sigma_inv[:, :, i, 0]) @ (P @ A @ mu0 + C.T @ inv(R) @ v_train[:, i, 0])
    for j in range(Sigma_inv.shape[3] - 1):
        P = inv(A @ inv(Sigma_inv[:, :, i, j]) @ A.T + Q)
        Sigma_inv[:, :, i, j + 1] = CTRC + inv(A @ inv(Sigma_inv[:, :, i, j]) @ A.T + Q)
        mu[:, i, j + 1] = inv(Sigma_inv[:, :, i, j + 1]) @ (P @ A @ mu[:, i, j] + C.T @ inv(R) @ v_train[:, i, j + 1])

Sigma_tilde = np.zeros((10, 10, 400, 16))
mu_tilde = np.zeros((10, 400, 16))

for i in range(Sigma_tilde.shape[2]):
    mu_tilde[:, :i, -1] = mu[:, :i, -1]
    Sigma_tilde[:, :, i, -1] = inv(Sigma_inv[:, :, i, -1])
    for j in range(Sigma_tilde.shape[3] - 1)[::-1]:
        Sigma_t = inv(Sigma_inv[:, :, i, j])
```

```
45          P_t = A @ Sigma_t @ A.T + Q
46          G_t = Sigma_t @ A.T @ inv(P_t)
47          mu_tilde[:, i, j] = mu[:, i, j] + G_t @ (mu_tilde[:, i, j+1] - A @ mu[:,
                i, j])
48          Sigma_tilde[:, :, i, j] = Sigma_t + G_t @ (Sigma_tilde[:, :, i, j+1] -
                P_t) @ G_t.T
49
50
51  # 3.2 Building an LDS model of neural data using supervised learning
52
53
54  def center_data(data):
55      mean = data.mean(axis=(1, 2))
56      mean = mean.reshape(162, 1, 1)
57      data_centered = data - mean
58      return data_centered
59
60
61  neural_train_cent = center_data(data['neural_train'])
62  neural_test_cent = center_data(data['neural_test'])
63
64  D_opt = np.tensordot(neural_train_cent, mu_tilde, axes=([1, 2], [1, 2])) @ inv(
        np.tensordot(mu_tilde, mu_tilde, axes=([1, 2], [1, 2])))
65
66  S_opt = (np.tensordot(neural_train_cent, neural_train_cent, axes=([1, 2], [1,
        2])) - D_opt @ np.tensordot(mu_tilde, neural_train_cent, axes=([1, 2], [1,
        2])))/(400*16)
67
68  # 3.3 Using Kalman filtering to predict the hand velocity
69  Sigma_inv_test = np.zeros((10, 10, 100, 16))
70  mu_test = np.zeros((10, 100, 16))
71
72  S_inv = inv(S_opt)
73  DTSD = D_opt.T @ S_inv @ D_opt
74  for i in range(Sigma_inv_test.shape[2]):
75      P = inv(A @ Sigma0 @ A.T + Q)
76      Sigma_inv_test[:, :, i, 0] = DTSD + P
77      mu_test[:, i, 0] = inv(Sigma_inv_test[:, :, i, 0]) @ (P @ A @ mu0 + D_opt.T
            @ S_inv @ neural_test_cent[:, i, 0])
78      for j in range(Sigma_inv_test.shape[3] - 1):
79          P = inv(A @ inv(Sigma_inv_test[:, :, i, j]) @ A.T + Q)
80          Sigma_inv_test[:, :, i, j + 1] = DTSD + inv(A @ inv(Sigma_inv_test[:, :,
                i, j]) @ A.T + Q)
81          mu_test[:, i, j + 1] = inv(Sigma_inv_test[:, :, i, j + 1]) @ (P @ A @
                mu_test[:, i, j] + D_opt.T @ S_inv @ neural_test_cent[:, i, j + 1])
82
83  v_test = np.tensordot(C, mu_test, axes=([1], [0]))
```