Particle Filtering for Bayesian Inference in Brain Machine Interface

Jenna Luchak
CID: 01429938
jkl17@ic.ac.uk

Edward McLaughlin
CID:01092693
em12509@ic.ac.uk

Laura Palacio Garcia
CID: 01322823
lp2816@ic.ac.uk

Louis Rouillard Odera
CID: 01388687
llr17@ic.ac.uk

Abstract

Brain-Machine Interfaces (BMIs) may be used to decode brain signals to translate the intention of a user into signals for peripheral devices (e.g. prosthetic limbs).

In this report, an algorithm to predict the direction of motion of an arm movement is to be determined solely from neuron spike data provided. Initially, the algorithm is trained on the spike trains from 98 neurons during 800 trial movements (100 trials in 8 separate directions). Subsequently, it will determine the trajectory of a hand movement given only the neuron spike rates.

Due to their real-world applications, BMIs must be accurate, fast (pseudo real-time) and generalised (being applicable in scenarios not explicitly trained on). As a result, the authors intend to create a process with a low RMSE relative to the actual trajectory, a quick runtime and a general decoder for arm movement.

The algorithm developed estimated the hand position with a root-mean-square error (RMSE) of 27.56 cm in 275.76 seconds, averaged over 10 testing trials. Appropriate comparisons are made with similar and contrasting techniques made in literature.

1. Submitted Solution

1.1. Multi-Layer Perceptron (MLP) Classifier

Although the users arm is stationary for the first 300ms of the provided data, the users intention is believed to be encoded. This belief is inspired from [1], where such information can be extracted from the preparatory data. To gain a premonition about the direction of the arm movement, a Multi-Layer Perception (MLP) is implemented on the training data. Having trained this MLP, the first 300ms of testing data may be used to classify the general direction of the arm movement.

1.2. Filtering the neurons

The filtering of neurons was implemented as some neurons may not be responsible for motor movement - such neurons should be removed from the data set. The filtering of neurons was determined from the analysis of the training data across all trials and directions using, where relevant, Peri-Stimulus Time Histograms (PSTHs). A neuron's *baseline* activity is defined as an average of the neuron's activity during the preparatory period. By comparing this baseline activity to the neuron's activity during movement, it is possible to extract information about the neuron's sensitivity. The filtering function selects acceptable neurons from one of three different methods: Fano Factor, Firing rate ratio or no filtering. The Fano Factor, F, was calculated over the total experiment time duration, T, for every i^{th} neural unit. By calculating the average variance of firing rates during movement, σ, and the average baseline firing rate, μ, the Fano Factor was calculated from Equation 1.

$$F_i(T) = \frac{\sigma_i(0,T)}{\mu_i(0,T)} \tag{1}$$

The firing rate ratio, R, was calculated over the total experiment time duration, T, for every i^{th} neural unit. By calculating the average firing rate during movement, μ_M, and baseline, μ_B, the ratio was calculated from Equation 2.

$$R_i(T) = \frac{\mu_{M,i}(0,T)}{\mu_{B,i}(0,T)} \tag{2}$$

Acceptable neurons were found by comparing their F or R value to threshold criteria. The firing rate method removes neurons that fire less often during movement than their baseline, while Fano Factor method removes neurons with a value greater than 3 standard deviations from the average across all neurons. The original number of neural units in the data was 98. The resulting number of neurons qualified from each filtering method can be seen in Table 1. It was found that the firing rate method removed the most neurons from the data set, followed by the Fano factor method.

Fano Factor	Firing rate	None
95	67	98

Table 1: Number of neurons used for future analysis after filtering them based on 3 different methods.

After appropriately filtering the neurons, the data for each trial is compartmentalised into bins and averaged across these bins producing the average firing rate and arm velocity for every orientation of movement. A data structure which contains the neuron baselines, average firing rates and arm speeds is created to be used for the remainder of the process.

1.3. Particle Filter

In order to be able to infer the hand velocity from spike trains, Gaussian inference based on the observed probability of spike trains given a hand velocity is used. This means of inferring motor control from spike trains is inspired from [2], where it was implemented to read arm movements from neuron spike data in premotor cortex of Rhesus monkeys. In that case, the algorithm outperformed population vector and optimal linear estimator in terms of mean-squared error by 10x and 5x respectively.

