Homework 6, Problem 1 on reaching data and neural data

ECE C143A/C243A, Spring Quarter 2022, Prof. J.C. Kao, TAs T. Monsoor and W. Yu

Total: 20 points

Download the dataset `JR_2015-12-04_truncated2.mat` from BruinLearn.

This data structure contains simultaneous reaching (kinematic) data and neural data. A large part of this question will be getting acquainted with this data.

Loading the downloaded file will drop a variable called `R`, in your Python workspace. We will call this variable the R-dict (or sometimes R-struct).

It is an array of dictionaries, with each dictionary in the array corresponding to one trial performed by Monkey J. In this data, Monkey J is performing a reaching task in which he acquires a center target, and then a peripheral target. After acquiring the peripheral target, he comes back to acquire the center target, and then acquires another peripheral target. This task is called a "center-out-and-back" task as the monkey continuously reaches from the center to a peripheral target, and then back to the center.

This assignment incorporates the `scipy` package. If you do not have it installed, you can install it by running:

```
pip install scipy
```

You may also install a package compiler for scientific computing, like `anaconda`: https://www.anaconda.com/download/

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import scipy.special
import scipy.io as sio
import math
import nsp as nsp
# Load matplotlib images inline
%matplotlib inline
# Reloading any code written in external .py files.
%load_ext autoreload
%autoreload 2
data = sio.loadmat('JR_2015-12-04_truncated2.mat') # load the .mat file.
R = data['R'][0,:]
```

(a) (1 point) How many trials?

How many trials were performed by Monkey J in this R-struct?

In [2]:
```
#======================================================#
# YOUR CODE HERE:
#   Calculate the number of trials in the R-dict
#======================================================#
print(len(R))
#======================================================#
# END YOUR CODE
#======================================================#
```

```
506
```

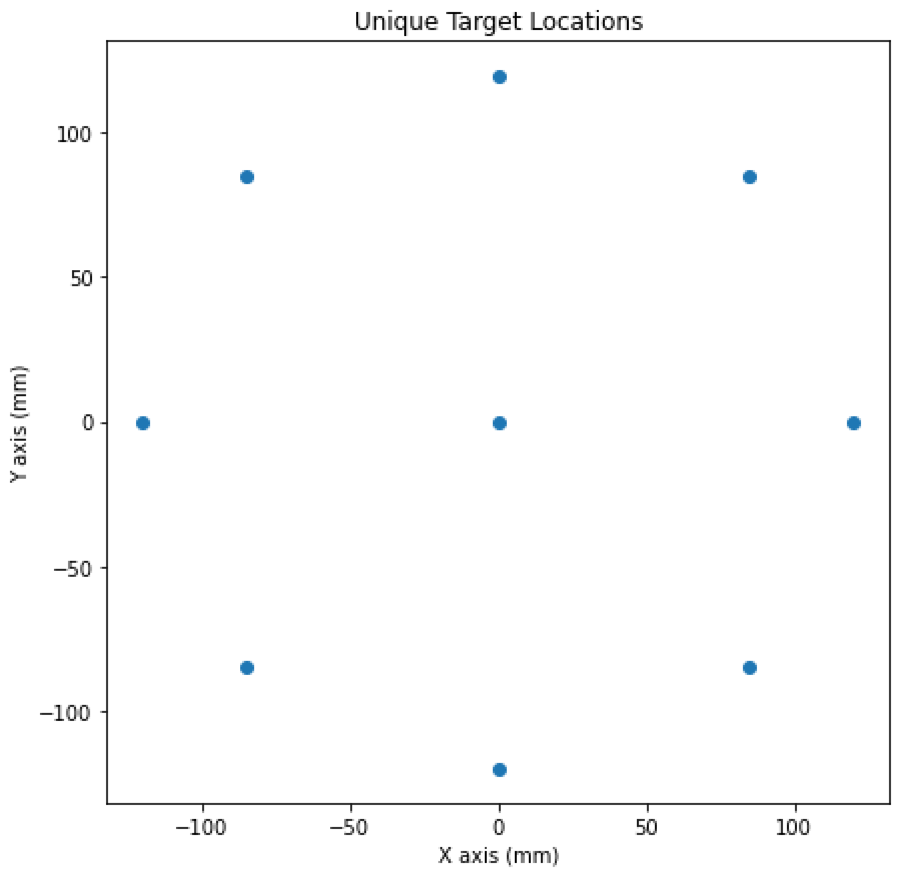
Answer: 506 trials

(b) (2 point) Where are the targets?

`R[i]['target']` is the target that Monkey J reached to on trial `i`. How many unique targets are there? Please provide a 2D plot of all the target locations. Ignore the 3rd dimension (z-location) which is by default set at -70 (i.e., it was not used). The units of `R[0,i]['target']` are in millimeters.

In [3]:
```
plt.figure(figsize=(7,7))
#======================================================#
# YOUR CODE HERE:
#   Generate 2D plot of all the target locations
#======================================================#
unique_locations = []
target_x = []
target_y = []
for trial in range(len(R)):
    location = [R[trial]['target'][0], R[trial]['target'][1]]
    if location not in unique_locations:
        unique_locations.append(location)
        target_x.append(R[trial]['target'][0])
        target_y.append(R[trial]['target'][1])

