Decoding of Monkey Arm Movements: A Study Using kNN and Kalman Filters

Josephine Cao
CID: 02546684
jc8823@ic.ac.uk

Jiayu Liu
CID: 02439119
jl5223@ic.ac.uk

Xinyi Liu
CID: 02473070
xl3123@ic.ac.uk

Fangyuan Wang
CID: 02518739
fw1123@ic.ac.uk

Abstract

This study investigates the decoding of a monkey's arm movement trajectories using neural spike data. We evaluated four decoding models: Linear Discriminant Analysis (LDA), k-nearest Neighbour (kNN), Linear Regression (LR), and Kalman Filter, to identify the most effective method. The evaluation involved a detailed performance analysis using root mean squared error (RMSE) with respect to the actual movement trajectories. Among the models tested, the kNN and Kalman filters performed the best, achieving an RMSE of 15.51cm and runtime of 17.62 seconds.

Keywords: BMI, decoder, kNN, Kalman filter

I. Introduction

In the current field of neuroscience and robotics, accurately decoding and predicting the motor intentions of animals has become a challenge with significant scientific and applied value. Especially in brain-machine interface (BMI) technology, efficient and accurate decoding of movement trajectories is of crucial importance for improving the quality of life of people with disabilities and enhancing the naturalness of human-computer interaction. This study is dedicated to exploring the optimal motion decoding strategy to predict the direction of arm movements in monkeys based on neural spike data recorded from different brain regions.

In order to select the optimal decoder model, we compare several commonly used statistical and machine learning methods, including Linear Discriminant Analysis (LDA), k-Nearest Neighbour (kNN), Linear Regression (LR) and Kalman Filter. Each method exhibits different strengths and limitations in terms of its ability to handle spatio-temporal data, its efficiency in handling high-dimensional data, its sensitivity to outliers, and its ability to generalise the model. Through detailed performance evaluations and comparative analyses, we aim to determine which method performs best in accurately decoding neural spiking data, thereby providing a guide on how to implement these techniques in real BMI systems.

In addition, the study involves the optimisation and validation of the chosen decoder to ensure reliability and efficiency in a real-world environment.

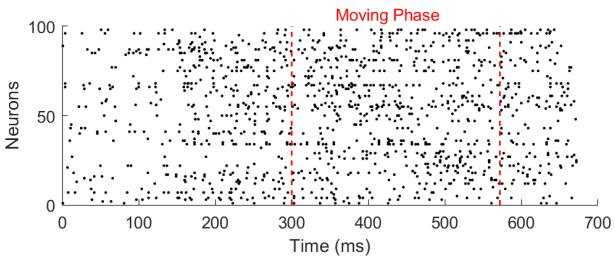


Figure 1: Raster plot for a single trial

I.A. Data Overview

The data comprises firing events from 98 neurons across 100 trials for 8 reaching directions, spanning from 300 ms before to the movement onset to 100 ms after its end. The neural activity is encoded in a binary format, where '0' indicates the absence of a spike, and '1' signifies a firing event.

The raster plot, as shown in figure1 illustrates a single neuron's activity along the time axis where each row represents a single neuron. The red dotted lines mark the arm moving phase, during which there is an increase in neuronal firing. Actually, neuronal activity become progressively active before the movement, possibly indicating a preparatory phase for the movement. The plot also shows the varied activity levels among neurons, with some exhibiting frequent spikes while others show fewer. This may reflect the different roles of neurons associated with different directions of movement.

To explore this difference even further, the tuning curves of certain neurons (neuron 65, neuron 73 and neuron 75) are drawn in figure A2. Different neurons exhibit preferences for specific directions of movement, suggesting that certain neurons are specialized for processing specific types of motor actions. Conversely, some neurons show no preference for any direction, indicating a more generalized or

supportive role in the neural network handling movement.

Given the observed preferences, it becomes important to classify the directional tendencies of neurons and then train regression models to predict arm velocities. This combined approach ensures that the neural signals are accurately translated into arm trajectories.

I.B. Data Preparation and Feature Extraction

Data Preparation: Data preparation are performed before analysis to keep a uniformed data length and avoid the loss of information at the end of the sequence. This is achieved by padding shorter sequences with zeros to match the length of the longest data. Meanwhile, the continuous neural and hand position data are segmented into smaller time windows, which facilitates the feature extraction.

Feature Extraction: Critical features are extracted after the data preparation. The firing rate for each neuron is calculated by dividing the total number of spikes by the length of each respective window (in seconds), which are representative of the neural activity for the specific intervals. Moreover, kinematic data such as hand position and velocity for each time window are assessed. Since the kinematic data are the same for all neurons within the same time window, the extraction of kinematic features is performed only at the first neuron for each time window for each direction in order to avoid repeated calculations.

II. Methods

This section outlines the features and parameter settings of classifiers and regressors. The data analysis of a training and testing split ratio is 50:50.

II.A. Classifier

II.A.1 Linear Discriminant Analysis

LDA is used to predict initial direction of the monkey's arm. The training data window size is set to 20 ms and then the change in position within each window period is calculated along with the pulse training data within the same window period. For predicting the angle of arrival, the sum of the number of pulses within the first 340 ms is processed using LDA. This involves recombining the pulse data by neuron and trial to create a feature matrix with each row representing a trial and each column representing the total number of pulses for a neuron within that particular time window. A linear discriminant analysis model is trained using the fitcdiscr function, which is able to predict the direction of the action based on the pulse features.

II.A.2 k-Nearest Neighbors

kNN is a simple, yet powerful algorithm used in data mining and machine learning. As shown in figure 2, it classifies a data point based on how its neighbours are classified. This project focuses on using kNN to predict the direction of arm movements based on neural spike counts from the first 320 ms of trials. This model assumes that neural activity patterns within this brief initial period are indicative of movement direction, which remains consistent in the short term. A formal definition of kNN can be found in the appendix B.

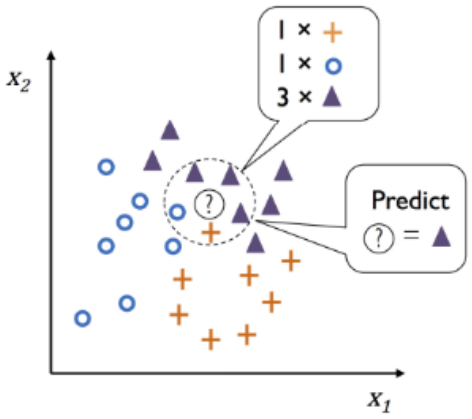


Figure 2: Illustration of kNN for a 3 class problem with k=5.[4]

II.B. Regressor

II.B.1 Linear Regression

Linear regression is a fundamental statistical method to model the relationship between variables. In this study, linear regression of firing rates and the velocity in the x and y directions, respectively, is performed using least-paradigm least squares (lsqminnorm) to obtain coefficients for predicting velocity. For each angle (direction of arrival), a matrix of coefficients in the x and y directions is calculated and stored in the modelParameters structure.)

