ECE 189, Fall 2024 **Project**
UCLA Prof. J.C. Kao
Due: Dec 5, 2024, e-mailed to `kao@seas.ucla.edu`

In this project, we'll first walk you through getting a brain-machine interface decoder built in `MATLAB` using the optimal linear estimator algorithm we derived in class. The dataset we'll be working with comes from monkey experiments performed in the laboratory of Prof. Krishna V. Shenoy, and are shared for instructional purposes only. Do not distribute this dataset; any other use of this dataset outside of this project requires permission from Profs. Kao & Shenoy.

The dataset can be downloaded from Bruin Learn:

`JR_2015-12-04_truncated2.mat`

We'll walk you through various modules here before we get to the project prompt. All the code written here is uploaded to Bruin Learn.

The `R`-struct: trial information

The `R` struct is where all the data is kept. It is an array of structures. Let's go ahead and load it.

```
>> load JR_2015-12-04_truncated2.mat
```

This will drop a variable called `R` into your workspace, and if we look at what `R` is, we'll see the following.

```
>> R

R =

  1 x 506 struct array with fields:

    startDateNum
    startDateStr
    timeTargetOn
    timeTargetAcquire
    timeTargetHeld
    timeTrialEnd
    subject
    counter
    state
    cursorPos
    spikeRaster
    spikeRaster2
    isSuccessful
    trialNum
    timeFirstTargetAcquire
    timeLastTargetAcquire
    trialLength
    target
```

As you can see, this is an array of structs. Here, we happen to have 506 of these structs, and each struct has the fields that you see from the `MATLAB` output. In case you are unfamiliar with structs,

let's play around. First, we should extract out a single struct from the array of structs. Let's get the first element.

```
>> R(1)

ans =

  struct with fields:

              startDateNum: 7.3630e+05
              startDateStr: '2015-12-04 13:39:23'
              timeTargetOn: 21
         timeTargetAcquire: 362
            timeTargetHeld: 862
              timeTrialEnd: 882
                   subject: 'JenkinsC'
                   counter: [1 901 double]
                     state: [1 901 double]
                 cursorPos: [3 901 double]
               spikeRaster: [96 901 double]
              spikeRaster2: [96 901 double]
              isSuccessful: 1
                  trialNum: 420
    timeFirstTargetAcquire: 362
     timeLastTargetAcquire: 362
               trialLength: 862
                    target: [3 1 double]
```

This is the first struct in the R-struct (when we say R-struct, this refers to the array of structs). Since we're referencing just one struct, MATLAB will go ahead and print out the values that it can for each field. You can see elements like **startDateStr**, which as you might guess, is the exact time the data in this particular struct was collected.

It turns out that each struct element in the R-struct, i.e., this array of structs, is a *trial*. Therefore, R(i) extracts the information on the ith trial. A trial corresponds to the monkey being shown a target, and subsequently the monkey reaching to that target. (Note that in this dataset, there are no preparatory periods for the monkey to prepare the reach; here, the monkey just continues to reach from target to target.) That is, in the R-struct we provided to you, the monkey makes 506 reaches, all contiguously in time (i.e., he does not pause at any point). The struct R(i) corresponds to all the data collected during the ith trial.

So let's take a look at what happened on trial 1. The monkey had to reach to a target, and this information is in the field **target**. To reference fields in a struct, we use a . in MATLAB. So, to get the first trial's target location, we evaluate R(i).target, i.e.,

```
>> R(1).target

ans =

     0
     0
   -70
```

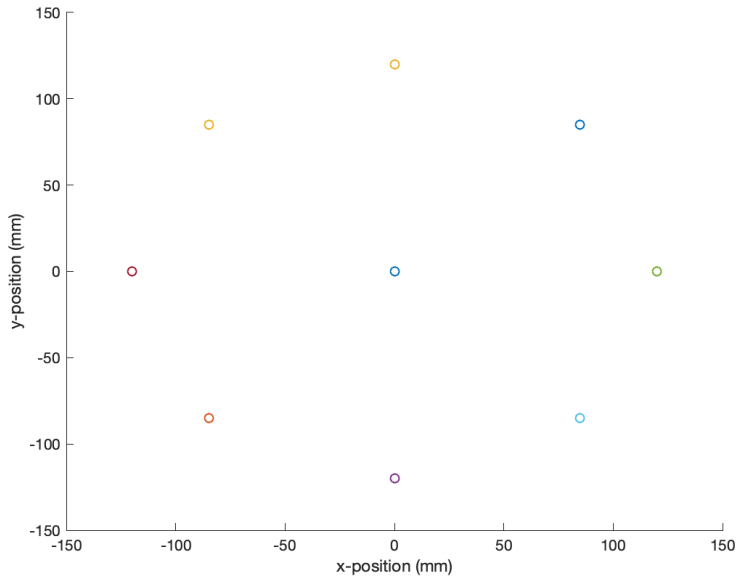
This is a vector with 3 elements; they correspond to the x, y, z position of the target. The units of all targets and movements in the R-struct are in mm. In this experiment, we ignored the z-position, and so all z-positions will be -70. We can safely ignore this information. This tells us that on trial 1, the monkey had to reach to a target at (x, y) position $(0, 0)$. This is the center of the screen.