plt.scatter(target_x, target_y)
plt.xlabel("X axis (mm)")
plt.ylabel("Y axis (mm)")
plt.title("Unique Target Locations")
print(len(unique_locations))
int_arr = []
for x in range(len(target_x)):
 int_arr.append(target_x[x][0])
print(int_arr)

#======================================================#
# END YOUR CODE
#======================================================#
plt.show()
```

```
9
[0, 0, -84.85000000000001, 120, -84.85000000000001, -120, 0, 84.85000000000001, 84.85
000000000001]
```



Question: How many unique targets(including the center target) are there?

Answer: 9 unique targets

(c) (1 point) How many failures?

`R[i]['isSuccessful']` indicates if Monkey J successfully acquired the target on trial i. This means that he reaches to the target and holds his position within a 4cm by 4cm box centered around the target for 500ms. How many failures are there in this R-struct?

Hint: `isSuccessful` attribute has the dtype of uint8. Be careful about data types; in general it's always good practice to perform operations on same data types. If the number of trials is encoded as an `int`, you should cast the `isSuccessful` variable to be an int. You may achieve this by calling `isSuccessful = np.asarray(R['isSuccessful'], dtype=int)`.

In [4]:
```
#=====================================================#
# YOUR CODE HERE:
#   Calculate the number of failures in the R-dict
#=====================================================#
failures = 0
for trial in range(len(R)):
    isSuccessful = np.asarray(R[trial]['isSuccessful'][0][0], dtype=int)
    if (isSuccessful == 0):
        failures +=1
print(failures)

#=====================================================#
```

```
# END YOUR CODE
#======================================================#
```

```
0
```

Question: How many failures are there in this R-struct?

Answer: There were 0 failures

(d) (2 points) Kinematic sampling. [Code is provided; but you still need to analyze the output and answer the question]

`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.
- The units are in millimeters.
- Ignore the 3rd row -- which is z position (set to -70 mm by default in this data).
- Each column represents 1 ms. That is, if $T_i = 1032$, this indicates that the trial lasted for 1032 milliseconds. Note that `R[i]['cursorPos'][:,-1]` and `R[i+1]['cursorPos'][:,0]` are also separated by 1 ms, so that the R-struct contains millisecond resolution data and no segments of time are unobserved.
- The kinematics in `R[0,i]['cursorPos']` are sampled from a system called 'Polaris' that tracks a bead taped on the monkey's finger.

Observe the values of `R[i]['cursorPos']`. Does the ''Polaris'' system sample the monkey's kinematics at 1 ms resolution (i.e., 1000 Hz)? If not, approximately at what frequency does Polaris sample the kinematics?

In [5]:
```
#set printoption so that you can view the whole array by call print
import sys
#np.set_printoptions(threshold=np.nan)
np.set_printoptions(threshold=sys.maxsize)
#======================================================#
# YOUR CODE HERE:
#   Analysis the frequency of sampling
#======================================================#
i=1
print(R[i]['cursorPos'][:,0:200])
first_pos_x = 1.73
last_pos_x = 3.22
last_pos = 0
samples_in_pos = 0
samples_per_pos = []
for sample in range(200):
    pos = R[i]['cursorPos'][0,sample]
    if ((pos == first_pos_x) or (pos == last_pos_x)):
        continue

    if (pos != last_pos):
        samples_per_pos.append(samples_in_pos)
        samples_in_pos = 1
    else:
        samples_in_pos += 1
    last_pos = pos

samples_per_pos.pop(0)
freq = 1/(sum(samples_per_pos)/len(samples_per_pos))*1000
print(freq)


# we just print 0:200 because there are too many elements, 200 is enough to estimate t

#======================================================#
# END YOUR CODE
#======================================================#
```