II.B.2 Kalman Filter

Kalman filter is used to decode the neural activity in a real-time fashion. The real-time Kalman algorithm is adapted from [7], where the recursive Bayesian filter is used to map the firing rate feature into hand positions.

$$\mathbf{z}_k = \mathbf{H}_k \mathbf{x}_k + \mathbf{q}_k \tag{1}$$

$$\mathbf{x}_{k+1} = \mathbf{A}_k \mathbf{x}_k + \mathbf{w}_k \tag{2}$$

Where $\boldsymbol{z_k}$is the measurements of time stamp $\boldsymbol{k}$, $\boldsymbol{x_k}$ is the state representation, $\boldsymbol{H_k}$ is the transition matrix from state to measurements, and $\boldsymbol{A_k}$ is the transition from the previous state to the current state. $\boldsymbol{q_k}$ and $\boldsymbol{w_k}$ are the noise covariance representing the uncertainty of the transition.

As the Kalman filter adopts the linear model, the practice divides the overall kinematics into 8 directions. The kinematics of each direction are assumed to be linear. This eliminates the non-linear estimation errors caused by the kinematic discontinuity of changing hand-moving directions.

II.B.3 Extended Kalman

Compared to the Kalman filter, the state transition and observation matrix of the extended Kalman filter are not necessarily linear but may instead be differentiable functions $\boldsymbol{h}$ and $\boldsymbol{f}$.

$$\mathbf{z}_k = \mathbf{h}(\mathbf{x}_k) + \mathbf{v}_k \tag{3}$$

$$\mathbf{x}_{k+1} = \mathbf{f}(\mathbf{x}_k, \mathbf{u}_k) + \mathbf{w}_k \tag{4}$$

Where $\boldsymbol{v_k}$ and $\boldsymbol{w_k}$ are the noise covariance of the transition.

Afterwards, the derivatives of the kinematics model are used to make predictions in the extended Kalman filter. In this practice, the transition function is assumed to be the linear Kalman filter transition matrix, which minimizes the prediction error.

III. Results

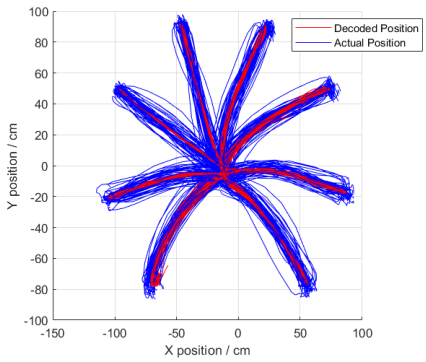


Figure 3: Decoded and actual monkey arm trajectories using kNN and Kalman

Table 1: Decoding methods and their performance

Decoding method	RMSE	Time	Score
kNN+LR(ref)	22.25	34.43	23.468
LDA+LR(ref)	27.31	221	46.679
kNN+Kalman constant parameters	15.84	13.42	15.598
kNN+Kalman adaptive parameters	15.51	21.68	16.127
kNN+Extended Kalman	18.66	20.09	18.803

Comparing the scores, calculated as a weighted combination of RMSE (0.9) and Time (0.1), revealed that the kNN+Kalman with constant parameters method yielded the best performance. As shown in Table 1, this method achieved the lowest score of 15.598, with an RMSE of 15.84 and a Time of 13.42. Given its superior balance of accuracy and computational efficiency, the kNN+Kalman with constant parameters method was selected as the final decoder. Among the decoding methods, linear regression is a ready-made program[1], which is referenced here for comparison purposes.

IV. Discussion

LDA: LDA uses linear projections to improve class separation by maximizing distances between classes while reducing variance within the same class. Since spike data usually exhibits strong correlations for the same movement directions, this contributes to effective classification outcomes, as illustrated in A6. However, due to the prolonged classification times associated with LDA, we have opted for kNN instead.

kNN: The optimal selection process for k In the kNN model involved analyzing 100 trials, with 50 used for training and the other 50 for testing. Given the 8 directions to predict in each of the 50 trials, the total number of predictions evaluated was 400.

By examining the validation errors across k values from 1 to 400, as shown in figure A3, it was observed that the lowest error rates occurred when k was within the range of 1 to 80.

In the range of k= 1-80 as depicted in the figure A4, the lowest validation errors are observed around the k values of 10, 20, 30, 50 and 60. Considering the balance between minimizing error and ensuring computational efficiency, k=10 is chosen for its combination of low error rates and relatively quick runtime. The confusion chart for k = 10 (figure A5) demonstrates high accuracy and an average F1 score of 0.95, indicating that the classifier performs well in distinguishing between the different classes.

One potential problem with the k-NN technique is that it is biased toward classes that have the most examples in the training data. A variant of the technique addresses this problem by taking into account the distance from the input to each of the k-nearest neighbors and using an inverse-distance weighted average of the class predicted by the k-NN[3].

LR: When decoding arm movement trajectories, LR provided less accurate and efficient predictions compared to Kalman filtering, as it struggled to handle noise and uncertainty in the data. This resulted in less accurate tracking of dynamic movement. Additionally, the predicted trajectories obtained using Kalman filtering were noticeably smoother than those from LR (Figure 3). Therefore, we chose to use

Kalman filtering for regression.

Kalman: Compared to using fixed Kalman parameters, parameters can be recursively calculated for every non-overlapping 20ms time window. However, the computational complexity for training the adaptive Kalman decoder is increased by the number of 20ms time windows. Therefore, the theoretical computing time required for the training is 14 times of the original algorithm. Nevertheless, as the Kalman prediction accumulates errors for each time step, using adaptive Kalman parameters breaks the continuous accumulation of estimation error and increases the decoding accuracy.

Due to the complexity of biological data, the linear model is usually insufficient to describe the actual biological activity. From linear regression to the Kalman method, the decoding accuracy witnessed a great increase. This increase proved the complexity of both the neural activity and the hand movement trajectory.

From Figure.A7a to Figure.A7b, it can be seen that the estimated hand position trajectories are smoother. Thus, the additional first derivative of the hand kinematics incorporated into the extended Kalman filter increases the space continuity of the estimation. This continuous estimation increases the model's adaptability to challenging real-world situations.

V. Future Developments

V.A. Increasing the signal-noise ratio

From the introduction section I.A, it is known that certain neurons have low firing rates for all trails. For example, neuron No.52 from the Figure.A8. Also, Kalman input matrices that have ranks smaller than the dimensions cause Kalman parameters to diverge. This introduces noise to the model estimation. Therefore, these silent neurons can be termed as producing noisy signals. While the current decoding framework utilises all 98 neurons, it is useful to eliminate the noisy neuron information. For example, principle component analysis is a method to focus on the most significant neuron information.

V.B. Angular Nonlinearity

From Figure.A9a, the actual trajectory of each trial occupies a larger area in the x,y plane, while the decoded position is narrower and more linear. This phenomenon could be caused by ignoring the non-linearity of the hand trajectory in the angular plane. Therefore, encoding angular information of the hand trajectory and generating a Cartesian-Angular kinematic feature might help increase the decoding accuracy.

In addition, the practice of the Kalman filter requires a knowledge of the kinematic model. In occasions of unknown environment model could be challenging for the design. Thus, using deep neural networks might help to capture the nonlinear characteristics of the decoded model while not needing a description of the environment model.[2]. However, for this practice, DNN might not be the best choice, as the training of DNN could be slow. While fast decoding is one of the most important requirements of this practice.