Let's look at trial 2 now. What target did the monkey reach to on this trial?

```
>> R(2).target

ans =

     0
  -120
   -70
```

And so the monkey reached to a downward target that is 120 mm (or equivalently, 12 cm) below the center target, since its location is $(0, -120)$. It turns out that for the data we're giving you, the monkey was doing a "center-out-and-back" task, where the monkey acquires a center target, then reaches out to a radial target (on a circle with radius 12 cm), then back to the center target, then out to another radial target, etc. Let's plot all the target locations.

```
figure;
hold all;
xlabel('x-position (mm)');
ylabel('y-position (mm)');

for i = 1:numel(R)
    plot(R(i).target(1), R(i).target(2), 'marker', 'o');
end
```



This confirms the task; there are only radial targets and a center target, and across all 506 trials, the monkey repeatedly reaches between these targets. Further, there are only 8 unique radial tar-

gets.

You may have also noticed a field `R(i).isSuccessful`. This is a `logical` or `boolean` variable that tells us if the trial was successful. We would only want to train on trials that were successful, since if the monkey wasn't successful, he may not have been paying attention. Let's calculate how many successful trials there were.

```
>> sum([R.isSuccessful])

ans =

   506
```

In this syntax, if you type `[R.field]` (noting the enclosed brackets), it will concatenate all of the values of that field across the array, i.e., `[R.isSuccessful]` is equivalent to

```
success_array = []

for i = 1:numel(R)
    success_array = [success_array R(i).isSuccessful];
end
```

but of course, much quicker to type. All this is to say that of our 506 trials, all of them were successful, so we can use all of them to build our brain-machine interface decode algorithm.

The R-struct: ms-level information

We can't do anything if all we know are the target locations and whether a trial is successful. As we discussed in class, we need to know the kinematics at every single point in time and the neural data at every single point in time. It turns out the R-struct also has this information.

Cursor movements

`R(i).cursorPos` is a $3 \times T_i$ matrix that contains the monkey's hand position over time on trial i, where T_i is the length of trial `i` in milliseconds. The 1st row is the x-position and the 2nd row is the y-position of Monkey J. Ignore the 3rd row – which is z-position (set to -70 mm by default in this data). The units are in millimeters. Each column represents 1 ms. That is, if $T_i = 901$, this indicates that the trial lasted for 901 milliseconds.

If we look at `R(1)`, we see that `R(1).cursorPos` is a 3×901 matrix, indicating that this trial lasted for 901 ms. If we look at some of the values, we'll see the following:

```
>> R(1).cursorPos(1:2, 1:30)

ans =

  Columns 1 through 7

   81.9700   81.9700   81.9700   81.9700   81.9700   81.9700   81.9700
   89.7900   89.7900   89.7900   89.7900   89.7900   89.7900   89.7900

  Columns 8 through 14
```

```
    81.9700   81.9700   81.9700   81.9700   81.5200   81.5200   81.5200
    89.7900   89.7900   89.7900   89.7900   89.5700   89.5700   89.5700

  Columns 15 through 21

    81.5200   81.5200   81.5200   81.5200   81.5200   81.5200   81.5200
    89.5700   89.5700   89.5700   89.5700   89.5700   89.5700   89.5700

  Columns 22 through 28

    81.5200   81.5200   81.5200   81.5200   81.5200   81.5200   80.9100
    89.5700   89.5700   89.5700   89.5700   89.5700   89.5700   89.2100

  Columns 29 through 30

    80.9100   80.9100
    89.2100   89.2100
```

One thing to note here is that the same value (e.g., (81.52, 89.57)) are held for about 16 ms. This is because our system that tracks the monkey's finger only samples at 60 Hz, and so we only get a new sample every 60 Hz. If we want to build a BMI decoder that e.g., decodes every 5 ms, we would have to interpolate over these values. However in this project, we'll build a BMI decoder that decodes every 25 ms, and so we can safely ignore this interpolation.