Training In order to estimate the hand velocity given a set of neuron spike trains, first the probability of a set of spike trains given a hand velocity must be determined. Assuming that neurons fire with a probability relating to a *Poisson distribution*, the likelihood of a spike train given a speed $\vec{V}$ and a neuron's determined characteristics, λ_i, can be calculated from

$$\mathbb{P}(y_i|\vec{V}, \lambda_i) = e^{-\lambda_i dt} \frac{(\lambda_i dt)^{y_i}}{y_i!} \tag{3}$$

where each neuron $i = 1, ..., N$ and $N = 98$ is parametrized by λ_i, according to Equation 4, which is a function of the arm endpoint velocity $\vec{V}$.

$$\lambda_i = \mathrm{b}_i + \mathrm{ds}_i \times \vec{D}_i . \frac{\vec{V}}{\|\vec{V}\|} + \mathrm{ss}_i \times \|\vec{V}\| \tag{4}$$

Each neuron is characterised by four parameters according to the training data:

- b_i represents the neuron's baseline fire rate (average calculated over the preparatory period of 320ms across every trial and direction for a specific neuron)
- ds_i represents the neuron's direction sensitivity (variation of the firing rate due to the alignment of the arm endpoint velocity with the neuron's preferred direction)
- $\vec{D}_i$ represents the neuron's preferred direction (unit vector)
- ss_i represents the neuron's speed sensitivity (variation of the firing rate due to the norm of the arm endpoint velocity)

To identify the aforementioned parameters (see Figure 1), the averaged trial data obtained from the data handling is fit with the relevant functions (cf. code in the appendix).

Decoder Having identified the probability of a spike train y_i over a time-step dt for each neuron i, using Bayes' theory, it is thus possible to infer $\mathbb{P}(\vec{V}|y_i, \lambda_i)$ from Equation 3. Unfortunately, since Poisson distributions have no common conjugate, the integrals implied are too complicated to calculate numerically. A particle filter was chosen as oppose to a Kalman filter (which both utilise Bayesian inference) as it allows for non-Gaussian dynamics, namely, in this case, Poisson dynamics for the neuron firing rates. Thus, a common Monte Carlo technique known as *Particle filtering* is implemented to determine $\vec{V}$. The algorithm for the implemented particle filter is as follows:

1. A population of particles of size $K = 500$ is considered, each with a velocity $\vec{V}_k$, where $k = 1, ..., K$.
2. Given $\vec{V}_k$, each neuron's 20ms spike train (given by the *trial* structure fed to the decoder) has a certain

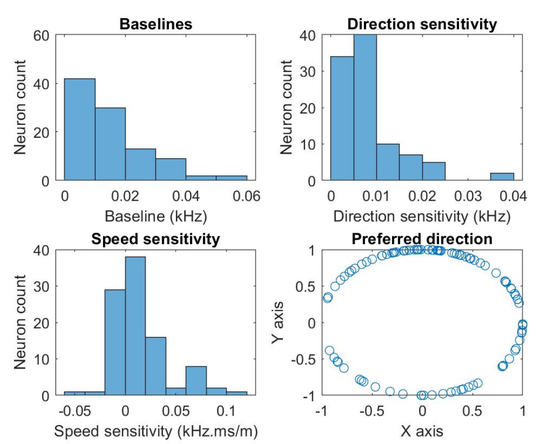

Figure 1: Neuron characteristics after training.

probability of occurring according to equation 3. For the first iteration, the particle velocities are initialised to be randomly Gaussian distributed around the velocity determined by a population vector algorithm ran on the spike data from the first 320ms.

3. The particles are given weights determined by the likelihood that their speed is representative of the true hand velocity according to the probability $\prod_{i=1}^{98} \mathbb{P}(y_i|\vec{V}_k, \lambda_i)$ (considering independent neurons). A Gaussian re-sampling, $\mathcal{N}(V_k, \sigma_1)$, is made about these velocities according to the particle weights. This step is iterated on the spike train data of the given 20ms of spike data until the cloud of particles converges sufficiently to satisfy the stopping criteria. This stopping criteria was given as a certain number of iterations but could equally have been based on the spread of particles about their mean coordinates in velocity space.

4. At the end of the iterations, the "true" speed is taken as the center of gravity of the speed particle population. Finally, the population of particles undergoes a Gaussian re-sampling, $\mathcal{N}(V_{true}, \sigma_2)$ (where $\sigma_2 > \sigma_1$), to initialize the cloud of particle velocities for the next batch of 20ms of spike data. A Gaussian random walk ensures to explore the velocity space without falling into local minima nor ignoring some possibilities.

1.4. Speed Correction

Principle: knowing the information provided by the MLP (see section 1.1), the broad direction of the movement is known from the initial 300ms of data. This is used to correct the decoded 'true' velocity given by the particle filter.

Implementation: considering the decoded position at the current time $\mathbf{X}_{est}$, the decoded velocity returned by the particle filter $\vec{V}_{PF}$, and the desired direction of movement provided by the MLP, Θ, a corrected velocity is determined $\vec{V}_{corr}$. To achieve this, a *pseudo-PD controller* is used such that:

$$\vec{V}_{corr} = \vec{V}_{PF} - K_d \times (\mathbf{X}_{est}.\vec{D}_{ort}^T) \times \vec{D}_{ort} + K_p \times (\mathbf{X}_{end} - \mathbf{X}_{est}) \quad (5)$$

where $\vec{D}_{ort}$ is the vector orthogonal to Θ, and K_d and K_p are gains empirically chosen to minimise the RMSE of the trajectory.