V.C. Frequency Deconstruction for Better Classification Results

The classification result could be improved to increase the decoding accuracy. It is known that EEG-based BMI could be noisy [6], containing both brain and non-brain signals. Also, neural signals from different frequency bands could represent different brain functions. Therefore, analysing the signal in the frequency domain could be useful. For example, the study showed that the wavelet [5] techniques could increase the decoding accuracy in certain practices. Through frequency deconstruction, a clearer feature responsible for hand movement direction control could reveal itself.

VI. Authors contributions

JC wrote code for kNN classifiers & wrote kNN parts in the report. **JL** wrote the code for the LDA classification & wrote the introduction, LDA and conclusion sections of the report. **XL** wrote code for the pre-analysis of the data and the customized kNN function & wrote abstract, data overview, data preparation and feature extraction. **FW** wrote code for Kalman and extended Kalman filter & contributed to Kalman and future development sections.

References

[1] BMIYF. https://github.com/nickcsimp/bmiyf, 2024.

[2] S. Nagel and M. Spüler. World's fastest brain-computer interface: Combining eeg2code with deep learning. *PloS one*, 14(9):e0221909–e0221909, 2019.

[3] R. P. N. Rao. *Machine Learning*, page 71–98. Cambridge University Press, 2013.

[4] S. Raschka. Stat 479: Machine learning lecture notes, 2018.

[5] N. Robinson, A. P. Vinod, K. K. Ang, K. P. Tee, and C. T. Guan. Eeg-based classification of fast and slow hand movements using wavelet-csp algorithm. *IEEE transactions on biomedical engineering*, 60(8):2123–2132, 2013.

[6] J. R. Wolpaw and E. W. Wolpaw. *Brain-computer interfaces : principles and practice*. Oxford University Press, Oxford ;, 2012.

[7] W. Wu, A. Shaikhouni, J. Donoghue, and M. Black. Closed-loop neural control of cursor motion using a kalman filter. In *The 26th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, volume 2, pages 4126–4129. IEEE, 2004.

A. Appendix - Supplementary plots

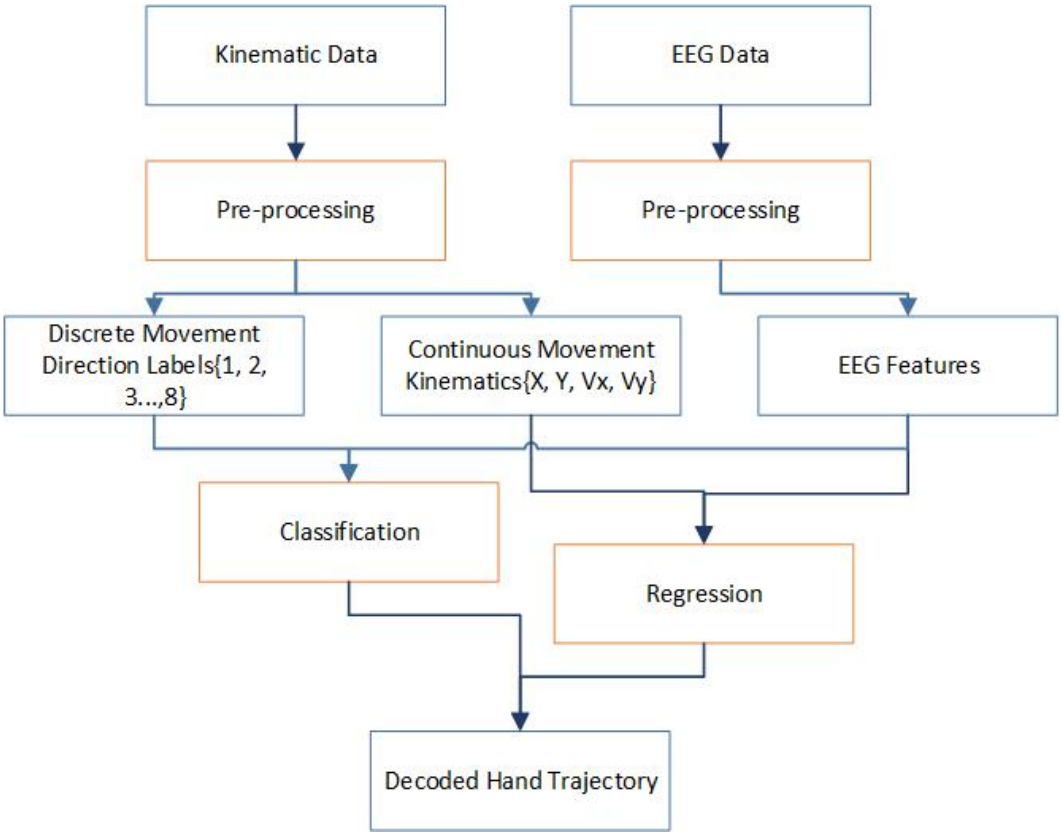


Figure A1: Overall framework of the design

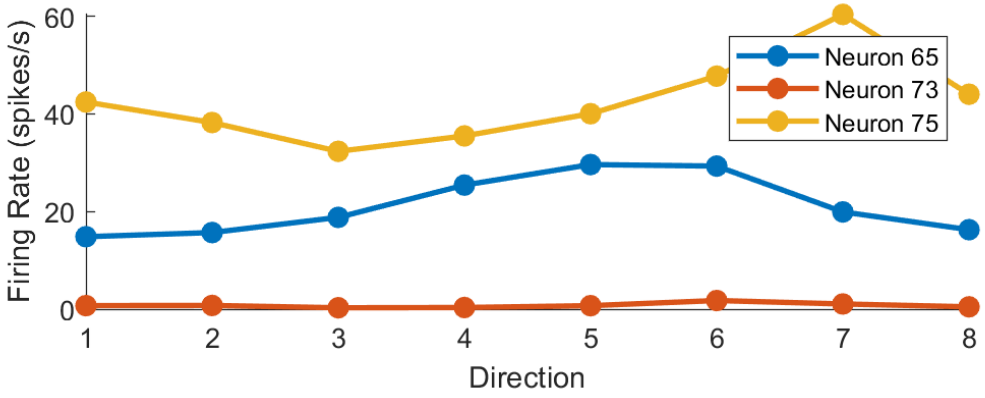


Figure A2: Tuning curves of neurons (neuron 65, neuron 73 and neuron 75)