```
[[  1.73   1.73   1.73   1.73   1.73   1.73   1.73   1.73   1.73   1.73
    1.96   1.96   1.96   1.96   1.96   1.96   1.96   1.96   1.96   1.96
    1.96   1.96   1.96   1.96   1.96   1.96   2.23   2.23   2.23   2.23
    2.23   2.23   2.23   2.23   2.23   2.23   2.23   2.23   2.23   2.23
    2.23   2.23   2.23   2.47   2.47   2.47   2.47   2.47   2.47   2.47
    2.47   2.47   2.47   2.47   2.47   2.47   2.47   2.47   2.47   2.47
    2.66   2.66   2.66   2.66   2.66   2.66   2.66   2.66   2.66   2.66
    2.66   2.66   2.66   2.66   2.66   2.66   2.83   2.83   2.83   2.83
    2.83   2.83   2.83   2.83   2.83   2.83   2.83   2.83   2.83   2.83
    2.83   2.83   2.83   2.93   2.93   2.93   2.93   2.93   2.93   2.93
    2.93   2.93   2.93   2.93   2.93   2.93   2.93   2.93   2.93   2.93
    2.99   2.99   2.99   2.99   2.99   2.99   2.99   2.99   2.99   2.99
    2.99   2.99   2.99   2.99   2.99   2.99   3.05   3.05   3.05   3.05
    3.05   3.05   3.05   3.05   3.05   3.05   3.05   3.05   3.05   3.05
    3.05   3.05   3.05   3.11   3.11   3.11   3.11   3.11   3.11   3.11
    3.11   3.11   3.11   3.11   3.11   3.11   3.11   3.11   3.11   3.11
    3.19   3.19   3.19   3.19   3.19   3.19   3.19   3.19   3.19   3.19
    3.19   3.19   3.19   3.19   3.19   3.19   3.24   3.24   3.24   3.24
    3.24   3.24   3.24   3.24   3.24   3.24   3.24   3.24   3.24   3.24
    3.24   3.24   3.24   3.22   3.22   3.22   3.22   3.22   3.22   3.22]
 [  0.66   0.66   0.66   0.66   0.66   0.66   0.66   0.66   0.66   0.66
    0.64   0.64   0.64   0.64   0.64   0.64   0.64   0.64   0.64   0.64
    0.64   0.64   0.64   0.64   0.64   0.64   0.48   0.48   0.48   0.48
    0.48   0.48   0.48   0.48   0.48   0.48   0.48   0.48   0.48   0.48
    0.48   0.48   0.48   0.36   0.36   0.36   0.36   0.36   0.36   0.36
    0.36   0.36   0.36   0.36   0.36   0.36   0.36   0.36   0.36   0.36
    0.26   0.26   0.26   0.26   0.26   0.26   0.26   0.26   0.26   0.26
    0.26   0.26   0.26   0.26   0.26   0.26   0.3    0.3    0.3    0.3
    0.3    0.3    0.3    0.3    0.3    0.3    0.3    0.3    0.3    0.3
    0.3    0.3    0.3    0.31   0.31   0.31   0.31   0.31   0.31   0.31
    0.31   0.31   0.31   0.31   0.31   0.31   0.31   0.31   0.31   0.31
    0.34   0.34   0.34   0.34   0.34   0.34   0.34   0.34   0.34   0.34
    0.34   0.34   0.34   0.34   0.34   0.34   0.37   0.37   0.37   0.37
    0.37   0.37   0.37   0.37   0.37   0.37   0.37   0.37   0.37   0.37
    0.37   0.37   0.37   0.42   0.42   0.42   0.42   0.42   0.42   0.42
    0.42   0.42   0.42   0.42   0.42   0.42   0.42   0.42   0.42   0.42
    0.46   0.46   0.46   0.46   0.46   0.46   0.46   0.46   0.46   0.46
    0.46   0.46   0.46   0.46   0.46   0.46   0.5    0.5    0.5    0.5
    0.5    0.5    0.5    0.5    0.5    0.5    0.5    0.5    0.5    0.5
    0.5    0.5    0.5    0.49   0.49   0.49   0.49   0.49   0.49   0.49]
 [-70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.
  -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.
  -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.
  -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.
  -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.
  -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.
  -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.
  -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.
  -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.
  -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.
  -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.
  -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.
  -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.
  -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.
  -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.
  -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.
  -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.
  -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.
  -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.
  -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.   -70.  ]]
60.24096385542168
```

Question: Does the 'Polaris' system sample the monkey's kinematics at 1 ms resolution (i.e., 1000Hz)? If not, approximately at what frequency does Polaris sample the kinematics?

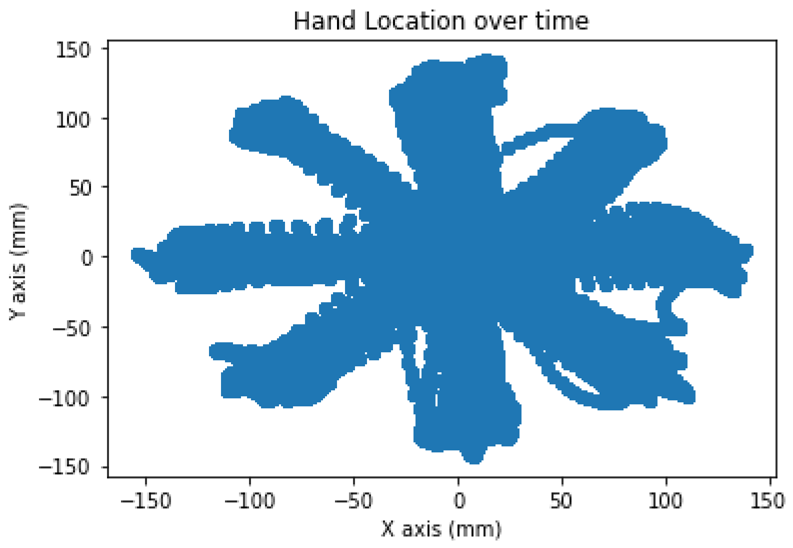
Answer: No they sample Around 60Hz (~17ms res)

(e) (1 points) Reach trajectories.

Present a 2D plot showing Monkey J's hand position across the entire R-dict. Does it look like the plot of a center-out-and-back task?

In [6]:

```
#=====================================================#
# YOUR CODE HERE:
#   Generate 2D plot of hand position
#=====================================================#
x_hand_pos = []
y_hand_pos = []
for trial in range(len(R)):
    for hand_pos in range(len(R[trial]['cursorPos'][0, :])):
        x_hand_pos.append(R[trial]['cursorPos'][0,hand_pos])
        y_hand_pos.append(R[trial]['cursorPos'][1,hand_pos])
plt.scatter(x_hand_pos, y_hand_pos)
plt.xlabel("X axis (mm)")
plt.ylabel("Y axis (mm)")
plt.title("Hand Location over time")
plt.show()

#=====================================================#
# END YOUR CODE
#=====================================================#
```



Question: Does it look like the plot of a center-out-and-back task?

Answer: Yes it does

(f) (1 points) Number of electrodes.

The R-struct contains neural data recorded from electrode arrays implanted in Monkey J's motor cortex.

We'll only consider `R[i]['spikeRaster']`, which is an $N \times T_i$ matrix of activity from primary motor cortex (M1).

(There is an `R[i]['spikeRaster2']`, which we will ignore. It contains activity from PMd.)

There are N electrodes, and each electrode is measuring spiking activity. Each row of `R[i]['spikeRaster']` corresponds to the spiking activity, at millisecond resolution, for Monkey J. 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.

Note that we store `R[i]['spikeRaster']` as a sparse matrix, since a neuron is often not spiking (corresponding to 0's).

It is more memory efficient to store the locations of the non-zero values in the matrix than to store the entire matrix.

In Python, one may use the command `.todense()` to make this a standard matrix.

How many electrode channels are there?

In [7]:
```
#=====================================================#
# YOUR CODE HERE:
#   Calculate the number of electrode channels
#=====================================================#
print(len(R[0]['spikeRaster'].todense()))
#=====================================================#
# END YOUR CODE
#=====================================================#
```

