CUED - Engineering Tripos Part IIB 2024-2025 **Module Coursework**

Module	4G10	Title of report	4G10 Coursework 2

Date submitted: 25 Dec	Assessment for this module is ☑ 100% / ☐ 25% coursework of which this assignment forms 50 %

UNDERGRADUATE and POST GRADUATE STUDENTS		
Candidate number:	5641G	☑ 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:

4G10 Coursework 2

Auther: 5641G

1 Introduction

This report addresses the task of predicting hand kinematics from neural data recorded in the primary motor cortex (M1) of a macaque monkey. The dataset consists of neural spike counts from 162 cortical neurons and corresponding 2D hand velocity measurements across multiple reaching trials. Each trial begins at a 'go' cue and lasts 800 ms, divided into 16 bins of 50 ms. The data is partitioned into training and testing sets, with 400 and 100 trials respectively. Various decoding techniques are explored, including Gaussian smoothing combined with ridge regression and Kalman filtering. The results are analyzed to assess prediction accuracy and computational feasibility for online decoding applications.

2 Baseline Decoder: Simple Gaussian Smoothing + Linear Regression

The baseline decoding approach consists of two stages: smoothing the spike count time series using a Gaussian filter and applying ridge regression for linear decoding. Ridge regression minimizes the following cost function to balance data fidelity and regularization:

$$\mathcal{L} = ||\mathbf{V} - \mathbf{W}\tilde{\mathbf{X}}||_2^2 + \lambda||\mathbf{W}||_2^2, \tag{1}$$

where $\mathbf{V}$ is the $2 \times (400 \cdot 16)$ matrix of hand velocities, $\tilde{\mathbf{X}}$ is the $N \times (400 \cdot 16)$ matrix of smoothed neural spike counts, λ is the regularization parameter, and $\mathbf{W}$ represents the decoder weights. The closed-form solution for $\mathbf{W}$ is given by:

$$\mathbf{W}^\star = \mathbf{V}\tilde{\mathbf{X}}^\top \left(\tilde{\mathbf{X}}\tilde{\mathbf{X}}^\top + \lambda\mathbf{I}_N\right)^{-1}. \tag{2}$$

Here, $\tilde{\mathbf{X}}\tilde{\mathbf{X}}^\top$ forms a Gram matrix, as it is not mean-centered.

2.1 Choice of λ

To determine an optimal value for λ, several values were tested with σ fixed at 40 ms. Table 1 summarizes the results. The optimal value, $\lambda = 75$, balances regularization and prediction accuracy:

λ	0	0.1	0.5	1	2	10	50	100	200	10^4
R^2	0.422	0.422	0.422	0.423	0.423	0.425	0.428	0.428	0.426	0.390

Table 1: Selection of λ with $\sigma = 40$ ms

2.2 Effect of Smoothing Window Length σ

The impact of smoothing window length σ on decoding accuracy was analyzed using offline ridge regression with $\lambda = 75$. As shown in Table 2 and the blue curve in Figure 1, the quality of hand velocity predictions peaks at $\sigma \approx 60$ ms. Shorter windows (e.g., $\sigma = 20$ ms) fail to capture sufficient temporal structure, likely overfitting noise. Conversely, overly large windows (e.g., $\sigma = 80$ ms) blur meaningful temporal variations, degrading performance.

σ (ms)	20	25	30	40	50	55	60	65	70	75	80
R^2	0.345	0.375	0.401	0.428	0.441	0.444	0.445	0.445	0.443	0.440	0.437

Table 2: Effect of σ on offline decoding performance with $\lambda = 75$

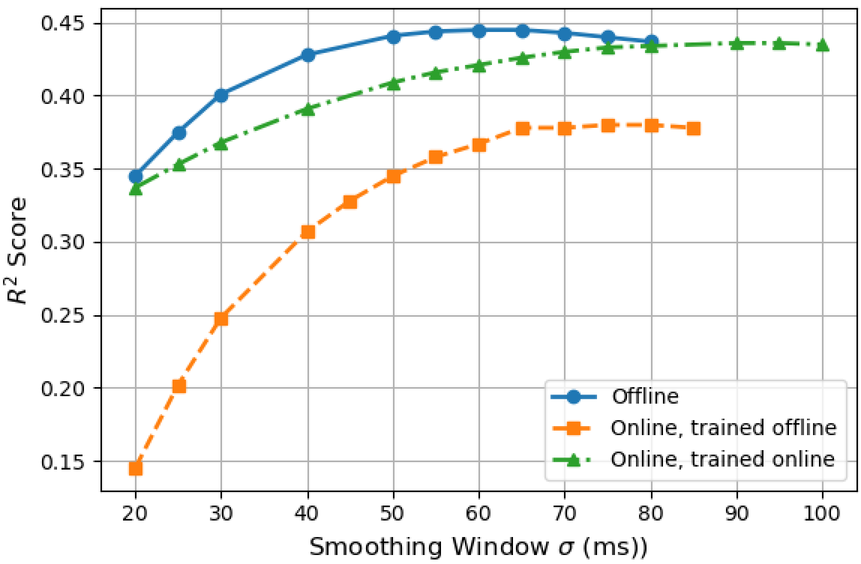

Figure 1: Effect of σ on R^2 for baseline decoder and its variants.

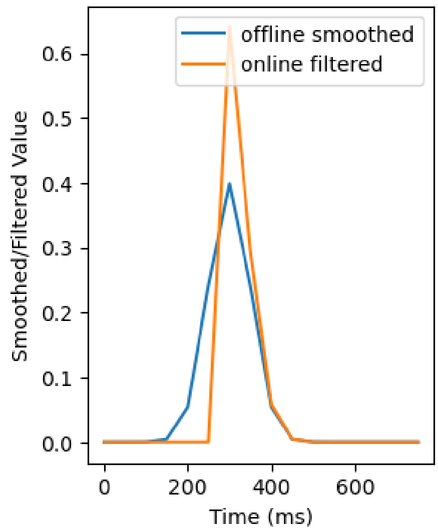

Figure 2: Comparison of online filtering and offline smoothing.

2.3 Suitability for Online Decoding

The simple decoding strategy is computationally efficient and feasible for online decoding in a brain-machine interface (BMI) context. However, its accuracy drops significantly when applied online, as shown in Table 3 and the orange curve in Figure 1. This decline arises because online filtering does not access future information, leading to suboptimal smoothing. Figure 2 illustrates this limitation, where online filtering differs markedly from offline smoothing for a single spike.