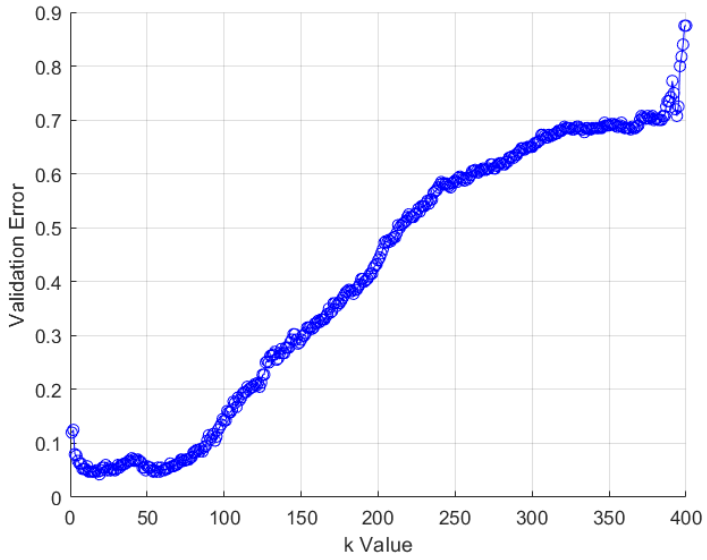


Figure A3: Validation error as a function of k in kNN algorithm (k=1-400)

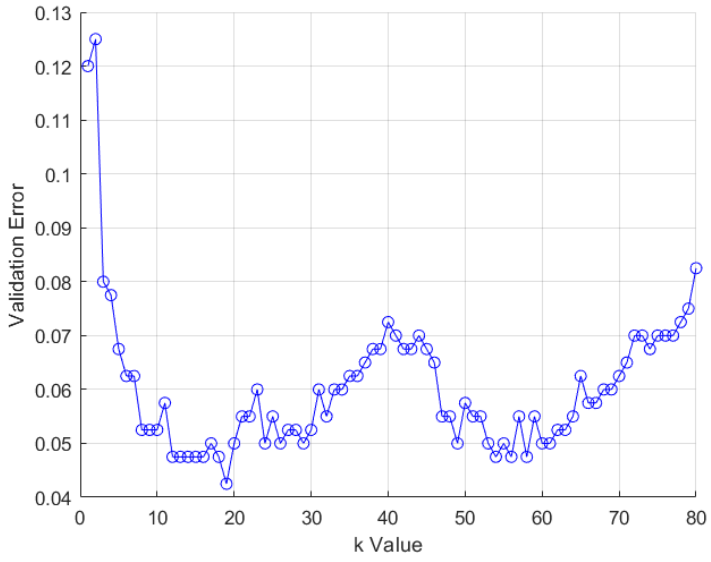


Figure A4: Validation error as a function of k in kNN algorithm (k=1-80)

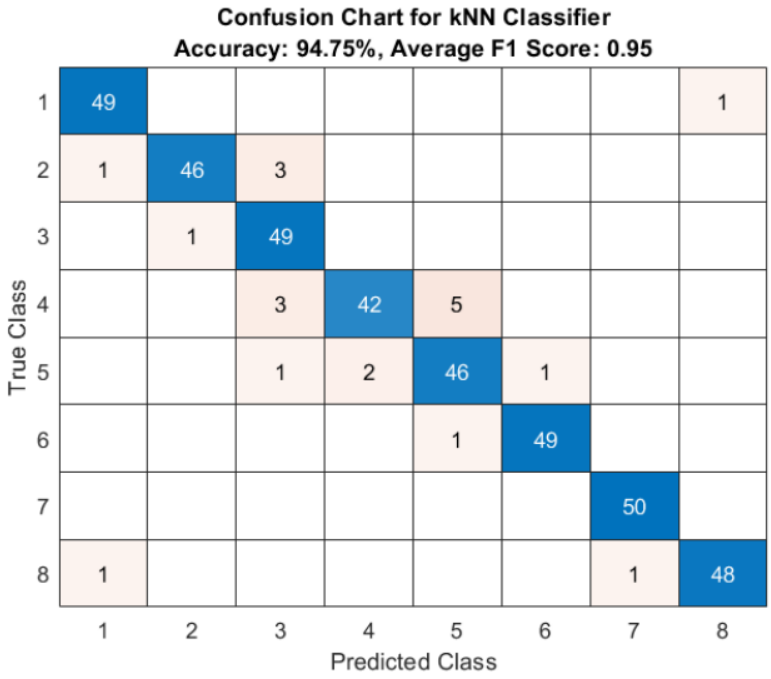


Figure A5: Confusion chart of kNN classifier (k=10). Figure shows true class labels on the vertical axis and predicted class labels on the horizontal axis. The diagonal cells indicate correct predictions, while off-diagonal cells show misclassifications. The classifier achieved an accuracy of 0.9475 and an average F1 score of 0.95, indicating accurate classification with very few misclassifications. Notably, the most frequent error occurred with direction 4 being misclassified as direction 5.

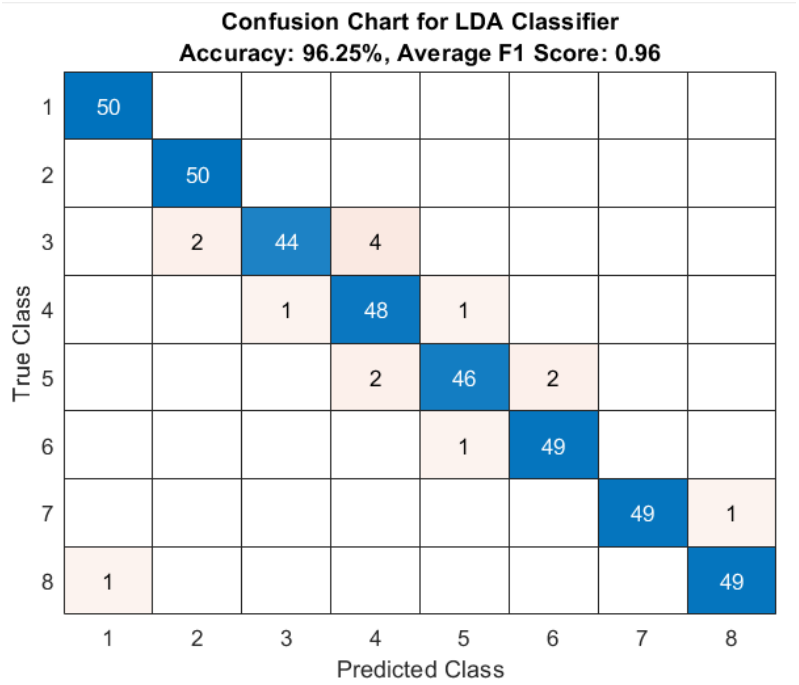
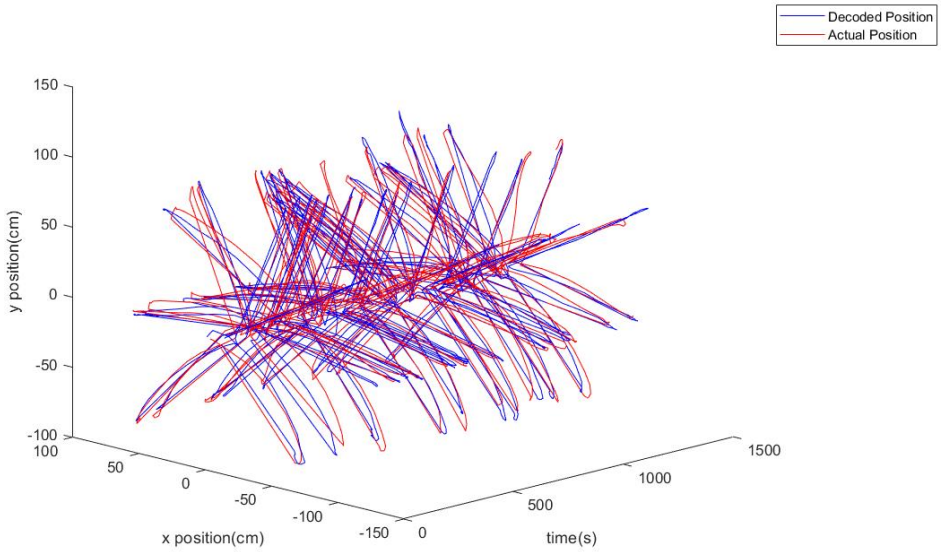
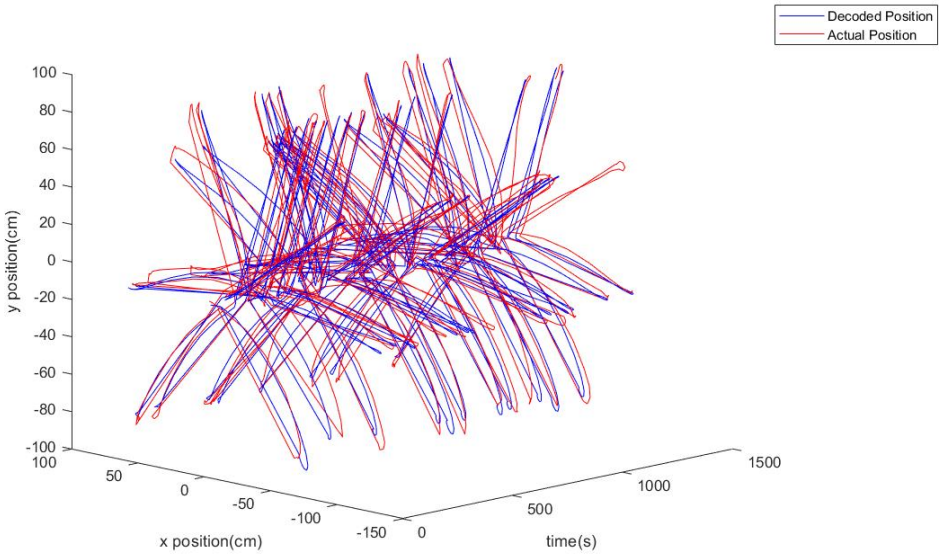