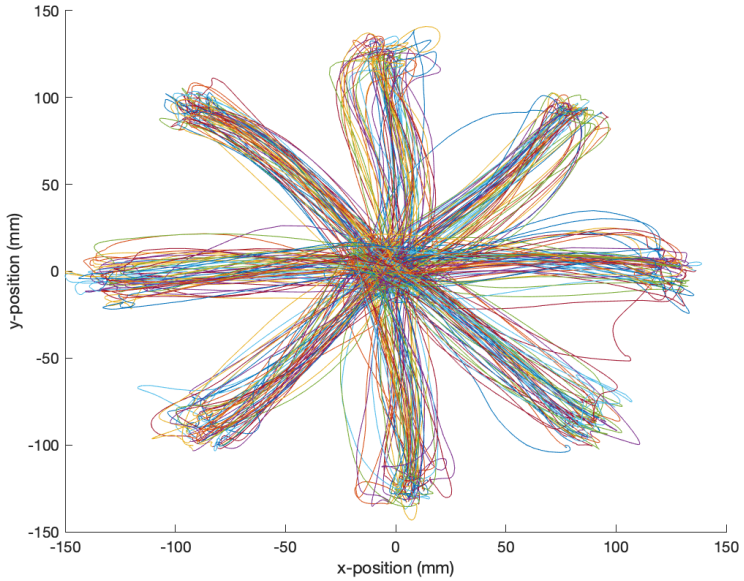
Note that `R(i).cursorPos(:, end)` and `R(i+1).cursorPos(:, 1)` are also separated by 1 ms, so that the `R`-struct contains millisecond resolution data and no segments of time are unobserved. (The monkey never pauses.) Therefore, we can go ahead and plot the monkey's reaching movements via the following code:

```
figure;
hold all;
xlabel('x-position (mm)');
ylabel('y-position (mm)');

for i = 1:numel(R)
    plot(R(i).cursorPos(1,:), R(i).cursorPos(2,:));
end

xlim([-150, 150]);
ylim([-150, 150]);
```

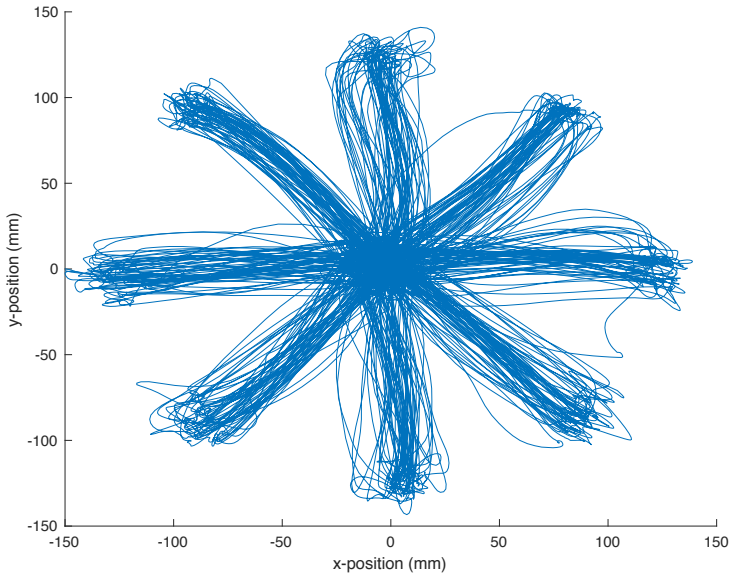
(Figure on next page.) With this, we see hand trajectories that make sense. They go from target to target. The monkey also makes pretty straight reaches, though e.g., if you look to reaches towards the upwards target at $(0, 120)$, some trajectories he goes slightly left and curves right at the end, where other trials he goes slightly right. We trained our monkeys to have very precise and consistent behavior. This is an important for doing computational neuroscience analyses.