σ (ms)	20	30	40	50	60	65	70	75	80	85
R^2	0.145	0.248	0.307	0.345	0.367	0.378	0.378	0.380	0.380	0.378

Table 3: Online decoding performance using offline-trained weights with $\lambda = 75$

Additionally, neural tuning properties may drift over time, reducing decoding accuracy. Adaptive updates or retraining of the model may mitigate this issue.

2.4 Improving Online Decoding

A simple improvement involves training the weights $\mathbf{W}$ on online-filtered training data instead of smoothed data. This modification aligns the training process with the online setting, resulting in significant performance improvements. As shown in Table 4 and green curve in Figure 1, the R^2 score improves to 0.436 at $\sigma \approx 90$ ms, approaching the optimal offline performance of 0.445.

σ (ms)	20	30	40	50	60	70	80	90	95	100
R^2	0.337	0.368	0.391	0.409	0.421	0.430	0.434	0.436	0.436	0.435

Table 4: Online decoding performance using online-trained weights with $\lambda = 75$

2.5 Temporal Shift in Hand Velocity Data

The hand velocity data in `data['hand_train']` was intentionally shifted backward by 120 ms relative to the neural data. This adjustment likely accounts for the natural delay in the motor control process. Neural activity in the primary motor cortex (M1) precedes the execution of motor actions because it encodes planning and initiation signals for movement. By aligning the neural data with the hand velocity data in this manner, the dataset ensures that neural signals related to planning and initiation are temporally synchronized with the corresponding behavioral outputs. Without this shift, the behavioral data would lag behind the neural data, introducing a temporal mismatch that could compromise the performance of decoding algorithms.

3 Kalman Filter-Based Decoding

We now introduce a more advanced decoding method based on a Kalman filter and smoother.

3.1 An Autoregressive Prior for Hand Kinematics

A pre-trained 10-dimensional linear latent dynamical system (LDS) is utilized to model hand velocity data from the training set. The generative model is defined as follows:

$$\mathbf{z}_{k,0} \sim \mathcal{N}(\boldsymbol{\mu}_0, \boldsymbol{\Sigma}_0) \quad \text{(initial prior)} \tag{3}$$
$$\mathbf{z}_{k,t+1} = \mathbf{A}\mathbf{z}_{k,t} + \boldsymbol{\epsilon}_{k,t+1}, \quad \boldsymbol{\epsilon}_{k,t+1} \sim \mathcal{N}(\mathbf{0}, \mathbf{Q}) \quad \text{(state transition)} \tag{4}$$
$$\mathbf{v}_{k,t} = \mathbf{C}\mathbf{z}_{k,t} + \boldsymbol{\eta}_{k,t}, \quad \boldsymbol{\eta}_{k,t} \sim \mathcal{N}(\mathbf{0}, \mathbf{R}) \quad \text{(observation model)} \tag{5}$$

Here, $\mathbf{z}_{k,t} \in \mathbb{R}^{10}$ represents the latent state at time bin t of trial k, and $\mathbf{v}_{k,t} \in \mathbb{R}^2$ represents the corresponding hand velocity.

The Kalman smoother combines forward (filtering) and backward (smoothing) passes to estimate latent states. The goal is to implement a Kalman smoother to compute the mean $\hat{z}_{k,1:T}$ of the smoothing distribution $p(z_{k,t}|v_{k,1:T})$ for each trial k in the training set.

3.1.1 Kalman Filter (Forward Pass)

The Kalman filter recursively estimates the latent state z_t based on observations up to time t:

$$\mathbf{P}_t = \mathbf{A}\boldsymbol{\Sigma}_t\mathbf{A}^\top + \mathbf{Q}, \tag{6}$$
$$\mathbf{K}_t = \mathbf{P}_t\mathbf{C}^\top \left(\mathbf{R} + \mathbf{C}\mathbf{P}_t\mathbf{C}^\top\right)^{-1}, \tag{7}$$
$$\boldsymbol{\Sigma}_{t+1} = \left(\mathbf{I} - \mathbf{K}_t\mathbf{C}\right)\mathbf{P}_t, \tag{8}$$
$$\boldsymbol{\mu}_{t+1} = \mathbf{A}\boldsymbol{\mu}_t + \mathbf{K}_t\left(\mathbf{x}_{t+1} - \mathbf{C}\mathbf{A}\boldsymbol{\mu}_t\right). \tag{9}$$

Here, $\mathbf{K}_t$ is the Kalman gain, which balances the prior and the observation likelihood. Observations $\mathbf{x}_t$ are the hand velocity measurements $v_{k,t}$.

3.1.2 Kalman Smoother (Backward Pass)

The Kalman smoother refines the filtered estimates by incorporating future observations:

$$\mathbf{P}_t = \mathbf{A}\boldsymbol{\Sigma}_t\mathbf{A}^\top + \mathbf{Q}, \tag{10}$$
$$\mathbf{G}_t = \boldsymbol{\Sigma}_t\mathbf{A}^\top\mathbf{P}_t^{-1}, \tag{11}$$
$$\tilde{\boldsymbol{\mu}}_t = \boldsymbol{\mu}_t + \mathbf{G}_t\left(\tilde{\boldsymbol{\mu}}_{t+1} - \mathbf{A}\boldsymbol{\mu}_t\right), \tag{12}$$
$$\tilde{\boldsymbol{\Sigma}}_t = \boldsymbol{\Sigma}_t + \mathbf{G}_t\left(\tilde{\boldsymbol{\Sigma}}_{t+1} - \mathbf{P}_t\right)\mathbf{G}_t^\top. \tag{13}$$

Here, $\tilde{\boldsymbol{\mu}}_t$ and $\tilde{\boldsymbol{\Sigma}}_t$ represent the smoothed mean and covariance, respectively.

These updates were implemented and applied to compute smoothed latent state estimates for the training data. The resulting $\hat{z}_{k,1:T}$ values correspond to the smoothed means $\tilde{\boldsymbol{\mu}}_t$, which represent the posterior mean of the latent state distribution after incorporating both past and future observations. These values enables the predictive decoding of hand velocity from neural data.