Figure A6: Confusion chart of LDA classifier. Figure illustrates an accuracy rate of 0.9625, indicating that the majority of predictions correctly classified the actual direction of movement. The average F1 score of 0.96 signifies that the model not only accurately identified the correct directions but also rarely mislabeled one direction as another. For directions 1 and 2, the model achieved perfect prediction accuracy, with each direction correctly predicted 50 times without any errors. Misclassifications in other directions may be due to the neural spike data exhibiting similar patterns across certain directions, making it challenging for the model to differentiate among these similar features. Additionally, the linear nature of the LDA model might limit its ability to handle complex features.



(a) Kalman Filter Decoding Results



(b) Extended Kalman Filter Decoding Results

Figure A7: Figure illustrates the decoded trajectory and actual trajectory in the 3D space using different methods, where the additional dimension is given by the time axis. Compared to (a), (b) Extended Kalman Filter generates a smoother trajectory, where fewer movement discontinuities are given. Therefore, extended Kalman generates a more realistic prediction. Also, the trajectory end-point limits are better for (b) than (a) Kalman Filter results.

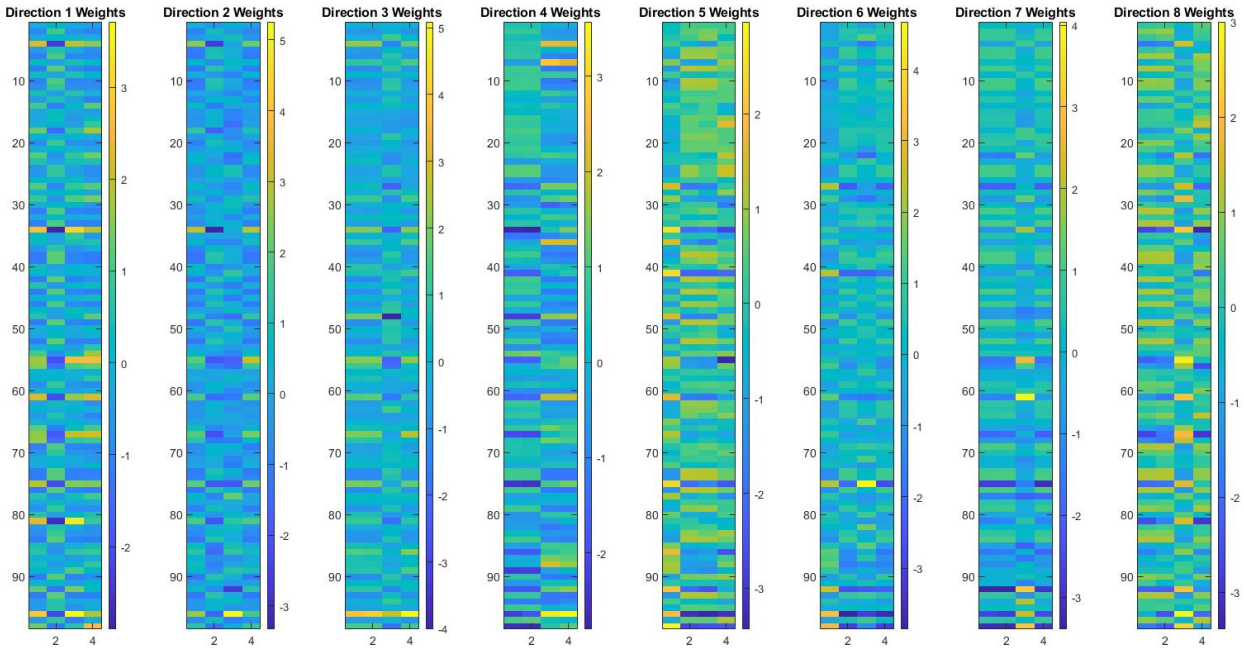
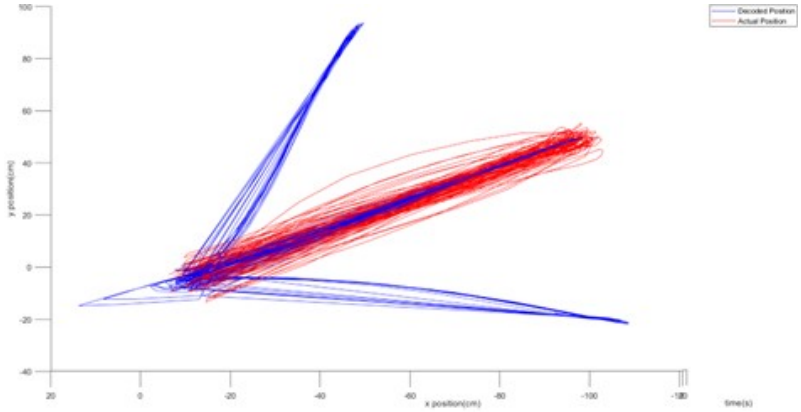
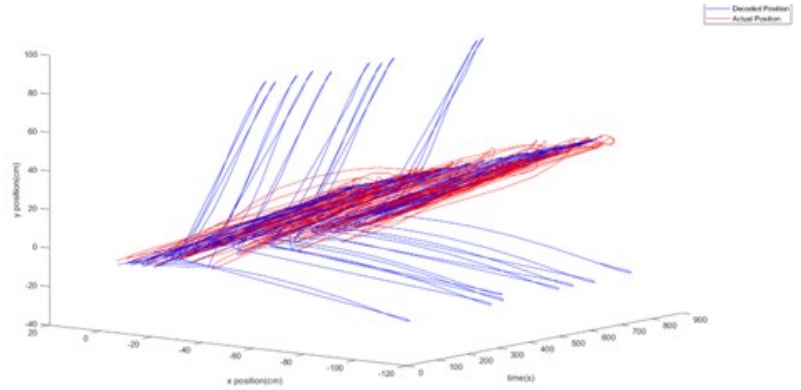