To return to our earlier notation, another way to plot all the trials is to concatenate all the `cursorPos`'s together, i.e.,

```

figure;
```




```
3  xlabel('x-position (mm)');
4  ylabel('y-position (mm)');
5
6  X = [R.cursorPos];
7
8  plot(X(1,:), X(2,:));
9
10 xlim([-150, 150]);
11 ylim([-150, 150]);
```



(Note, it's all blue as opposed to multi-colored, since `MATLAB`'s `hold all` command, used in the

prior plot, plots each different call of the `plot` command with a different color.)

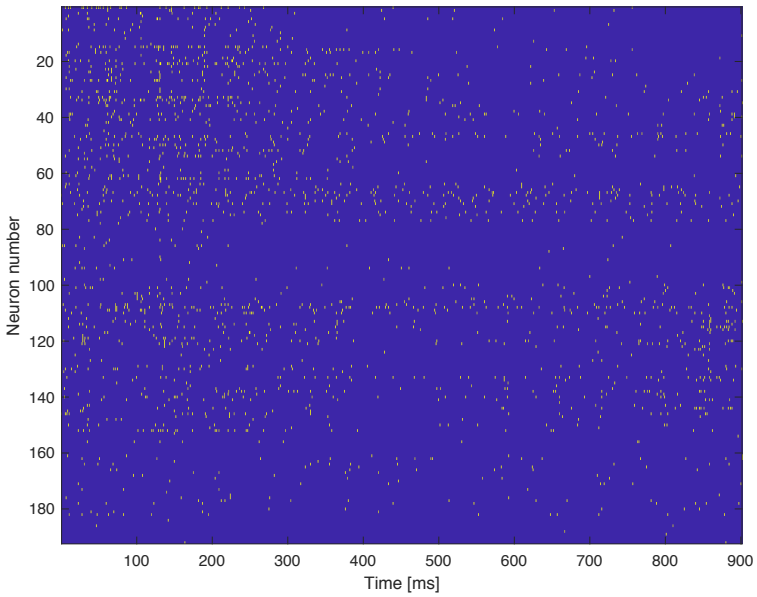
Neural data

The `R`-struct contains neural data recorded from electrode arrays implanted in Monkey J's motor cortex. Monkey J has two electrode arrays, one implanted in primary motor cortex (`R(i).spikeRaster`) and the other implanted in dorsal premotor cortex (`R(i).spikeRaster2`). Each is an $96 \times T_i$ matrix of spiking activity. Each of the 96 rows corresponds to a different electrode.

Like we said in class, a spike has a width of 1 ms and therefore, no more than 1 spike can occur every ms. Each `spikeRaster` is therefore an array that tells if there was a spike or not every millisecond. It has only 0's and 1's in it. To save memory, we store it as a sparse matrix but you can make it a normal matrix by calling `full(R(i).spikeRaster))`. (It is more memory efficient to store the locations of the non-zero values in the matrix than to store the entire matrix.) To be concrete, if the value of row n and column t is 1, then it indicates that a neuron spiked on electrode n at time t. If the value of this entry is 0, it indicates that no spike occurred on electrode n at time t.

Let's take a look at what the spike raster matrix looks like.

```
figure;

imagesc([R(1).spikeRaster; R(1).spikeRaster2]);
xlabel('Time [ms]');
ylabel('Neuron number');

print -dpdf spikeraster.pdf
```



In this spike raster, blue indicates 0's and the yellow dots indicate ones. You can see here a rough picture of how each neuron spikes.

Now, we see we have our kinematics data, `cursorPos`, and our neural data, `spikeRaster`. Our decoder is going to predict kinematic data from neural data. But remember, in class, we said the neural data needs to be binned (i.e., we count the number of spikes in a given bin width; for this project, we'll make the bin width 25 ms). That's how we interpret neural data – as spike rates.

Binning the data

We provide you with the `bin` function on CCLE (it is called `bin.m`), and for convenience, we have copy and pasted it here. Please make sure you understand the code below.

```
function binX = bin(X, binWidth, binType)
% binX = bin(X, binWidth, binType)
%
% assumes X is formatted NxT
%
% binType can be: 'sum', 'mean', 'first', 'last'

    if ~regexp(binType, 'sum|mean|first|last')
        error('bin:binType', 'Wrong binType entered');
    end

    [dims numSamples] = size(X);

    if strcmp(binType, 'first')
        numBins = ceil(numSamples / binWidth);
    else
        numBins = floor(numSamples/binWidth);
    end

    binX = zeros(dims, numBins);

    for i = 1 : numBins

        binStart = (i - 1)*binWidth + 1;
        binStop  = i*binWidth;

        switch binType
            case 'sum'
                binX(:, i) = nansum( X(:, binStart : binStop), 2);
            case 'mean'
                binX(:, i) = nanmean( X(:, binStart : binStop), 2);
            case 'first'
                binX(:, i) = X(:, binStart);
            case 'last'
                binX(:, i) = X(:, binStop);
        end
    end
end
```