3.2 Building an LDS Model of Neural Data Using Supervised Learning

In this section, we extend the generative LDS model to neural data:

$$\mathbf{x}_{k,t} = \mathbf{D}\mathbf{z}_{k,t} + \xi_{k,t}, \quad \xi_{k,t} \sim \mathcal{N}(\mathbf{0}, \mathbf{S}) \tag{14}$$

where $\mathbf{x}_{k,t}$ represents the neural activity (spike counts) for trial k at time t, and $\mathbf{z}_{k,t}$ is the latent state variable, which can be obtained from the Kalman smoother in Section 3.1.

3.2.1 Centering the Neural Data

To ensure proper alignment between neural data and latent state dynamics, we first center the neural data:

- Compute the mean activity across trials and time for each neuron in the training set.
- Subtract this mean from both the training and testing neural data.

3.2.2 Fitting Parameters

The likelihood parameters $\mathbf{D}$ (observation matrix) and $\mathbf{S}$ (observation noise covariance) are estimated by maximizing the average joint log-likelihood:

$$\log p(\hat{\mathbf{z}}_{k,1:T}, \mathbf{x}_{k,1:T}) = \frac{1}{K}\sum_k \log p(\mathbf{x}_{k,1:T}|\hat{\mathbf{z}}_{k,1:T}) + \log p(\hat{\mathbf{z}}_{k,1:T}) \tag{15}$$

$$= \frac{1}{K}\sum_k \left[\sum_{t=1}^{T} \log p(\mathbf{x}_{k,t}|\hat{\mathbf{z}}_{k,t}) + \log p(\hat{\mathbf{z}}_{k,1:T})\right]. \tag{16}$$

where K is the number of trials, $\hat{\mathbf{z}}_{k,1:T}$ is the posterior mean from Section 3.1, and $\mathbf{x}_{k,1:T}$ is the neural data. Since $p(\hat{\mathbf{z}}_{k,1:T})$ is independent of $\mathbf{D}, \mathbf{S}$, we only maximize:

$$\log p(\mathbf{x}_{k,t}|\hat{\mathbf{z}}_{k,t}) = \frac{1}{K}\sum_{k,t}\left[-\frac{1}{2}\log(2\pi) - \frac{1}{2}\log|\mathbf{S}| - \frac{1}{2}(\mathbf{x}_{k,t} - \mathbf{D}\hat{\mathbf{z}}_{k,t})^\top \mathbf{S}^{-1}(\mathbf{x}_{k,t} - \mathbf{D}\hat{\mathbf{z}}_{k,t})\right]. \tag{17}$$

Gradient w.r.t D

$$\frac{\partial}{\partial \mathbf{D}} \log p = \frac{1}{K}\sum_{k,t} \frac{\partial}{\partial \mathbf{D}}\left[-\frac{1}{2}(\mathbf{x}_{k,t} - \mathbf{D}\hat{\mathbf{z}}_{k,t})^\top \mathbf{S}^{-1}(\mathbf{x}_{k,t} - \mathbf{D}\hat{\mathbf{z}}_{k,t})\right] \tag{18}$$

$$= \frac{1}{K}\sum_{k,t}\left[\mathbf{S}^{-1}\mathbf{x}_{k,t}\hat{\mathbf{z}}_{k,t}^\top - \mathbf{S}^{-1}\mathbf{D}\hat{\mathbf{z}}_{k,t}\hat{\mathbf{z}}_{k,t}^\top\right]. \tag{19}$$

Setting this gradient to zero and solving for $\mathbf{D}^\star$:

$$\mathbf{S}^{-1}\frac{1}{K}\sum_{k,t}\left[\mathbf{x}_{k,t}\hat{\mathbf{z}}_{k,t}^\top - \mathbf{D}^\star\hat{\mathbf{z}}_{k,t}\hat{\mathbf{z}}_{k,t}^\top\right] = \mathbf{0} \tag{20}$$

$$\mathbf{D}^\star = \left(\sum_{k,t}\mathbf{x}_{k,t}\hat{\mathbf{z}}_{k,t}^\top\right)\left(\sum_{k,t}\hat{\mathbf{z}}_{k,t}\hat{\mathbf{z}}_{k,t}^\top\right)^{-1}. \tag{21}$$

Gradient w.r.t S

Applying the formula $\frac{\partial}{\partial \mathbf{A}}\left(\mathbf{x}^\top\mathbf{A}^{-1}\mathbf{x}\right) = -\mathbf{A}^{-1}\mathbf{x}\mathbf{x}^\top\mathbf{A}^{-1}$.

$$\frac{\partial}{\partial \mathbf{S}} \log p = \frac{1}{K}\sum_{k,t}\left[-\frac{1}{2}\mathbf{S}^{-1} + \frac{1}{2}\mathbf{S}^{-1}(\mathbf{x}_{k,t} - \mathbf{D}\hat{\mathbf{z}}_{k,t})(\mathbf{x}_{k,t} - \mathbf{D}\hat{\mathbf{z}}_{k,t})^\top\mathbf{S}^{-1}\right]. \tag{22}$$

Setting this gradient to zero, solving for $\mathbf{S}^\star$

$$\mathbf{S}^\star = \frac{1}{KT}\sum_{k,t}(\mathbf{x}_{k,t} - \mathbf{D}^\star\hat{\mathbf{z}}_{k,t})(\mathbf{x}_{k,t} - \mathbf{D}^\star\hat{\mathbf{z}}_{k,t})^\top \tag{23}$$

Substituting $\mathbf{D}^\star$

$$\mathbf{S}^\star = \frac{1}{KT}\sum_{k,t}\left[\mathbf{x}_{k,t}\mathbf{x}_{k,t}^\top - \mathbf{D}^\star\hat{\mathbf{z}}_{k,t}\mathbf{x}_{k,t}^\top - \mathbf{x}_{k,t}\hat{\mathbf{z}}_{k,t}^\top\mathbf{D}^{\star\top} + \mathbf{D}^\star\hat{\mathbf{z}}_{k,t}\hat{\mathbf{z}}_{k,t}^\top\mathbf{D}^{\star\top}\right] \tag{24}$$

$$= \frac{1}{KT}\left(\sum_{k,t}\mathbf{x}_{k,t}\mathbf{x}_{k,t}^\top - \mathbf{D}^\star\sum_{k,t}\hat{\mathbf{z}}_{k,t}\mathbf{x}_{k,t}^\top\right) \tag{25}$$

3.2.3 Final Parameter Estimates

The fitted parameters are:

$$\mathbf{D}^{\star} = \left(\sum_{k,t} \mathbf{x}_{k,t} \hat{\mathbf{z}}_{k,t}^{\top} \right) \left(\sum_{k,t} \hat{\mathbf{z}}_{k,t} \hat{\mathbf{z}}_{k,t}^{\top} \right)^{-1} \tag{21}$$

$$\mathbf{S}^{\star} = \frac{1}{KT} \left(\sum_{k,t} \mathbf{x}_{k,t} \mathbf{x}_{k,t}^{\top} - \mathbf{D}^{\star} \sum_{k,t} \hat{\mathbf{z}}_{k,t} \mathbf{x}_{k,t}^{\top} \right) \tag{25}$$

These parameters define the observation model for neural data and will be used for decoding hand velocity in subsequent sections.