1.5. Implementing a Time Delay

There is a biological delay between the firing neurons and the arm movement which must be accounted for or else the training and testing will be based on an incorrect fitting between firing rates and arm velocities.

For training, the time delay is implemented inside the $handVelocity$ function in the $positionEstimatorTraining$. In order to obtain the hand velocity and position corresponding to the firing rate, a delay is implemented by fitting the spike trains from the current bin to hand velocity data in the subsequent bin. Thus, in the case where the length of each bin is 20ms, hand movement data from the time period from 320 to 520 ms is correlated to the neuron data from 300 to 500 ms. The length of the bin is relates to the real-time delay for a motor command signal to reach the muscles from the neuron location. Finally, this time delay will also be considered in the testing by inversing the logic.

2. Results

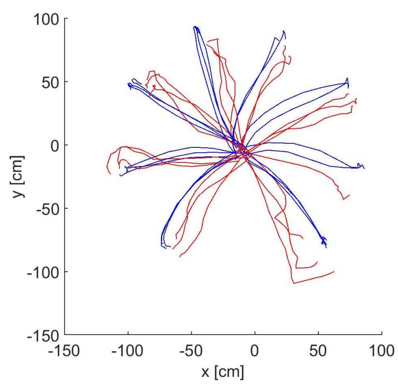

Figure 2: True (in blue) and predicted (in red) hand position for an example of the first 6 blocks.

A root-mean-square error (RMSE) of 27.56 cm was obtained when comparing the true hand position and the estimated hand position. The algorithm took 275.76 seconds to execute after averaging over 10 testing trials. As can be seen in Figure 2, the estimator built approximates the true trajectory.

3. Discussion

3.1. Comparison of Results

The results were overall less accurate as the one seen in the literature for a similar algorithm in [2] where the Integrated Standard Error (ISE, equivalent in its definition with RMSE) was of 0.886cm on a circular arm movement for a Rhesus Monkey. Reasons to explain this difference are multiple: number of neuron units considered (258 in [2] after filtering), quality of train.ing data set (in [2] data were collected on reaching movements as well as on circular arm movement), and a greater calculating power allowing for more complex features (2500 particles for filtering).

3.2. Principle Component Analysis (PCA)

The elementary unit used in the whole project is the neuron (or possibly groups of neurons) which are characterised by individual Poisson parameters, λ_i. However, it is possible that these elementary units could be grouped in a manner to be more informative in explaining the arm movement and with a lower dimensionality. Through the use of PCA, eigenvectors which diagonalize the variance of the data may be used to construct combinations of neurons whose combined rate would be much more informative.

This type of combination, for instance performed by the MLP, could be a very efficient way to improve the decoder while maintaining a generalised solution.

3.3. The Speed Norm Problem

One of the challenges in the task at hand is underlined in the difficulty to reconstruct a non unit speed vector. Equation 4 shows that the rate is a function of 2 variables, the speed direction and norm. This function is not a bijection, meaning, it is impossible to get the full information about the speed without using several neurons. In [3], the intrinsic difficulty to extract information about the speed norm from neural data is highlighted and illustrated in Figure 3.

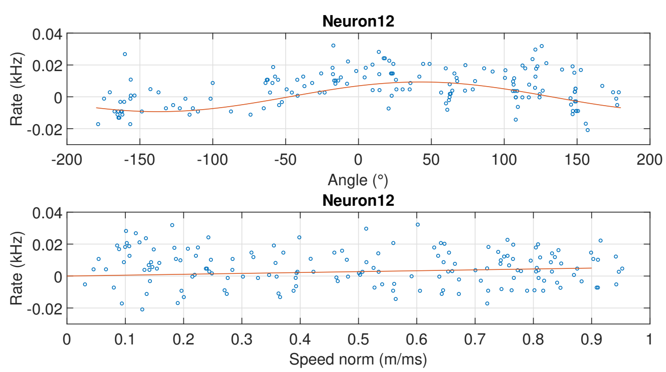

Figure 3: Example fit for 3 neurons: it is visible that the fit on the speed norm data has a low R value and thus is uninformative.

Attempting to decode both speed direction and norm renders the direction determination less precise. Efforts were made during this project to try to uncouple these two dimensions. That is to say more effort should have been placed on accurately determining the speed direction than the speed norm. As a clear mathematical background could not be found to solve this problem, this idea was put aside, but it could improve the resilience of the speed direction decoding when taking the speed norm direction into account.

3.4. Specializing the Model

The competition aspect of this challenge must not obscure the real-world applications BMI are addressing. As a consequence the model built here is meant to be for the most part *generalizable*. Indeed, the prior knowledge of the similarities between training set and testing set, used through the MLP is used only to correct a first estimate of the arm speed through the particle filtering. In an attempt to reduce the RMSE, and as it may be done to create specialized prosthetics, the knowledge of the movement's broad direction could be used further on to obtain 8 specialized sets of parameters, to filter the neurons depending on the situation (CSP), or even to return an artificial mimic of the training set mean trajectory.