Looking carefully at this code, we would pass it an array that is $N \times T_i$, a `binWidth` (which we will set to 25, i.e., 25 ms) and a `binType`. The `binType` tells us what to do with the data in a `binWidth`. Choosing "sum" means we sum the values (this is how we bin the spike raster). Choosing "first" means we take the first sample in each bin. Choosing "last" means we take the last sample in each

bin. We'll make use of these to build and test our optimal linear estimator decoder.

Choosing training and testing data

As we discussed in class, we can't evaluate how good our decoder is by testing its performance on training data. As we make any model more complex, it will always do better on the training data. Therefore we must partition the data into training and testing data. To do so, we'll choose the first 400 trials as the training data, and the last 106 trials as testing data.

We do this as follows:

```
Rtrain = R(1:400);
Rtest = R(401:end);
```

We train our decoder using `Rtrain` and we test our decoder using `Rtest`.

Building the decoder

Now, we're equipped to build the decoder. As we choose a bin width of 25 ms, what we need to do is count the number of spikes in contiguous 25 ms bins, and also get the kinematics every 25 ms. For the kinematics, we will decode velocity. Let's deal with the neural data first.

To make the neural data, first we concatenate all the neural data, at 1 ms resolution (via the command `Y = [Rtrain.spikeRaster; Rtrain.spikeRaster2]` – check the output to make sure you know what's going on here). After this, we bin the millisecond resolution neural data, `Y`, by counting the spikes in non-overlapping 25 ms bins. We call this output `Y_bin`. To be clear, each row, corresponding to one electrode of neural data, should be binned in non-overlapping 25 ms bins. Name this variable `Y_bin`. Finally, we append a row of 1's at the bottom of `Y_bin` via:

```
Y_bin = [Y_bin; ones(1, size(Y_bin, 2))]
```

What this does is it creates a neuron that is always 1. This allows a bias to be fit, i.e., it's a constant term with some coefficient, b, much like we have $y = mx+b\cdot 1$. This allows a bias term to be fit.

The code to do this is below:

```
dt = 25;
Y = [Rtrain.spikeRaster; Rtrain.spikeRaster2];
Y_bin = bin(Y, dt, 'sum');
Y_bin = [Y_bin; ones(1, size(Y_bin, 2))];
```

Here, `Y_bin` should be 193×16465. This means there are 192 neurons (plus 1 bias term) and we have this data for 16465×25 milliseconds, or $4,116.25$ seconds, or 6.86 minutes of data.

Now, let's get the kinematic data. We decode velocity because there is a much stronger correlation between neural activity and velocity of the movement than the position of the movement. We calculate the corresponding hand velocities in 25 ms intervals by using a first order Euler approximation, i.e.,

$$v(t) = \frac{\texttt{R(i).cursorPos(:, t+25) - R(i).cursorPos(:, t)}}{0.025}.$$

Be sure you understand what's going on here. (Aside: If you observe the velocities we calculate below, you'll notice that they are not as smooth as they could be; this could be improved by using higher-order approximations to compute velocity – however, we wanted to keep the velocity calculation simple in this homework.) We sample these velocities every 25 ms, and call the resultant matrix of velocities `X_bin`. We discard the last bin that does not have 25 ms worth of data. If we have done everything correctly, then `size(Y_bin, 2)` should equal `size(X_bin,2)`.

```
dt = 25;
X = [Rtrain(1:400).cursorPos];
X_bin = bin(X, dt, 'first');
X_bin = (X_bin(1:2, 2:end) - X_bin(1:2, 1:end-1)) / dt;
```

To check that we have the same amount of time for `X_bin` and `Y_bin`, we do the following:

```
>> size(X_bin)

ans =

           2       16465

>> size(Y_bin)

ans =

         193       16465
```

And then at this point, now our goal is to build a decoder `L`, which is a 2×193 matrix. It transforms our neural data, `Y` (193-dimensional), into our kinematic data, `X` (2-dimensional), through the calculation `x_predicted = L * y`.