3.3 Using Kalman Filtering to Predict the Hand Velocity

Using the model parameters obtained in Section 3.2, we implemented a Kalman filter to compute the filtered posterior $p(\mathbf{z}_{k',t}|\mathbf{x}_{k',0:t})$ for each trial k' in the test set. The mean of this posterior, $\hat{\mathbf{z}}_{k',t}$, serves as the best estimate of the latent state at time t. The hand velocity is then predicted as:

$$\hat{\mathbf{v}}_{k',t} = \mathbf{C}\hat{\mathbf{z}}_{k',t}. \tag{26}$$

The predicted velocities were submitted to the server, yielding an R^2 score of 0.78.

Why Are These Predictions Better? The Kalman filter outperforms the baseline decoder (Section 2) for several reasons:

- **Temporal Dynamics:** Unlike the baseline method, the Kalman filter incorporates the sequential dependencies of the latent states through the state transition model. This allows it to leverage past observations to make more accurate predictions of the current state.
- **Noise Reduction:** The Kalman gain dynamically balances the contribution of the prior estimate and the new observation, effectively filtering out noise in the neural data.
- **Latent State Representation:** By utilizing the latent state dynamics modeled in the LDS, the Kalman filter captures the underlying neural representation of hand movement, leading to more precise predictions.

Suitability for Online Decoding The Kalman filter is highly suitable for online brain-computer interface (BCI) applications due to its recursive, computationally efficient nature, requiring only the current state estimate and the latest observation at each time step. The adaptive Kalman gain ensures robust performance in noisy environments.
However, its assumptions of static model parameters ($\mathbf{A}$, $\mathbf{C}$, $\mathbf{Q}$, $\mathbf{R}$) may limit adaptability to neural tuning changes over time, and delays in signal acquisition could impact real-time performance.

Further Improvements To enhance decoding performance, the following approaches are proposed:

- **ReFIT-Kalman Filter:** Adjusts decoder parameters during closed-loop operation based on user feedback, improving accuracy and robustness to neural activity changes. However, it requires additional calibration time.
- **Nonlinear State-Space Models:** Methods such as Recurrent Neural Networks (RNNs) and Gaussian Process Dynamical Models (GPDMs) better capture complex neural dynamics but increase computational requirements.
- **Adaptive Noise Models:** Dynamically adjusting $\mathbf{Q}$ and $\mathbf{R}$ improves noise handling, though it adds computational overhead.
- **Cortical Dynamics Integration:** Incorporating models that explicitly account for cortical interactions can improve temporal decoding accuracy at the cost of higher model complexity.

A Code Implementation

```
# [markdown]
# # 4G10 Coursework 2: predicting hand kinematics from neural data
#
#
# Please read carefully the last section of this notebook, which gives some of our expectations regarding your report.
#
# In this handout,
# - <u>text that is underlined</u> corresponds to things you have to do / implement.
# - **text in bold** corresponds to questions you need to answer in some form in your report.
#

#
from io import BytesIO
import numpy as np
import matplotlib.pyplot as plt
import requests

#  [markdown]
# # 1. Setup
#
# In this piece of 4G10 coursework, you will use neural data recorded in the primary motor cortex (M1) of a reaching monkey to predict the kinematics of the monkey's hand.
#
# The monkey initiated each trial by placing their hand in the center of a fronto-parallel screen. A target then appeared on the screen. The monkey had to wait for a 'go' cue before making a reaching movement towards the instructed target. The targets were placed in various positions in a virtual maze, which changed in each trial, forcing the monkey to make a variety of reaching movements across trials.
#
# The activity of $N=162$ motor cortical neurons was recorded simultaneously, alongside the kinematics of the animal's hand.
#
# In the dataset presented below, all time series are partitioned into trials. Each trial begins at the go cue and lasts 800ms (T = 16 bins of 50ms duration) roughly the duration of a reach.

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

#  [markdown]
# Among other things (detailed later), this dictionary numerical arrays indexed by the following keys:
# - "hand_train" (2 x 400 x T): 2D velocity (X/Y) of the monkey's hand in 400 'train' trials;
# - "neural_train" (N x 400 x T): neural activity (spike counts) in the same 400 'train' trials;
# - "neural_test" (N x 100 x T): neural activity (spike counts) in 100 'test' trials.
#
# E.g.:

#