It was deliberately chosen not to dwell more than necessary on those aspects, but implementing them would surely drastically reduce the RMSE.

References

[1] Churchland MM, Santhanam G, Shenoy KV. Preparatory Activity in Premotor and Motor Cortex Reflects the Speed of the Upcoming Reach. Journal of Neurophysiology. 2006;96(6):3130–3146. Available from: http://jn.physiology.org/cgi/doi/10.1152/jn.00307.2006.

[2] Brockwell AE. Recursive Bayesian Decoding of Motor Cortical Signals by Particle Filtering. Journal of Neurophysiology. 2004;91(4):1899–1907. Available from: http://jn.physiology.org/cgi/doi/10.1152/jn.00438.2003.

[3] Golub MD, Yu BM, Schwartz AB, Chase SM. Motor cortical control of movement speed with implications for brain-machine interface control. Journal of Neurophysiology. 2014;112(2):411–429. Available from: http://jn.physiology.org/cgi/doi/10.1152/jn.00391.2013.

Contributions Edward: MLP classifier, particle filter, time delay in testing; Jenna and Laura developed the *filtering_neurons* function and time delay; and Louis: bibliography, estimator training, estimator, speed correction, time delay, data display. Everyone: report.

Appendix

PositionEstimatorTraining

```matlab
function [Param] = positionEstimatorTraining(trial_train)
    % We call the filtering function to obtain the data
    trial = filtering_neurons(trial_train , 'None');
    Param = positionEstimatorTrainingCNN(trial_train);

    % Some dimensions for loops
    K = size(trial,2);
    I = size(trial(1).rate,1);
    B = size(trial(1).rate,2);

    speed_angle = zeros(K,B);
    speed_norm = zeros(K,B);

    for k=1:1:K
        for b=1:1:B
            speed_angle(k,b) = atan2(trial(k).speed(2,b),trial(k).speed(1,b));
            speed_norm(k,b) = sqrt(trial(k).speed(2,b)^2+trial(k).speed(1,b)^2);
        end
    end

    % Particle filtering parameters
    N_particles = 100;

    % Returned values initialization
    direction = zeros(I,2);
    speed_sensitivity = zeros(I,1);
    direction_sensitivity = zeros(I,1);
    baseline = trial(1).baseline(:,1);

    % We create the point cloud useful for fitting, to obtain rate as a function of speed
    for i=1:1:I
        Cloud{i}=[0;0;0];
        for k=1:1:K
            Cloud{i}=[Cloud{i},[speed_angle(k,:);speed_norm(k,:);trial(k).rate(i,:)]];
        end
        Cloud{i} = Cloud{i}(:,2:end);
    end

    ft = fittype('exp(a)*cos(x+b)','independent','x','dependent','height');
    options = fitoptions(ft);
    options.StartPoint = [0.1,0];
    for i=1:1:I
        pref_fit{i} = fit(Cloud{i}(1,:)',Cloud{i}(3,:)',ft,options);
        direction(i,:) = [cos(-pref_fit{i}.b),sin(-pref_fit{i}.b)];
        direction_sensitivity(i) = exp(pref_fit{i}.a);
        speed_sensitivity(i,1) = pinv(Cloud{i}(2,:)')*(Cloud{i}(3,:)'-pref_fit{i}(Cloud{i}(1,:)'));
    end

    % Returned parameters
    Param.baseline = baseline;
    Param.direction = direction;
    Param.direction_sensitivity = direction_sensitivity;
    Param.speed_sensitivity = speed_sensitivity;
    Param.particles = zeros(N_particles,2);
    Param.decodedPos = [0,0];
    Param.isfirst = 1;
    Param.N_particles =N_particles;
    Param.bool_neurons = trial(1).bool_neurons(:,1);
    Param.previous_length = 0;

    % Plot
    f2 = figure(2);
    f2.Name = 'Neurons characteristics';
    subplot(2,2,1)
    histogram(Param.baseline)
    ylabel('Neuron count')
    xlabel('Baseline (kHz)')
    title('Baselines')
    subplot(2,2,2)
    histogram(Param.direction_sensitivity)
    ylabel('Neuron count')
    xlabel('Direction sensitivity (kHz)')
```

```
    title('Direction sensitivity')
    subplot(2,2,3)
    histogram(Param.speed_sensitivity)
    ylabel('Neuron count')
    xlabel('Speed sensitivity (kHz.ms/m)')
    title('Speed sensitivity')
    subplot(2,2,4)
    plot(Param.direction(:,1),Param.direction(:,2),'o')
    xlabel('X axis')
    ylabel('Y axis')
    title('Preferred direction')

    neurons_id = 11:13;
    f4 = figure(4);
    f4.Name = 'Neuron fitting';
    angle = -180:1:180;
    speed = 0:0.01:0.9;
    for i=neurons_id
        subplot(length(neurons_id),2,2*(i-neurons_id(1))+1)
        plot(Cloud{i}(1,:)*180/pi,Cloud{i}(3,:),'o',angle,pref_fit{i}(angle*pi/180))
        xlabel('Angle (    )')