As in class, our solution to solve for `L` is to calculate

$$\begin{aligned} L &= X \cdot Y^{\dagger} \\ &= XY^T(YY^T)^{-1} \end{aligned}$$

In `MATLAB`, the command to calculate this is `L = X * pinv(Y)`. Let's build the decoder, and plot its decoded kinematic trajectories. Again, to test the decoder, we want to evaluate how it performs on test data. For each decoded velocity, we can calculate the position by integrating velocity. The following code below does all of this.

```
% this line below builds the decoder
L = X_bin * pinv(Y_bin);

% now, let's see how it does on testing data.
dt = 25;
decoded_positions = {};

for i = 1:numel(Rtest)

    % Get the new neural data we're going to decode.
    Ytest = [Rtest(i).spikeRaster; Rtest(i).spikeRaster2]; % extract
      test data
    Ytest = bin(Ytest, dt, 'sum'); % bin the data
```

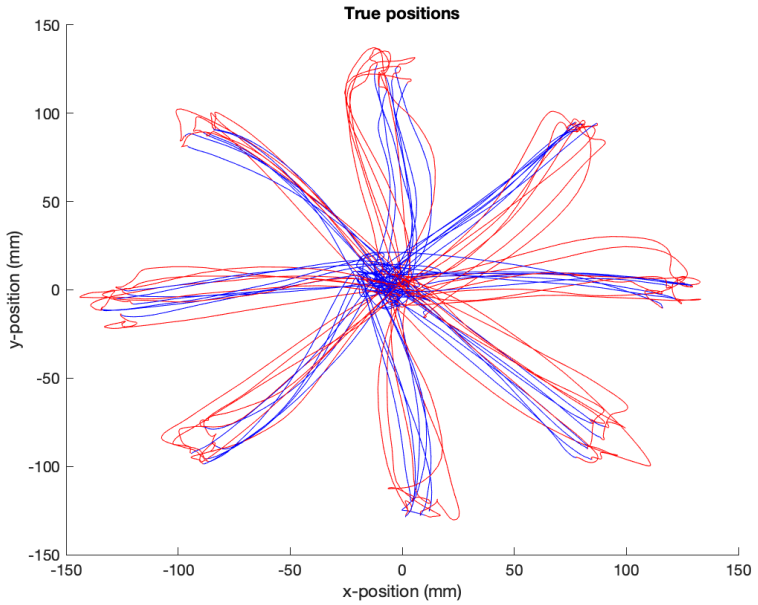
```
    Ytest = [Ytest; ones(1, size(Ytest, 2))]; % append 1's for the bias
      term

    % Decode the data.
    Xtest = L * Ytest;
    decodedPos = [Rtest(i).cursorPos(1:2, 1)]; % the trajectory starts
      at the same point; then we'll integrate velocities from this
      starting point.

    % Integrate the decoded velocities.
    for j = 1:size(Xtest, 2)
        decodedPos = [decodedPos decodedPos(:, end) + dt * Xtest(:, j)];
    end

    % store decoded positions plotting later.
    decoded_positions = {decoded_positions{:}, decodedPos};

end
```