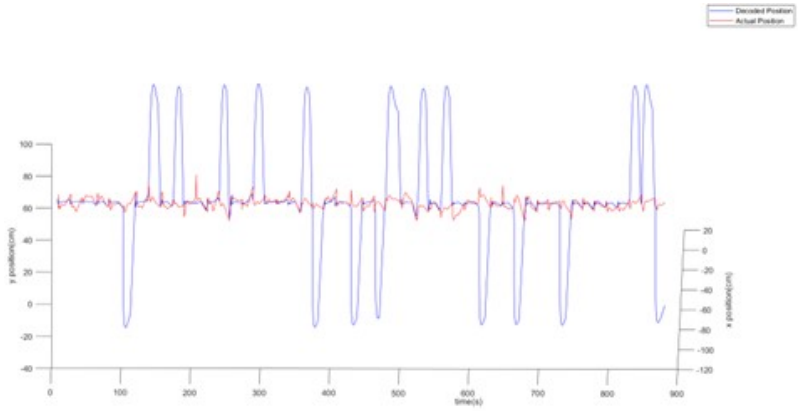
Figure A8: Plot the Kalman Filter transition matrix for 8 directions. As the column is the neuron number, the higher the absolute value of a row of the matrix, the more weight is given to the firing of this specific neuron. This neuron therefore has greater influence in predicting the hand kinematic of the next state.



(a) Trajectory of direction 4.



(b) Trajectory of direction 4 from a second viewpoint.



(c) Trajectory of direction 4 from a third viewpoint.

Figure A9: Figure illustrates the predicted and actual hand movements of direction 4. The actual trajectory can be easily spotted in (a), where the predicted trajectory shows an ideal linear movement while the actual movement is more noisy. (b) gives a comprehensive 3D overview of the whole movement. From (c), the direction classification error can be easily calculated, as the error rate is the number of blue peaks divided by the total trail number of 50. For direction 4, this error rate = 18/50 = 0.36

B. Appendix - Math

A more formal definition of kNN can be encapsulated by the following[4]:

Let $f : \mathbb{R}^2 \rightarrow \{1, \ldots, t\}$ be a target function that maps an input vector to a class label. The kNN algorithm predicts the class label as the most frequently occurring class label among the k nearest neighbors of the query point, which can be expressed as:

$$D_k = \left\{ \left(x^{[1]}, f(x^{[1]}) \right), \ldots, \left(x^{[k]}, f(x^{[k]}) \right) \right\} \tag{5}$$

where $x^{(l)}$ represents the l-th nearest neighbor.

The kNN hypothesis is thus defined by:

$$h(x^{[t]}) = \text{mode}(\{f(x^{[1]}), \ldots, f(x^{[k]})\}) \tag{6}$$

A common metric used for determining the k-nearest neighbors is the Euclidean distance, defined for two points x_j^a and x_j^b in Rm as:

$$d(x^{[a]}, x^{[b]}) = \sqrt{\sum_{j=1}^{m} (x_j^{[a]} - x_j^{[b]})^2}, \tag{7}$$

C. Appendix - Code

All code used in the making of this report can be found in the following *GitHub* repository: *BMI-Report*. The **positionEstimator.m** and **positionEstimatorTraining.m** functions code can be found below.