        ylabel('Rate (kHz)')
        title(strcat('Neuron ',num2str(i)))
        subplot(length(neurons_id),2,2*(i-neurons_id(1))+2)
        plot(Cloud{i}(2,:),Cloud{i}(3,:),'o',speed,speed_sensitivity(i,1)*speed)
        xlabel('Speed norm (m/ms)')
        ylabel('Rate (kHz)')
        title(strcat('Neuron ',num2str(i)))
    end

end

function [Param] = positionEstimatorTrainingCNN(trial)
    rates = zeros(8,800);
    output = zeros(8,800);
    Param.meanTraj = {};
    J = size(trial,1);
    K = size(trial,2);
    I = size(trial(1,1).spikes,1);

    for k = 1:K
        Param.meanTraj{k} = zeros(2,550);
        for j = 1:J
            for i = 1:I
                rates(i,(k-1)*100+j) = sum(trial(j,k).spikes(i,1:320),2)/320;
                output(k,(k-1)*100+j) = 1;
            end
            deltaMeanTraj = trial(j,k).handPos(1:2,1:550)/50;
            Param.meanTraj{k} = Param.meanTraj{k} + deltaMeanTraj;
        end
    end

    net = feedforwardnet([10 5 10 5 10 5 10]);
    net = configure(net, rates, output);
    net = init(net);
   [Param.NET, ~] = train(net, rates, output);
end


function filtered_trial = filtering_neurons(trial, type)
% It filters the neurons that we want to use for our analysis.
% We decide if we take into account the neuron or not based on if their firing
% rate is higher compared to the baseline firing rate (before movement).
% Input:
%       trial: A structure that contains the data (100 trials across 8 angles
%       type: A string that can be the type of filtering we want to do (FF, Partial)
% Output:
%       filtered_trial: A structure that contains baseline, rate and speed (divided into bins) for each orientation.
    bins = 20; % number of divisions we want
```

```matlab
    neural_data = getNeuronData(trial, bins);
    baseline = neural_data{1}.baseline;

    num_angles = size(trial,2); % number angles
    num_trials = size(trial,1); % number trials

    for i = 1:size(baseline,1) % neuron
        rate_movement_orients = [];
        for k = 1:num_angles % orientation
            % Average firing rate for the movement phase for this neuron across all trials
            rate_movement_orients(k,:) = neural_data{k}.PSTH(i,:);
        end

        % Variance of firing rate for this neuron across all orientations
        var_neuron_orient(i,:) = var(rate_movement_orients);

        % Average variance across all orientations and across all trials for this neural unit
        var_movement_neuron(i,1) = mean(var_neuron_orient(i,:));

        % Average firing rate across all directions and across all trials for this neural unit
        avg_baseline_neuron(i,1) = mean(baseline(i,:)); % 0 - 300ms
        avg_movement_neuron(i,1) = mean(mean(rate_movement_orients)); % 300 - 500ms

        % Different measures
        FF(i,1) = var_movement_neuron(i)/avg_baseline_neuron(i);
        firing_rate_ratio(i,1) = avg_movement_neuron(i)/avg_baseline_neuron(i);
    end

    % Decide if we want to get rid of the neuron
    switch type
        case 'FF'
            outliers_idx = isoutlier(FF);
            for neuron = 1:length(outliers_idx)
                if outliers_idx(neuron) == 1 && FF(neuron) < mean(FF)
                    bool_neurons(neuron,1) = 0; % we remove
                else
                    bool_neurons(neuron,1) = 1; % we keep
                end
            end
        case 'firing_rate'
            for neuron = 1:size(baseline,1)
                if firing_rate_ratio(neuron) < 0 % we get rid of neuron
                    bool_neurons(neuron,1) = 0; % we remove
                else
                    bool_neurons(neuron,1) = 1; % we keep
                end
            end
        case 'None'
            for neuron = 1:size(baseline,1)
                bool_neurons(neuron,1) = 1;
            end
    end
    bool_neurons = logical(bool_neurons);

    % Remove rate and baseline of this neuron for all
    % orientations and all trials
    for k = 1:num_angles
        neural_data{k}.PSTH(~bool_neurons,:) = [];
    end
```

```
    avg_baseline_neuron(~bool_neurons,:) = [];

    % Average PSTH across all trials for each direction for each neuron
    for k = 1:num_angles
        rate{k} = neural_data{k}.PSTH;
        avg_velocity{k} = neural_data{k}.handVel;