```

```
# Load data
hand_train = data["hand_train"]
neural_train = data["neural_train"]
neural_test = data["neural_test"]

print(hand_train.shape)
print(neural_train.shape)
print(neural_test.shape)

# np.save('train_file.npy', hand_train[:,0:100,:].astype("float64"))

#
import numpy as np
import matplotlib.pyplot as plt

X = neural_train

# Compute the mean along dimension 1 (axis=1)
mean_dim1 = np.mean(X, axis=(1))

# Plot X over the mean of dim 1
plt.figure(figsize=(10, 6))
for i in range(5):
    plt.plot(mean_dim1[i,:], linewidth=2, label='Mean (dim 1)')  # Plot the mean
plt.title("Neural Data: Each Neuron vs. Mean (dim 1)")
plt.xlabel("Sample Index")
plt.ylabel("Value")
plt.legend(loc="upper right")
plt.grid(True)
plt.show()

#  [markdown]
# The goal of this CW is to implement some of the modelling / decoding techniques you have been taught in lectures, to predict the monkey's 2D hand velocity in the 100 test trials for which you are only given neural activity. Your predictions will be based on the training data provided (hand_train, neural_train).

#  [markdown]
# # 2. Baseline decoder: simple Gaussian smoothing + linear regression
#
# To establish a meaningful baseline, you will first implement a very simple two-stage decoder.
# In the first stage, you will smooth the spike count time series of each neuron by convolving it with a Gaussian filter of width $\sigma$; in continuous time, such a Gaussian filter is given by $f(t) \propto \exp(-t^2/2\sigma^2)$.
# In the second stage, you will use ridge regression to learn an instantaneous linear decoder given by
#
# $$ \hat{v}_{k,t} = W \tilde{x}_{k,t} $$
#
# where $\hat{v}_{k, t} \in \mathbb{R}^2$ is the predicted velocity of the hand in test trial $k$ and time bin $t$, $\tilde{x}_{k, t} \in \mathbb{R}^N$ is the $t^\text{th}$ time bin of the temporally smoothed spike counts in test trial $k$, and $W$ is a 2 x N matrix of decoding weights. Note that the hand velocity data has been centered already, so there is no need to include a bias term in the regression.
#
# The optimal ridge regression weights are given by
# $$  W^\star = V \tilde{X}^\top (\tilde{X} \tilde{X}^\top + \lambda I_N )^{-1} $$
# where $V$ is the $2 x (400*16)$ matrix of hand velocities from the training set (with all trials and time bins concatenated horizontally), and similarly $\
```

```
    tilde{X}$ is the $N x (400*16)$ matrix of smoothed neural spike counts in the training set.
#
# In the equation above, $\lambda$ is a regularisation parameter which helps protect against overfitting.
# The choice of value for this parameter is left up to you, so long as you can provide a justification (there are several sensible possibilities).
#
# The goal here is to make the best possible predictions you can of the held out hand velocity data in test trials, based on the neural activity in the same trials. When you are ready to test your predictions, you can submit them as a 3D numpy array of shape 2 x 100 x 16 to http://4G10.cbl-cambridge.org (note: http, not https). If you get a HTTP error 400 back, it probably means the format is wrong. Your numpy array must be saved using the np.save("filename.npy", my_array) function; the server also expects the array to be of float64 numerical type â  this should be the default in numpy, but if in doubt you can always cast using my_array.as_type("float64"). When you submit, please indicate your candidate number and choose "Simple Gaussian smoothing" in the dropdown list. Upon uploading, you will receive immediate feedback in the form of an R^2 coefficient. The closer to 1, the better!
#
# - <u>Implement Gaussian temporal smoothing + ridge regression as outlined above</u>.
# - **How does the quality of hand velocity predictions vary with the smoothing window length $\sigma$? How do you interpret that?** You might want to experiment with values between 20 and 80 ms.
# - **Comment on the suitability of this simple decoding strategy for online (on the fly) decoding of movement in a BMI context (consider e.g. feasability, computational tractability, and accuracy). Can you think of a small modification to the above approach that would improve applicability to online decoding?** (bonus points for  implementing it!)
# - The hand velocity data provided in data["hand_train"] had actually been shifted backward by 120ms relative to the neural data (and similarly for the test set, which was not given to you). **Can you speculate about why we did that**?

#
from scipy.ndimage import gaussian_filter1d

# Ridge regression
def ridge_regression(X, V, lambd):
    N = X.shape[0]
    XXt = X @ X.T
    reg_matrix = lambd * np.eye(N)
    W = V @ X.T @ np.linalg.inv(XXt + reg_matrix)
    return W

# Prepare data
sigma = 1.6  # Set Gaussian smoothing window length (20ms, sigma=0.4) (80ms, sigma=1.6)
lambda_reg = 75 # Regularization parameter

# Smooth training neural data
smoothed_neural_train = gaussian_filter1d(neural_train, sigma=sigma, axis=2)

# Reshape training data for regression
N, trials, T = smoothed_neural_train.shape
smoothed_neural_train_reshaped = smoothed_neural_train.reshape(N, trials * T)
hand_train_reshaped = hand_train.reshape(2, trials * T)

# Compute ridge regression weights
W_star = ridge_regression(smoothed_neural_train_reshaped, hand_train_reshaped,
    lambda_reg)
```

```python

# Predict on test data
smoothed_neural_test = gaussian_filter1d(neural_test, sigma=sigma, axis=2)
smoothed_neural_test_reshaped = smoothed_neural_test.reshape(N, -1)

# Predictions for test trials
predicted_hand_test = W_star @ smoothed_neural_test_reshaped
predicted_hand_test = predicted_hand_test.reshape(2, 100, T)

# Save predictions
np.save(f'hand_test_s={sigma}_l={lambda_reg}.npy', predicted_hand_test.astype("float64"))

#
from scipy.ndimage import gaussian_filter1d

# Ridge regression
def ridge_regression(X, V, lambd):
    N = X.shape[0]
    XXt = X @ X.T
    reg_matrix = lambd * np.eye(N)
    W = V @ X.T @ np.linalg.inv(XXt + reg_matrix)
    return W

# Gaussian smoothing function (incremental for online use)
def online_gaussian_smooth(data, sigma, t):
    """Applies Gaussian smoothing up to time step t."""
    # smoothed = gaussian_filter1d(data[:, :, :t+1], sigma=sigma, axis=2, mode='constant',cval=0)
    smoothed = gaussian_filter1d(data[:, :, :t+1], sigma=sigma, axis=2)
    return smoothed[:, :, t]  # Return the smoothed value at time t

# Prepare 5ata
sigma = 1.7  # Set Gaussian smoothing window length (20ms, sigma=0.4) (80ms, sigma=1.6)
lambda_reg = 75  # Regularization parameter

# Reshape training data for regression
N, trials, T = smoothed_neural_train.shape
predicted_hand_test_online = np.zeros((2, 100, T))  # To store online predictions
predicted_hand_train_online = np.zeros((2, 400, T))  # To store online predictions
smoothed_neural_test_online = np.zeros((N, 100, T))  # To store online predictions
smoothed_neural_train_online = np.zeros((N, 400, T))  # To store online predictions

## train online
# for t in range(T):  # Loop over each time step
#     smoothed_neural_train_online[:,:,t] = online_gaussian_smooth(neural_train, sigma, t)
# smoothed_neural_train_online_reshaped = smoothed_neural_train_online.reshape(N, 400 * T)
# hand_train_reshaped = hand_train.reshape(2, 400 * T)
# W_star = ridge_regression(smoothed_neural_train_online_reshaped, hand_train_reshaped, lambda_reg)

## train offline
W_star = ridge_regression(smoothed_neural_train_reshaped, hand_train_reshaped, lambda_reg)

for t in range(T):  # Loop over each time step
    # Smooth data incrementally up to time t
```

```
    smoothed_neural_test_online[:,:,t] = online_gaussian_smooth(neural_test,
       sigma, t)
    predicted_hand_test_online[:, :, t] = W_star @ smoothed_neural_test_online
       [:,:,t]

# Save online predictions
print()
# print(predicted_hand_test_online.shape)
np.save(f'hand_test_online_incremental_s={sigma}_l={lambda_reg}.npy',
    predicted_hand_test_online.astype("float64"))

print("Online prediction with incremental smoothing completed and saved.")

#
import numpy as np
import matplotlib.pyplot as plt
from scipy.ndimage import gaussian_filter1d

# Define the data and smoothing function
def online_gaussian_smooth(data, sigma, t):
    """Applies Gaussian smoothing up to time step t."""
    smoothed = gaussian_filter1d(data[:t+1], sigma=sigma, axis=0)
    return smoothed[t]  # Return the smoothed value at time t

# Initialize data
sigma = 1.0  # Example sigma value
test_zeros = np.zeros((16))
test_zeros[6] = 1

# Apply offline Gaussian smoothing
offline_smoothed = gaussian_filter1d(test_zeros, sigma=sigma)

# Apply online Gaussian smoothing
test_zeros_online = np.zeros((16))
for t in range(16):  # Loop over each time step
    test_zeros_online[t] = online_gaussian_smooth(test_zeros, sigma, t)

# Adjust x-axis values (multiply by 50)
time_steps = np.arange(16) * 50

# Plot the results
plt.figure(figsize=(3, 4))
plt.plot(time_steps, offline_smoothed, label='offline smoothed')
plt.plot(time_steps, test_zeros_online, label='online filtered')
plt.xlabel('Time (ms)')
plt.ylabel('Smoothed/Filtered Value')
plt.legend()
# plt.title('Comparison of Online and Offline Gaussian Smoothing')
plt.show()

#
import matplotlib.pyplot as plt
import numpy as np

# Data
sigma_offline = [20.0, 25.0, 30.0, 40.0, 50.0, 55.0, 60.0, 65.0, 70.0, 75.0,
    80.0]
r2_offline = [0.345, 0.375, 0.401, 0.428, 0.441, 0.444, 0.445, 0.445, 0.443,
    0.440, 0.437]

sigma_online_trained_offline =  [20.0, 25.0, 30.0, 40.0, 45.0, 50.0, 55.0, 60.0,