Training Function: `positionEstimatorTraining.m`

```
%%% Team NAME : bls
%%% Team Members: Josephine Cao, Jiayu Liu, Xinyi Liu, Fangyuan Wang
%%% BMI Spring 2024 (Update 17th March 2024)

function [modelParameters] = positionEstimatorTraining(training_data)
% POSITIONESTIMATORTRAINING Trains the model parameters for estimating hand
    position.
% Input:
%   training_data - Struct containing the neural and hand position data for
    training.
% Output:
%   modelParameters - Struct containing the trained model parameters.

%% Initialize variables for data processing and feature extraction
store_sr = []; % Temporary storage for spike rates
spikerates = []; % Collection of firing rates across all trials and neurons
Ave_sr = []; % Aggregated firing rates for feature vector
store_vx = []; % Temporary storage for velocity in x-direction
store_vy = []; % Temporary storage for velocity in y-direction
disp_x = []; % Displacement in x-direction
disp_y = []; % Displacement in y-direction
vel_x = []; % Velocity in x-direction
vel_y = []; % Velocity in y-direction
pos_x = []; % Position in x-direction
pos_y = []; % Position in y-direction
trainingData = {}; % Processed training data for model training
kinematics = {}; % Kinematic features extracted from training data
centers = {}; % Mean position for normalization purposes
lim_pos = {};
windowSize = 20; % Time bin size for spike rate calculation
current_length = []; % To keep track of data lengths

%% Format data length and pre-process
% Determine the maximum length to standardize the length of neural signals
t_max = 0;
for i = 1:size(training_data, 1)
    for j = 1:size(training_data, 2)
        leng = size(training_data(i,j).spikes(1, :),2);
        current_length = cat(2, current_length, leng);
    end
end
t_max = floor((mean(current_length)-50)/windowSize)*windowSize;

% Pad the data with zeros to make all trials of minimun equal length
for i = 1:size(training_data, 1)
    for j = 1:size(training_data, 2)
        n_neurons = size(training_data(i,j).spikes, 1);
```

```
        for k = 1:n_neurons
            current_length = size(training_data(i,j).spikes(k, :), 2);
            if current_length < t_max
                training_data(i,j).spikes(k, (current_length+1):t_max) = 0;
                training_data(i,j).handPos(:,(current_length+1):t_max) = repmat(
                    training_data(i,j).handPos(:,1), 1, (t_max-current_length));
            end
        end
    end
end

%% Feature Extraction
% Extracts spike rates and kinematic features (velocity and position) for model
    training
numTrials = length(training_data);
for angle = 1:8
    % Loop over each direction
    for neuron = 1:98
        % Loop over each neuron
        for n = 1:numTrials
            % Loop over each trial
            for t = 300:windowSize:t_max-windowSize
                % Calculate spike rates for each neuron within each time bin
                sum_spikes = sum(training_data(n,angle).spikes(neuron,t:t+
                    windowSize));
                store_sr = cat(2, store_sr, sum_spikes/windowSize);

                % Calculate velocities and positions if first neuron (done once
                    per trial)
                if neuron == 1
                    x1 = training_data(n,angle).handPos(1,t);
                    x2 = training_data(n,angle).handPos(1,t+windowSize);
                    y1 = training_data(n,angle).handPos(2,t);
                    y2 = training_data(n,angle).handPos(2,t+windowSize);

                    velocityX = (x2 - x1) / windowSize;
                    velocityY = (y2 - y1) / windowSize;

                    % Store displacement and velocities
                    disp_x = cat(2,disp_x, x1);
                    disp_y = cat(2,disp_y, y1);
                    store_vx = cat(2, store_vx, velocityX);
                    store_vy = cat(2, store_vy, velocityY);
                end
            end

            % Aggregate firing rates for feature extraction
            spikerates = cat(1, spikerates, store_sr);
            if neuron == 1
                vel_x = cat(1, vel_x, store_vx);
                vel_y = cat(1, vel_y, store_vy);
                pos_x = cat(1, pos_x, disp_x);
                pos_y = cat(1, pos_y, disp_y);
```

```
            end

            % Reset temporary variables for next iteration
            store_sr = [];
            store_vx = [];
            store_vy = [];
            disp_y = [];
            disp_x = [];
        end
        % Compute average firing rates across all trials for
        Ave_sr = cat(1, Ave_sr, mean(spikerates));
        % Reset firing_rate
        spikerates = [];

    end

    % Store processed data for each direction
    trainingData{angle} = Ave_sr;
    Ave_sr = [];

    % Calculate and store kinematic data and center positions for each direction
    kinematics{angle} = [mean(pos_x) - mean(pos_x,"all"); mean(pos_y) - mean(pos_y
        ,"all"); mean(vel_x); mean(vel_y)]';
    [~,~,id_x] = find(pos_x);
    [~,~,id_y] = find(pos_y);
    centers{angle} = [mean(id_x,"all"), mean(id_y,"all")];

    % Reset kinematic variables for the next direction
    vel_x = [];
    vel_y = [];
    pos_x = [];
    pos_y = [];

end

%% KNN Classifier Training
    % Calculate total number of spikes for each neuron within the first 320ms of
        each trial
    features = [];
    labels = [];
    spikeCounts = zeros(length(training_data), 98);
    for angle = 1:8
        for neuron = 1:98
            for n = 1:length(training_data)
                sum_spikes = sum(training_data(n,angle).spikes(neuron, 1:320));
                spikeCounts(n, neuron) = sum_spikes;
            end
        end
        features = cat(1, features, spikeCounts); % Compile spike count data
        Lable_angles = repmat(angle, length(training_data), 1); % Label data with
            direction
        labels = cat(1, labels, Lable_angles);
    end
```

```
145
146     K = 10; % Number of nearest neighbors for KNN
147     customKNNModel = customFitKNN(features, labels, K); % Train KNN model
148
149 %% Kalman Filter Training
150     kalfilter = kalman_fit(kinematics,trainingData,centers); % Train Kalman filter
            using kinematic data
151     % Store the trained model in modelParameters
152     modelParameters = struct('knnModel', customKNNModel, 'kalModel', kalfilter, '
            positionAverage', kinematics); % 'linear', beta,
153 end
154
155
156 function [filter] = kalman_fit(train,response,cent)
157     % KALMAN_FIT Trains Kalman filters for each direction based on the given
            training data.
158     % Algorithm based on [W. Wu, etc] paper "Neural Decoding of Cursor Motion
159     % Using a Kalman Filter".
160     % Assuming Kalman parameters A, Q, W, H remain unchanged during time
161     % updates. Therefore, parameters can be trained based on the average
162     % kinematics of the training data.
163
164     % Inputs:
165     %   train - Cell array where each cell contains kinematic data (position,
            velocity) for a specific direction.
166     %   response - Cell array where each cell contains the corresponding neural
            firing data for each direction.
167     %   cent - Cell array containing the mean position for the kinematic data in
            each direction.
168     % Outputs:
169     %   filter - Struct array containing the trained Kalman filter parameters for
            each direction.
170
171     % Train the Kalman filter for each of the 8 directions
172     for dirn = 1:8
173         spike_train = response{dirn}; % Extract spike train data for the current
                direction
174         kim = train{dirn}; % Extract kinematic data for the current direction
175
176         % Compute the variables required for filtering.
177         % Preparing data for least squares solution
178         X2 = kim(2:end, :)'; % Kinematics at time t+1
179         X1 = kim(1:end-1, :)'; % Kinematics at time t
180
181         % Compute the state transition matrix A using least squares
182         % This matrix predicts the next state based on the current state
183         A = X2*X1'*inv((X1*X1'));
184
185         % Compute the process noise covariance matrix W
186         % This matrix represents the uncertainty in the model prediction
187         W = ((X2 - A*X1)*((X2 - A*X1)'));
188
```

```
        % The least-squares-optimal transformation from kinematics to neural
            firing rate
        X = kim'; % Kinematic data
        Z = spike_train; % Neural firing data

        % Compute the observation matrix H using least squares
        % This matrix relates the state to the observed neural firing rates
        H = Z*X'*inv((X*X'));

        % Compute the observation noise covariance matrix Q
        % This matrix represents the uncertainty in the measurements (spike counts
            )
        Q = ((Z - H*X)*((Z - H*X)'));

        % Regularization to avoid singular matrices
        epsilon = 1; % Small regularization factor
        % epsilon = '1' - assume perfect linearity to minimize the random position
            error.

        W = W + epsilon * eye(size(W)); % Regularize process noise covariance
            inspect 'normalize(W,'center','mean')
        Q = Q + epsilon * eye(size(Q)); % Regularize measurement noise covariance
             inspect 'normalize(Q,'center','mean')

        % Store the computed matrices in the filter struct for the current
            direction
        filter{dirn}.Q = Q;
        filter{dirn}.H = H;
        filter{dirn}.A = A;
        filter{dirn}.W = W;
        % Store the mean position for the current direction
        filter{dirn}.center = cent{dirn};
    end
end


% %% Linear Regression
%     % Train linear regression models for velocity prediction using spiking data
%
% beta = struct([]);
%
% for dirn=1:8
%     vel = kinematics{dirn}(:,3:4);
%
%     spiketrain = trainingData{dirn};
%     beta{dirn} = lsqminnorm(spiketrain',vel);
% end
```

Testing Function: `positionEstimator.m`

```
%%% Team NAME : bls
%%% Team Members: Josephine Cao, Jiayu Liu, Xinyi Liu, Fangyuan Wang
%%% BMI Spring 2024 (Update 17th March 2024)