    end

    [rate_split, velocity_split] = data_stepping(trial, bins, rate, avg_velocity);

    % Output
    for k = 1:size(baseline,2)
        filtered_trial(k).baseline = avg_baseline_neuron;
        filtered_trial(k).rate = rate_split{k};
        filtered_trial(k).speed = velocity_split{k};
        filtered_trial(k).bool_neurons = bool_neurons;
    end
end

function [rate_split, velocity_split] = data_stepping(trial, bins, rate, avg_velocity)
% Splits the average firing rate and velocity (x and y) into the average
% of each bin for every orientation.
% Input:
%       trial: A structure that contains the data (100 trials across 8 angles)
%       bins: Number of divisions of data.
%       rate: A struct with the PSTH for all neural units across time for each orientation.
%       avg_velocity: A struct with the velocity (x and y) across time for each orientation.
% Output:
%       rate_split: A struct with the PSTH for all neural units across bins for each orientation
%       avg_velocity: A struct with the velocity (x and y) across bins for each orientation.
    num_angles = size(trial,2); % select number of angles to consider
    num_trials = size(trial,1);

    % Rate
    for k = 1:num_angles
        T = size(rate{k},2); % time
        bin_count = floor(T/bins);
        for num_bin = 1:bins
            rate_split{k}(:,num_bin) = ...
                sum(rate{k}(:,(num_bin-1)*bin_count+1:num_bin*bin_count),2)/bin_count;
        end
    end

    % Velocity
    for k = 1:num_angles
        T = size(avg_velocity{k},2); % time
        bin_count = floor(T/bins);
        for num_bin = 1:bins
            velocity_split{k}(:,num_bin) = ...
                sum(avg_velocity{k}(:,(num_bin-1)*bin_count+1:num_bin*bin_count),2)/bin_count;
        end
    end
end

function neural_data = getNeuronData(trial,bins)
% Calculates hand velocity, hand position, spikes without baseline,
% PSTH and baseline for every trial in each direction.
% Input:
%       trial: A structure that contains the data (100 trials across 8 angles)
%       bins: Number of divisions of data.
% Output:
%       neural_data: A structure that contains the PSTH, hand position
%                           and hand velocity for each trial for each direction.
```

```matlab
    numangles = 8; % select number of angles to consider

    %% Baseline
    % Parameters for baseline function
    params_baseline.n_trials = size(trial,1); % number trials
    params_baseline.n_units = size(trial(1,1).spikes,1); % number units
    params_baseline.t_start = 1; % start time
    params_baseline.t_end = 300; % end time
    params_baseline.direction = 1:numangles; % directions of movement

    % Calculate the baseline of each neuron
    baseline_spikedens = baseLine(trial, params_baseline);

    %% PSTH
    % Parameters for PSTH function
    params_PSTH.n_trials = 50; % average over this number of trials
    params_PSTH.n_units = 98; % number units
    params_PSTH.t_start = 300; % start time
    params_PSTH.t_end = 500; % end time
    params_PSTH.direction = 1:numangles; % directions of movement

    % Calculate the PSTH according to parameters chosen above
    spikedens = PSTH(trial, params_PSTH);

    %% Hand velocity
    neural_data = handVelocity(trial, spikedens, baseline_spikedens, numangles, params_baseline, params_PSTH, bins);
end

function neural_data = handVelocity(trial, spikedens, baseline_spikedens, numangles, params, params_PSTH, bins)
% This calculates the hand velocity and hand position averaged across all
% trials for each orientation.
% Input:
%       trial: A structure that contains the data (100 trials across 8 angles)
%       spikedens: For each neural unit, we return the average spike rate
%                      at each ms of movement for each direction.
%       baseline_spikedens: Average spike rate of the baseline for each
%                                  neural unit (rows) for each direction (columns).
%       numangles: A number that specifies the number of directions.
%       params: A structure containing the baseline parameters: number of trials,
%               number of units, start and end time, and the direction.
%       bins: Number of divisions of data.
% Output:
%       neural_data: A structure that contains the PSTH, baseline, hand position
%                        and hand velocity across all trials for each direction.

    % Find the longest handPos for each orientation
    time_movement = params_PSTH.t_start:1:params_PSTH.t_end;

    % Implement delay by taking the next bin_count ms of velocity
    T = time_movement(end)-time_movement(1); % time
    bin_count = floor(T/bins);
    time_begin = time_movement(1) + bin_count;
    time_end = time_movement(end) + bin_count;
    time_movement = time_begin:time_end;

    for k = 1:numangles
        max_length_pos(k) = -inf; % initialize
```

```
        for n = 1:params.n_trials
            if length(trial(n,k).handPos(1,time_movement)) > max_length_pos(k)
                max_length_pos(k) = length(trial(n,k).handPos(1,time_movement));
            end
        end
    end

    % Average handPos and handVel across all trials
    for k = 1:numangles
        % Position
        handPos_x = NaN(params.n_trials,max_length_pos(k));
        handPos_y = NaN(params.n_trials,max_length_pos(k));
        handPos_z = NaN(params.n_trials,max_length_pos(k));
        % Velocity
        handVel_x = NaN(params.n_trials,max_length_pos(k)-1);
        handVel_y = NaN(params.n_trials,max_length_pos(k)-1);
        handVel_z = NaN(params.n_trials,max_length_pos(k)-1);
        for n = 1:params.n_trials
            current_length = length(trial(n,k).handPos(1,time_movement));
            handVel = diff(trial(n,k).handPos(:,time_movement),1,2); % obtain hand velocity

            handPos_x(n,1:current_length) = trial(n,k).handPos(1,time_movement);
            handPos_y(n,1:current_length) = trial(n,k).handPos(2,time_movement);
            handPos_z(n,1:current_length) = trial(n,k).handPos(3,time_movement);

            handVel_x(n,1:current_length-1) = handVel(1,:);
            handVel_y(n,1:current_length-1) = handVel(2,:);
            handVel_z(n,1:current_length-1) = handVel(3,:);
        end
        neural_data{k}.handPos = [nanmean(handPos_x); nanmean(handPos_y); nanmean(handPos_z)];
        neural_data{k}.handVel = [nanmean(handVel_x); nanmean(handVel_y); nanmean(handVel_z)];
    end

    for k = 1:numangles
        for i = 1:params.n_units
            neural_data{k}.PSTH(i,:) = spikedens{i}(k,:) - baseline_spikedens(i,k);
        end
        neural_data{k}.baseline = baseline_spikedens;
    end
end

function spikedens = PSTH(trial, params)
% This calculates the Peristimulus time histogram. This is the histograms
% of the times at which neurons fire (which is from 300ms to 500ms: time of movement).
% Input:
%       trial: A structure that contains the data (100 trials across 8 angles)
%       params: A structure containing the number of trials, number of units,
%               start and end time, and the direction.
% Output:
%       spikedens: For each neural unit, we return the average spike rate
%                  at each ms of movement for each direction.
    for k = params.direction
        for i = 1:params.n_units
            for n = 1:params.n_trials
```

```
                spikecount{i}(n,:) = trial(n,params.direction(k)).spikes(i,params.t_start:params.t_end-1);
            end
            spikedens{i}(k,:) = sum(spikecount{i},1)/params.n_trials;
        end
        clear spikecount
    end

end

function baseline_spikedens = baseLine(trial, params)
% This function calculates the baseline (0-300ms of the monkey's non-movement).
% Input:
%       trial: A structure that contains the data (100 trials across 8 angles)
%       params: A structure containing the number of trials, number of units,
%               start and end time, and the direction.
% Output:
%       baseline_spikedens: Average spike rate of the baseline for each
%                           neural unit (rows) for each direction (columns).
    for k = params.direction
        for i = 1:params.n_units
            for n = 1:params.n_trials
                spikecount{i,k}(n,:) = trial(n,params.direction(k)).spikes(i,params.t_start:params.t_end-1);
            end
            spikedens{i,k}(1,:) = sum(spikecount{i,k},1)/params.n_trials;
            baseline_spikedens(i,k) = mean(spikedens{i,k});
        end
        clear spikecount
        clear spikedens
    end
end
```