    65.0, 70.0, 75.0, 80.0, 85.0]
r2_online_trained_offline = [0.145, 0.202, 0.248, 0.307, 0.328, 0.345, 0.358,
```

```
0.367, 0.378, 0.378, 0.380, 0.380, 0.378]

sigma_online_trained_online = [20.0, 25.0, 30.0, 40.0, 50.0, 55.0, 60.0, 65.0, 70.0, 75.0, 80.0, 90.0, 95.0, 100.0]
r2_online_trained_online = [0.337, 0.353, 0.368, 0.391, 0.409, 0.416, 0.421, 0.426, 0.430, 0.433, 0.434, 0.436, 0.436, 0.435]

# Plot
plt.figure(figsize=(6, 4))

plt.plot(sigma_offline, r2_offline, label="Offline", marker='o', linestyle='-', linewidth=2)
plt.plot(sigma_online_trained_offline, r2_online_trained_offline, label="Online, trained offline", marker='s', linestyle='--', linewidth=2)
plt.plot(sigma_online_trained_online, r2_online_trained_online, label="Online, trained online", marker='^', linestyle='-.', linewidth=2)

# Labels and legend
# plt.title("R2 vs $\sigma$ Values", fontsize=14)
plt.xlabel("Smoothing Window $\sigma$ (ms))", fontsize=12)
plt.ylabel("$R^2$ Score", fontsize=12)
plt.legend(loc="best", fontsize=10)
plt.grid(True)

# Show plot
plt.tight_layout()
plt.show()

#  [markdown]
# # 3. Kalman filter-based decoding
#
# We now turn to a more sophisticated decoder based on a Kalman filter/smoother.
#
# ### 3.1 An autoregressive prior for hand kinematics
#
# A 10-dimensional linear latent dynamical system (LDS; cf lecture notes) was pre-trained for you on the hand velocity data in the training set; specifically, we consider the following generative model:
#
# $$
# (1) \qquad z_{k, 0} \sim \mathcal{N}(\mu_0, \Sigma_0) \\
# (2) \qquad z_{k, t+1} = A z_{k, t} + \epsilon_{k, t+1} \quad \text{with } \epsilon_{k, t+1} \sim \mathcal{N}(0, Q) \\
# (3) \qquad v_{k, t} = C z_{k, t} + \eta_{k, t} \quad \text{with } \eta_{k, t} \sim \mathcal{N}(0, R)
# $$
#
# where $z_{k, t} \in \mathbb{R}^{10}$ is the latent state in time bin $t$ of trial $k$, and $v_{k, t} \in \mathbb{R}^2$ is the corresponding hand velocity.
#
# The parameters of this LDS can be found in the same data dictionary as above, with the following keys:
# - "hand_KF_A" (10 x 10): state matrix $A$
# - "hand_KF_C" (2 x 10): output matrix $C$
# - "hand_KF_mu0" (10 x 1): initial prior mean $\mu_0$
# - "hand_KF_Sigma0" (10 x 10): initial prior covariance $\Sigma_0$
# - "hand_KF_Q" (10 x 10): process noise covariance matrix $Q$
# - "hand_KF_R" (2 x 2): observation noise covariance matrix $R$
#
# <u>Write your own Kalman smoother implementation and use it to compute the mean $\hat{z}_{k, 1:T}$ of the smoothing distribution $p(z_{k, t} | v_{k, 1:T})$, for each trial $k$ in the training set.</u>
#
```

```python
#
# Retrieve parameters
A = data["hand_KF_A"]
C = data["hand_KF_C"]
mu0 = data["hand_KF_mu0"]
Sigma0 = data["hand_KF_Sigma0"]
Q = data["hand_KF_Q"]
R = data["hand_KF_R"]

#
print(mu0.shape)

#
import numpy as np


def kalman_filter(x, A, C, mu0, Sigma0, Q, R):
    """
    Kalman filter for the given LDS.

    Parameters:
        x: Observations (T x 2)
        A: State transition matrix (10 x 10)
        C: Observation matrix (2 x 10)
        mu0: Initial mean (10 x 1)
        Sigma0: Initial covariance (10 x 10)
        Q: Process noise covariance (10 x 10)
        R: Observation noise covariance (2 x 2)

    Returns:
        filtered_means: Smoothed means (T x 10)
        filtered_covariances: Smoothed covariances (T x 10 x 10)
    """
    T = x.shape[0]
    d = mu0.shape[0]  # Dimensionality of latent state

    # Initialize arrays
    filtered_means = np.zeros((T, d))
    filtered_covariances = np.zeros((T, d, d))

    # Forward pass: Kalman filter
    filtered_means[0] = mu0.ravel()
    filtered_covariances[0] = Sigma0

    for t in range(1, T, 1):
        Pt = A @ filtered_covariances[t-1] @ A.T + Q
        Kt = Pt @ C.T @ np.linalg.solve(C @ Pt @ C.T + R, np.eye(R.shape[0]))
        filtered_means[t] = A @ filtered_means[t-1] + Kt @ (x[t] - C @ A @
            filtered_means[t-1])
        filtered_covariances[t] = Pt - Kt @ C @ Pt

    return filtered_means, filtered_covariances

def kalman_smoother(x, A, C, mu0, Sigma0, Q, R):
    """
    Kalman smoother for the given LDS.

    Parameters:
        x: Observations (T x 2)
        A: State transition matrix (10 x 10)
        C: Observation matrix (2 x 10)
        mu0: Initial mean (10 x 1)
```

```python
        Sigma0: Initial covariance (10 x 10)
        Q: Process noise covariance (10 x 10)
        R: Observation noise covariance (2 x 2)

    Returns:
        smoothed_means: Smoothed means (T x 10)
        smoothed_covariances: Smoothed covariances (T x 10 x 10)
    """
    T = x.shape[0]
    d = mu0.shape[0]  # Dimensionality of latent state

    # Forward pass: Kalman filter
    filtered_means, filtered_covariances = kalman_filter(x, A, C, mu0, Sigma0, Q,
        R)

    # Backward pass: Kalman Smoother
    smoothed_means = np.zeros((T, d))
    smoothed_covariances = np.zeros((T, d, d))

    smoothed_means[-1] = filtered_means[-1]
    smoothed_covariances[-1] = filtered_covariances[-1]

    for t in range(T-2, -1, -1):
        Pt = A @ filtered_covariances[t] @ A.T + Q
        Gt = filtered_covariances[t] @ A.T @ np.linalg.solve(Pt, np.eye(Pt.shape
            [0]))
        smoothed_means[t] = filtered_means[t] + Gt @ (smoothed_means[t+1] - A @
            filtered_means[t])
        smoothed_covariances[t] = filtered_covariances[t] + Gt @ (
            smoothed_covariances[t+1] - Pt) @ Gt.T

    return smoothed_means, smoothed_covariances

def kalman_all_trials(X, A, C, mu0, Sigma0, Q, R, mode='filter'):
    # Velocity data shape: (N x 400 x 16)
    # Output: (400 x 16 x 10)
    d = A.shape[0]
    n_trials = X.shape[1]
    n_timesteps = X.shape[2]

    # Initialize a container for smoothed means
    all_kalmaned_means = np.zeros((d, n_trials, n_timesteps))  # (Latent Dim,
        Trials, Timesteps)
    all_kalmaned_covs = np.zeros((d, d, n_trials, n_timesteps))  # (Latent Dim,
        Latent Dim, Trials, Timesteps)

    # Loop over each trial
    if mode=='filter':
        for trial_idx in range(n_trials):
            x_trial = X[:, trial_idx, :].T  # Shape becomes (16 x 2)
            kalmaned_means, kalmaned_covs = kalman_filter(x_trial, A, C, mu0,
                Sigma0, Q, R)
            all_kalmaned_means[:, trial_idx, :] = kalmaned_means.T
            all_kalmaned_covs[:, :, trial_idx, :] = np.transpose(kalmaned_covs,
                axes=(1,2,0))
    elif mode=='smoother':
        for trial_idx in range(n_trials):
            x_trial = X[:, trial_idx, :].T  # Shape becomes (16 x 2)
            kalmaned_means, kalmaned_covs = kalman_smoother(x_trial, A, C, mu0,
                Sigma0, Q, R)
            all_kalmaned_means[:, trial_idx, :] = kalmaned_means.T
            all_kalmaned_covs[:, :, trial_idx, :] = np.transpose(kalmaned_covs,
                axes=(1,2,0))
```

```
    else:
        error
    return all_kalmaned_means, all_kalmaned_covs

print(hand_train.shape)
smoothed_latent_means, _ = kalman_all_trials(hand_train, A, C, mu0, Sigma0, Q, R, mode='smoother')
# smoothed_latent_means, _ = kalman_all_trials(hand_train, A, C, mu0, Sigma0, Q, R, mode='filter')
print(smoothed_latent_means.shape)



# [markdown]
# ### 3.2 Building an LDS model of neural data using supervised learning
#
# Conceptually, the latents $z_{k, 1:T}$ introduced above contain signals related to the velocity of the hand, its acceleration, and potentially higher-order derivatives too â  all signals which we have good reasons to suspect that neural activity in M1 is strongly related to. Eqs (1) and (2) above provide a good autoregressive prior model for the temporal dynamics of these signals, and you are now going to use this prior in a generative LDS model of _neural data_, substituting the hand-related likelihood (Eq 3) with a neural likelihood:
#
# $$
# (4) \qquad x_{k, t} = D z_{k, t} + \xi_{k, t} \quad \text{with } \xi_{k, t} \sim \mathcal{N}(0, S)
# $$
#
# where $x_{k, t}$ denotes neural spike counts in the $t^\text{th}$ time bin of trial $k$.
#
# The combination of Eqs (1), (2) and (4) forms an LDS model which you will be able to invert using Kalman filtering to obtain a filtered posterior $p(z_{k', t} | x_{k', 0:t})$ for any test trial $k'$. From there, you will use Eq. (3) to obtain a filtered predictive distribution for the hand velocity in each test trial, $p(v_{k', t} | x_{k', 0:t})$.
#
# - <u>Begin by centering the neural data (both training and testing sets) by removing, for each neuron, its mean activity across both trials and time in the training set.</u>
# - <u>Fit the likelihood parameters $D$ and $S$ through supervised learning, by maximizing the joint log-likelihood $\log p(\hat{z}_{k, 1:T}, x_{k, 1:T})$ averaged over all trials in the training set, where $\hat{z}_{k,1:T}$ is the posterior mean you obtained in Section 3.1.</u>
# To do this, write down the average joint log likelihood and **show that it is maximized by the following parameter settings**:
#     - $ \displaystyle D^\star = \left(\sum_{k, t} x_{k, t} \hat{z}_{k, t}^\top \right) \left( \sum_{k, t} \hat{z}_{k, t} \hat{z}_{k, t}^\top \right)^{-1} $
#     - $ \displaystyle S^\star = \frac{1}{KT} \left( \sum_{k, t} x_{k, t} x_{k, t}^\top - D^\star \sum_{k, t} \hat{z}_{k, t} x_{k, t}^\top \right) $ where $K$ is the number of trials in the training set. **Include your derivations in your report.**

#
import numpy as np

def center_neural_data(neural_train, neural_test):
    """
    Centers neural data by subtracting the mean activity of each neuron across all trials and time bins in the training set.