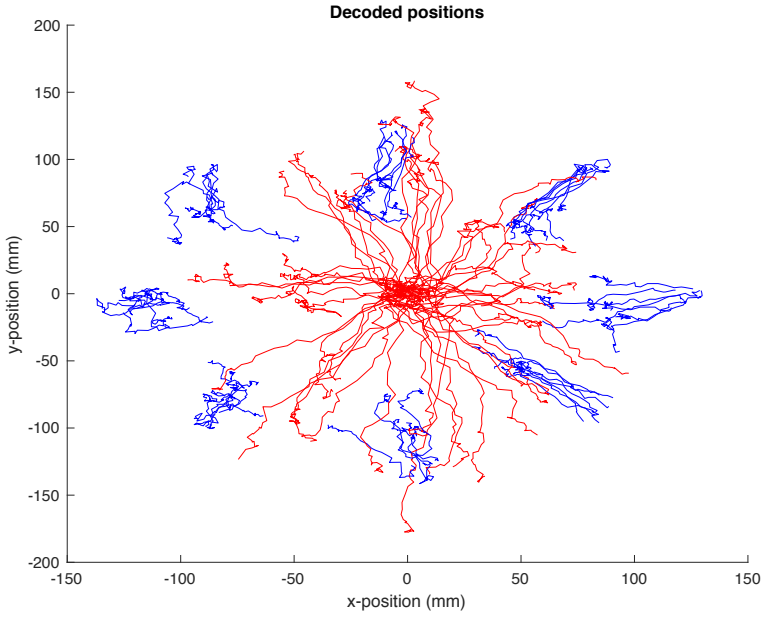
Now, let's plot these decoded positions.

```
%% plot the decoded positions.

% plot the true trajectories
% we'll color center-out reaches in red; center-back reaches in blue.
figure;
hold all;
xlabel('x-position (mm)');
ylabel('y-position (mm)');

for i = 1:length(Rtest)
    is_center_back = abs(R(i).target(1)) + abs(R(i).target(2)) == 0;
    if is_center_back
        plotcolor = 'b';
    else
        plotcolor = 'r';
    end
    plot(Rtest(i).cursorPos(1,:), Rtest(i).cursorPos(2,:), plotcolor);
end

% plot the decoded trajectories
figure;
hold all;
xlabel('x-position (mm)');
ylabel('y-position (mm)');

for i = 1:length(decoded_positions)
    is_center_out = abs(R(i).target(1)) + abs(R(i).target(2)) == 0;
    if is_center_out
        plotcolor = 'b';
    else
        plotcolor = 'r';
    end
    plot(decoded_positions{i}(1,:), decoded_positions{i}(2,:), plotcolor
      );
```

```
34 end
```

This generates the following plots.





You'll notice, this decoder does a fairly reasonable job at decoding center-out reaches, but not a great job at decoding center-back reaches.

We need a metric to quantify how good the decoder is. To do so, we'll calculate the squared error between the decoded positions and the true positions in `Rtest(i).cursorPos`. This metric is

called mean-squared error in the position, which is the error in reconstructing the trajectory. The following code calculates this:

```
%% calculate mean-square error

dt = 25;
mean_square_errors = [];
for i = 1:length(Rtest)
    true_traj = bin(Rtest(i).cursorPos(1:2,:), dt, 'first');
    decoded_traj = decoded_positions{i};

    % we may have decoded an extra dt relative to the true
    % trajectory, so we need to truncate to be sure.
    decoded_traj = decoded_traj(:, 1:size(true_traj, 2));

    mean_square_errors = [mean_square_errors mean(sum((true_traj -
      decoded_traj).^2, 1))];
end

mean_square_error = mean(mean_square_errors)
```

Running this code returns that the mean square error (MSE) is 3.6479e+03.

Project tasks

The goal of the project is to improve this mean-square error by trying different features of the neural data. Currently, we're only looking at the binned neural data. But what about different frequency features? What about other features?

For this project, we'd like you to implement at least 3 different features of the data. Please report if each feature increases or decreases the mean-square error. It's okay if it makes the decoder worse – that's still useful information. I will suggest three that you can try. Implementation of these will earn an 'A' on the project. You are welcome to go above and beyond; very memorable projects that find new features and drastically reduce MSE may receive an 'A+'.

1. As we discussed in class, low-pass filtering the data (i.e., convolution with a Gaussian kernel) increased the performance. One way to extract low frequency features is by convolving the data with a Gaussian bell-curve. However, you could imagine other low pass filters (like a truncated sinc function, or even a rect). You can choose whatever causal low pass filter you want that makes the MSE as small as possible; we'll give full credit for any low pass filter.

2. What about intermediate or higher frequency features? How do these compare to low-frequency features? Choose a frequency feature and evaluate the decoder performance. We let you decide on which frequencies you'd like to extract. You may concatenate these to the existing `Y_bin`, i.e., if your features are `Y_new_features`, then you may build a decoder using `Y_bin_new = [Y_bin; Y_new_features]`.

3. What about features like the derivative of the neural data? What if instead of decoding from `Y_bin` we decoded from differences in the neural data, i.e., `Y_bin(:,2:end) - Y_bin(:, 1:end-1)`?

For your three (or more) interventions, write a brief report (exceeding no more than 4 pages) describing the feature, plots of the decoded positions, and mean-square error. You should still use an optimal linear estimator decoder. When you implement your feature, you're effectively changing `Y_bin` and then re-running the existing code. You're welcome to try other decoders if you like.

In addition to the report, please submit all code of the features you implemented. Submit both your project report and the code to Prof. Kao directly by e-mail: `kao@seas.ucla.edu`.