PositionEstimator

```
function [decodedPosX, decodedPosY, newParameters] = positionEstimator(trial, Param)
    %Parameters
    N_iterations = 5;
    speed_std = 0.02 ;
    speed_std2 = 0.3 ;
    t_bin = 20;
    t_planning = 320;

    if size(trial.spikes,2)<Param.previous_length
        Param.isfirst = 1;
        Param.decodedPos = trial.startHandPos;
    end

    %Neuron filtering
    N = size(trial,1);
    K = size(trial,2);
    for n=1:1:N
        for k=1:1:K
            trial(n,k).spikes = trial(n,k).spikes(Param.bool_neurons,:);
        end
    end

    if Param.isfirst
        % For first estimate we use population vector as the expected value
        % for a Gaussian repartition of particles

        % We obtain the rates and normalized directions, and make a weighted
        % sum of the latter
        rates = sum(trial.spikes(:,:),2)/t_planning-Param.baseline;
        directions_norm = sqrt(Param.direction(:,1).^2+Param.direction(:,2).^2);
        planned_speed = rates'*(Param.direction(:,:)./directions_norm);

```

```
        % The first step is planification , there is no actual movement
        decodedPosX = trial.startHandPos(1,1);
        decodedPosY = trial.startHandPos(2,1);

        newParameters = Param;
        newParameters.Speed_estimate_prev = planned_speed;
        newParameters.particles = planned_speed + randn(Param.N_particles,2)*speed_std2;
        newParameters.decodedPos = [decodedPosX,decodedPosY];
        % We move on to next steps with movement
        newParameters.isfirst = 0;
        newParameters.previous_length = size(trial.spikes,2);
        % Calculate the prefered direction
        angles = [30, 70, 110, 150, 190, 230, 310, 350]/180*pi;
        directions = Param.NET(sum(trial.spikes,2)/size(trial.spikes,2));
        [~, idx] = max(directions);
        newParameters.prefdir = angles(idx);
        newParameters.idx = idx;
    else
        n_mini = 5;
        t_minibin = t_bin/n_mini;

        for mini=1:1:n_mini
            % We create a dummy Param structure for the iterations
            Param_iter = Param;

            % We increment the estimated position
            decodedPosX = Param_iter.decodedPos(1,1) + Param_iter.Speed_estimate_prev(1,1)*t_minibin;
            decodedPosY = Param_iter.decodedPos(1,2) + Param_iter.Speed_estimate_prev(1,2)*t_minibin;

            for iterations=1:1:N_iterations
                % We compute counts (observation)
                counts = sum(trial.spikes(:,end-(n_mini-mini+1)*t_minibin:end-(n_mini-mini)*t_minibin),2);
                % We calculate poisson parameter lambda for each neuron
                Particles_norm = sqrt(sum(Param_iter.particles.^2,2));
                lambda = max(0.0001, Param_iter.baseline(:,1)+Param_iter.direction_sensitivity.*Param_iter.direction*(Param_iter.particles./Particles_norm)'+Param_iter.speed_sensitivity(:,1)*Particles_norm');
                % Weights calculation (P(observation|state) for each particle)
                weights = zeros(1,Param_iter.N_particles);
                for p=1:1:Param_iter.N_particles
                    weights(1,p) = prod(exp(-lambda(:,p)*t_minibin).*(lambda(:,p)*t_minibin).^counts(:,1)./factorial(counts(:,1)));
                end
                % We resample particles according to the weights: "survival of
                % the fittest"
```

```
                weights = weights/sum(weights);
                PartIdx = randsample(1:length(Param_iter.particles), Param_iter.N_particles, true, weights);
                Particles = Param_iter.particles(PartIdx,:);

                %This plot helps to show whats happening.
%               f3 = figure(3);
%               f3.Name = 'Speed particles population';
%               if iterations == N_iterations
%                   plot(Param_iter.particles(:,1),Param_iter.particles(:,2), 'ro')
%               end
%               axis([-1 1 -1 1])
%               pause(0.1)
%
                % We add system noise
                Param_iter.particles = randn(Param_iter.N_particles,2)*speed_std + Particles;
            end
            % After all the iterations, the particle cloud has converged towards
            % the "true" state (i.e. true speed)
            Speed_estimate_prev = mean(Particles);

            % We store parameters for new iteration while adding a -slightly
            % bigger- system noise
            newParameters = Param_iter;
            newParameters.Speed_estimate_prev = correctingSpeed2(Param_iter, Speed_estimate_prev, trial, n_mini, t_minibin, mini);
            newParameters.particles = randn(Param.N_particles,2)*speed_std2 + Particles;
            newParameters.decodedPos = [decodedPosX,decodedPosY];
            newParameters.previous_length = size(trial.spikes,2);
        end
    end

end

function newSpeed = correctingSpeed2(Param, v, trial, n_mini, t_minibin, mini)
    longi = [cos(Param.prefdir),sin(Param.prefdir)];
    ortho = [-sin(Param.prefdir),cos(Param.prefdir)];
    x = Param.decodedPos;

    k = 0.035;
    k2 = 0.01;

%     Magic = [100.118067194997;96.3527537142964;96.3698024072
%     magic = Magic(Param.idx,1);

%       error = abs((x-Param.meanTraj{Param.idx}(1:2,length(trial.spikes)-(n_mini-mini)*t_minibin))*ortho');
%     attractor = magic*longi+trial.startHandPos;
     Magic = [100.118067194997;96.3527537142964;96.3698
     magic = Magic(Param.idx,1);
     attractor = magic*longi+trial.startHandPos;

    if length(trial.spikes) > length(Param.meanTraj{Param.idx})
      error = x*ortho';
    else
%         attractor = Param.meanTraj{Param.idx}(1:2,length(trial.spikes)-(n_mini-mini)*t_minibin);
```

```
        sn = sign((x-(Param.meanTraj{Param.idx}(1:2,length(trial.spikes)-(n_mini-mini)*t_minibin)-Param.meanTraj{Param.idx}(1:2,1)+trial.startHandPos))*ortho');
        error = sn.*(x-(Param.meanTraj{Param.idx}(1:2,length(trial.spikes)-(n_mini-mini)*t_minibin)-Param.meanTraj{Param.idx}(1:2,1)+trial.startHandPos))*ortho';
    end

    correction = -k2*error*ortho+k*(attractor-x);%/norm(attractor-x);

    newSpeed = v+correction;

end
```