```

```
    Parameters:
        neural_train: Training neural data (N x K x T)
        neural_test: Testing neural data (N x K' x T)

    Returns:
        centered_train: Centered training data (N x K x T)
        centered_test: Centered testing data (N x K' x T)
    """
    # Compute mean activity across trials and time bins
    mean_activity = np.mean(neural_train, axis=(1, 2), keepdims=True)

    # Subtract mean from both training and testing sets
    centered_train = neural_train - mean_activity
    centered_test = neural_test - mean_activity

    return centered_train, centered_test

def fit_likelihood_params(X, smoothed_means):
    """
    Fits the parameters D and S using the provided formulas.

    Parameters:
        X: Centered training neural data (N x K x T)
        smoothed_means: Posterior means of latent states z (d x K x T)
    Returns:
        D_star: Fitted observation matrix (N x d)
        S_star: Fitted observation noise covariance (N x N)
    """
    # Extract dimensions
    N, K, T = X.shape  # N = neurons, K = trials, T = timesteps
    d, _, _ = smoothed_means.shape  # d = latent dimensionality

    # Initialize accumulators with correct dimensions
    sum_xzT = np.zeros((N, d))  # Shape (N x d)
    sum_zzT = np.zeros((d, d))  # Shape (d x d)
    sum_xxT = np.zeros((N, N))  # Shape (N x N)

    # Iterate over trials and time bins
    for k in range(K):
        for t in range(T):
            # Extract data for time t and trial k
            x_kt = X[:, k, t].reshape(-1, 1)  # Shape (N x 1)
            z_kt = smoothed_means[:, k, t].reshape(-1, 1)  # Shape (d x 1)

            # Accumulate sums
            sum_xzT += x_kt @ z_kt.T  # Outer product (N x 1) @ (1 x d) -> (N x d)
            sum_zzT += z_kt @ z_kt.T  # Outer product (d x 1) @ (1 x d) -> (d x d)
            sum_xxT += x_kt @ x_kt.T  # Outer product (N x 1) @ (1 x N) -> (N x N)

    # Compute D_star
    D_star = sum_xzT @ np.linalg.solve(sum_zzT, np.eye(d))  # Shape (N x d)
    # Compute S_star
    S_star = (sum_xxT - D_star @ sum_xzT.T) / (K * T) # Shape (N x N)

    return D_star, S_star


centered_neural_train, centered_neural_test = center_neural_data(neural_train, neural_test)