```

```
function [x, y, newModelParameters] = positionEstimator(testData, modelParameters)
% POSITIONESTIMATOR estimates the position (x, y) of a hand movement based on
    neural signals.
% Inputs:
%   test_data - The neural signals data for testing.
%   modelParameters - Struct containing the parameters of the model trained using
    training data.
% Outputs:
%   x, y - Estimated coordinates of hand movement.
%   newModelParameters - Updated model parameters after processing test_data.
    t_fla = 0;
    % Time window for processing signals
    dt = 20;
    % Determine the maximum time point in the given test data
    tmax = length(testData.spikes);
    % Calculate the minimum time point for the current window
    tmin = tmax - dt;
    t_fla = ceil((tmax - 300)/dt);
    if t_fla > 14
        t_fla = 14;
    end
    % Get the number of neurons based on the size of the spikes matrix
    num_neurons = length(testData.spikes(:,1));
    % Determine if the current test data segment is within the initial 320ms
       window
    fla = t_fla == 1;

    %% Estimate reaching angle
    % If the test data is within the initial 320ms, predict movement direction
    % otherwise, use the previously predicted direction.
    if fla
        % Sum spikes across all neurons to get the spike count for each neuron
        sum_spikes = sum(testData.spikes(:,:),2)';
        % Access the trained KNN model from the model parameters
        mdl = modelParameters.knnModel;
        % Predict the movement direction using the custom KNN predictor
        predict_angles = mode(customPredictKNN(mdl, sum_spikes));
        % Update the direction in the model parameters
        modelParameters(1).direction = predict_angles;
    else
        % Use the previously predicted direction if beyond initial 320ms
        predict_angles = modelParameters(1).direction;
    end

    %% Firing Rate Calculation
    % Initialize array for storing firing rates of all neurons
    store_spikerates = zeros(1,num_neurons);
    % Calculate firing rate for each neuron in the specified time window
    for i = 1:num_neurons
        sum_spikes = sum(testData.spikes(i,tmin:tmax));
        store_spikerates(i) = sum_spikes / dt; % Normalize by the window duration
    end

```

```
    %% Kalman Filter Update
    % Update the hand position estimates using the Kalman filter
    [x, y, newModelParameters] = kalmanFilter(testData, store_spikerates,
       modelParameters, predict_angles, fla, t_fla);

end


function [x_prediction, y_prediction, newModelParameters] = kalmanFilter(datainput
   , response, modelpar, direct, flag, tima_fla)
% KALMANFILTER updates the position estimate using a Kalman filter.
% Inputs:
%   datainput - Test data input containing neural signals.
%   response - The response or the firing rate of neurons.
%   modelpar - Current model parameters.
%   direct - Predicted direction of movement.
%   flag - Indicates if the position should be initialized (true) or updated (
   false).
% Outputs:
%   x_prediction, y_prediction - Updated position estimates.
%   newModelParameters - Updated model parameters including the Kalman filter
   state.

    % Access the Kalman filter parameters for the estimated direction
    filter = modelpar(direct).kalModel;

    % Initialize arrays for corrected state and covariance (unused in this context
       )
    xCorrected = [];
    PCorrected = [];

    % Extract the trained matrices from the Kalman filter model
    A = filter.A; % State transition matrix
    W = filter.W; % Process noise covariance matrix
    H = filter.H; % Measurement matrix
    Q = filter.Q; % Measurement noise covariance matrix

    % Initialize or retrieve the state and covariance matrix based on the flag
    if flag
        % For initial position, calculate relative to the center
        x = datainput.startHandPos(1,1); % Initial x position
        y = datainput.startHandPos(2,1); % Initial y position
        x0 = [(x - filter.center(1)), (y - filter.center(2)), 0, 0]'; % Initial
           state vector
        p0 = eye(length(A)); % Initial covariance matrix (identity matrix scaled
           by A's size)
    else
        % Use the previous state and covariance if not initializing
        x0 = filter.x0;
        p0 = filter.P0;
    end

    % Prediction step: predict the next state and covariance
```

```
101     xPred = A * x0; % Predicted state
102     PPred = A * p0 * A' + W; % Predicted covariance
103
104     % Measurement update step: incorporate the new measurement
105     y = response' - H * xPred; % Measurement residual (difference between observed
            and predicted measurement)
106     S = H * PPred * H' + Q; % Residual covariance
107
108     K = PPred * H' / S; % Kalman gain: weights the amount of the measurement
            update
109     xCorrected = xPred + K * y; % Corrected state estimate
110     PCorrected = (eye(size(K * H)) - K * H) * PPred; % Corrected covariance
            estimate
111
112     % Update the model parameters with the new state and covariance for future
            time steps
113     filter.x0 = xCorrected;
114     filter.P0 = PCorrected;
115     modelpar(direct).kalModel = filter;
116     newModelParameters = modelpar; % Pass along any unchanged parameters
117
118     % Scale estimated position according to average position
119     pos_bound = modelpar(direct).positionAverage; % Retrive position bounds
120     scale_x = 0.9;
121     scale_y = 1 - scale_x;
122     lim_pos = [max(pos_bound(:,1)), max(pos_bound(:,2)); min(pos_bound(:,1)), min(
            pos_bound(:,2))];
123     x_corrected = xCorrected(1);
124     y_corrected = xCorrected(2);
125
126     x_corrected = (x_corrected)*scale_x + scale_y*pos_bound(tima_fla,1);
127     y_corrected = (y_corrected)*scale_x + scale_y*pos_bound(tima_fla,2);
128
129     % Apply max position constraints
130     if x_corrected > lim_pos(1,1)
131         x_corrected = lim_pos(1,1);
132     elseif x_corrected < lim_pos(2,1)
133         x_corrected = lim_pos(2,1);
134     end
135
136     if y_corrected > lim_pos(1,2)
137         y_corrected = lim_pos(1,2);
138     elseif y_corrected < lim_pos(2,2)
139         y_corrected = lim_pos(2,2);
140     end
141
142     % Use x_corrected and y_corrected as the final position estimates
143     % Convert corrected state back to original coordinates and output
144     x_prediction = x_corrected + filter.center(1); % Updated x position
145     y_prediction = y_corrected + filter.center(2); % Updated y position
146 end
147
148 % Linear regression method. (not submitted)
```

```
% Use the start position if it is the beginning of the test data
% Otherwise, update the position using previous position + velocity * 20ms

% % Estimate velocity
% reach = modelParameters(direction).linear;
% velocity_x = total_firing_rate*reach(:,1);
% velocity_y = total_firing_rate*reach(:,2);
% reach = [];
% if length(test_data.spikes) <= 320
%     xl = test_data.startHandPos(1);
%     yl = test_data.startHandPos(2);
%     fla = 1;
% else
%     xl = test_data.decodedHandPos(1,length(test_data.decodedHandPos(1,:))) +
    velocity_x*(dt);
%     yl = test_data.decodedHandPos(2,length(test_data.decodedHandPos(2,:))) +
    velocity_y*(dt);
%     fla = 0;
% end
% % Update modelparameters
% newModelParameters.linear = modelParameters.linear;
% newModelParameters.knnModel = modelParameters.knnModel;
% newModelParameters.direction = direction;
%
% % previous positions
% past_pos = test_data.decodedHandPos;
% % Measurement from KNN
% measurement = [x;y];
```