```

```python
# Fit the likelihood parameters
D_star, S_star = fit_likelihood_params(centered_neural_train, smoothed_latent_means)
# plt.figure()
# plt.imshow(D_star)
# plt.figure()
# plt.imshow(S_star)


#  [markdown]
# ### 3.3 Using Kalman filtering to predict the hand velocity
#
# - Based on the model obtained in Section 3.2, <u>write your own Kalman filter implementation and compute the filtered posterior $p(z_{k', t} | x_{k', 0:t})$ for each trial $k'$ in the test set.</u> What we are really interested in is the mean $\bar{z}_{k', t}$ of this filtered posterior, as our best prediction of the momentary hand velocity $v_{k', t}$ is then given by $\hat{v}_{k', t} = C \bar{z}_{k', t}$.
# - <u>Submit your predictions to http://4G10.cbl-cambridge.org (note: http, not https) in the same format as described in Section 2. Please select "Kalman filtering" in the dropdown list. Once again, you will receive immediate feedback in the form of an $R^2$ coefficient.</u> **Include this result in your report, and discuss; in particular, why do you think these predictions are much better than those of Section 2?**
# - As in Section 2, **comment on the suitability of this more sophisticated decoding strategy for online decoding of movement in a BCI context**.
# - Finally, **what approach(es) would you suggest to improve decoding performance even further? Include a discussion of the tradeoffs that would arise with these alternative approaches.** (max 1 page in your report)

#
# filtered_latent_means, _ = kalman_all_trials(centered_neural_train, A, D_star, mu0, Sigma0, Q, S_star, mode='filter')
filtered_latent_means, _ = kalman_all_trials(centered_neural_test, A, D_star, mu0, Sigma0, Q, S_star, mode='filter')
v_pred = np.einsum('ij,jkl->ikl', C, filtered_latent_means)

# np.save(f'hand_test_online_incremental_s={sigma}_l={lambda_reg}.npy', predicted_hand_test_online.astype("float64"))
np.save(f'hand_test_kalman.npy', v_pred.astype("float64"))



#
filtered_latent_means, _ = kalman_all_trials(centered_neural_train, A, D_star, mu0, Sigma0, Q, S_star, mode='filter')

idx1 = 4
idx2 = 60
plt.plot(smoothed_latent_means[idx1,idx2,:], label='from v')
plt.plot(filtered_latent_means[idx1,idx2,:], label='from x')
plt.legend()

#  [markdown]
# # Writing up
#
#
# Please write up your findings in a report to be submitted on Moodle in PDF format, and **include all your code in the Appendix**. Please clearly include your candidate number, NOT your name, on the front page. Your report should address all the questions raised in this notebook, be structured around the Sections of this notebook, and **be a maximum of five A4 pages** excluding any Appendix (minimum font size 11pt, minimum margins 1.5cm on each side).
```

```
#
# You are very much encouraged to think of data/results visualisations to best
    support the exposition of your results. You are also encouraged to report on
    any specific problems/difficulties that arose in your implementation of the
    various algorithms, and how you addressed those.
```