```
96
```

Question: How many electrode channels are there?

Answer: There are 96 electrodes

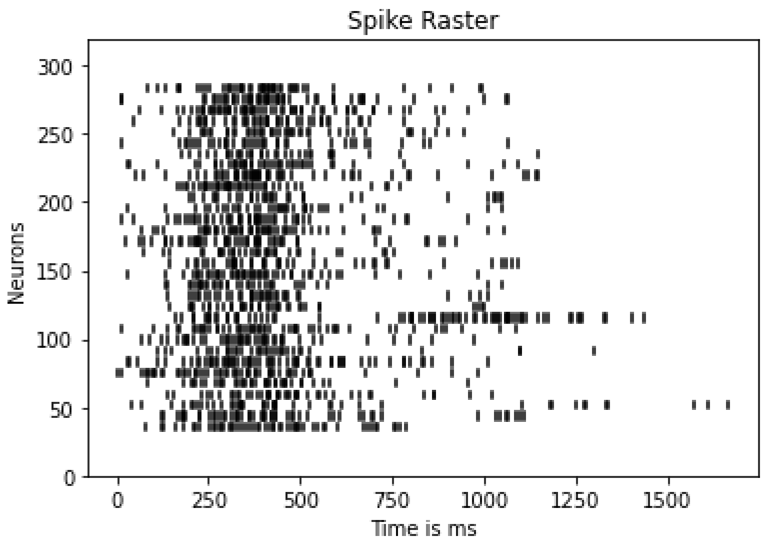
(g) (4 points) Spike raster. [Code solution provided; please understand it]

Plot a spike raster of all reaches to the right target, located at (120 mm,0 mm), for electrode 17. (Hint: use the `nsp.PlotSpikeRaster` function in `nsp.py`.)

In [8]:
```
#=====================================================#
# YOUR CODE HERE:
#  Generate spike raster of all reaches to the right target
#=====================================================#
spike_train = np.empty((0,0),dtype = list)
f = 0
```

```
for i in range(len(R)):
    target = R[i]['target'][0:2]
    if target[0]==120 and target[1]== 0 :
        spike_train = np.append(spike_train,0)
        spike_train[f] =  R[i]['spikeRaster'][16,:].todense().nonzero()[1]
        f = f + 1
nsp.PlotSpikeRaster(spike_train)
plt.xlabel('Time is ms')
plt.ylabel('Neurons')
plt.title('Spike Raster')
#=====================================================#
# END YOUR CODE
#=====================================================#
```

Out[8]:
```
Text(0.5, 1.0, 'Spike Raster')
```



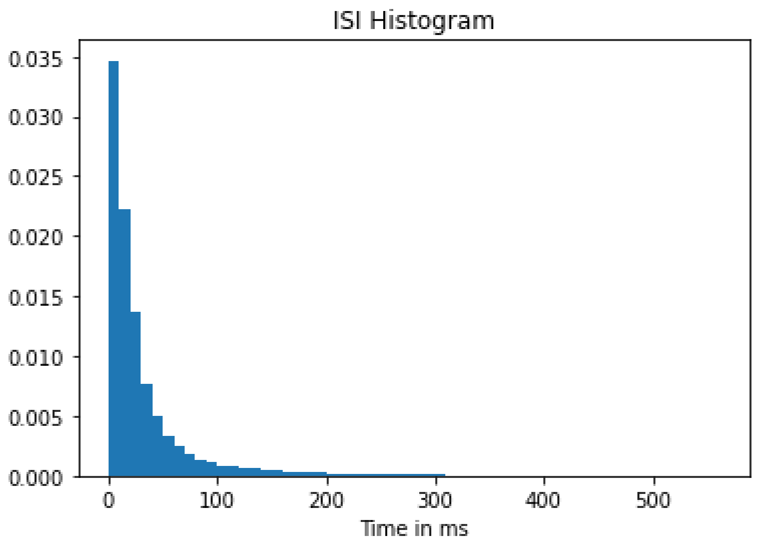
(h) (4 points) ISI distribution.

Plot the ISI distribution for electrode 17 across all trials in the R-dict (i.e., one ISI histogram with data from all reaches).
Make the ISI histogram bins 10 ms wide. Did we spike sort this neural data?

In [9]:
```
#=====================================================#
# YOUR CODE HERE:
#   Generate the ISI distribution for electrode 17
#=====================================================#

ISI = []
for x in range(len(R)):
    prev_time = 0
    for time in range(len(R[x]['spikeRaster'][16,:].todense().nonzero()[1])):
        curr_time = R[x]['spikeRaster'][16,:].todense().nonzero()[1][time]
        ISI.append((curr_time-prev_time))
        prev_time = curr_time

plt.hist(ISI, bins = range(min(ISI), max(ISI)+ 10, 10), density = True)
plt.xlabel("Time in ms")
plt.title("ISI Histogram")
#=====================================================#
```

```
# END YOUR CODE
#====================================================#
```

Out[9]:
```
Text(0.5, 1.0, 'ISI Histogram')
```



Question: Did we spike sort this neural data?

Answer: No becuase refractory periods were violated as there are spikes counted as spiking in <10ms.

(i) (4 points) PSTH.

Plot the average firing rate for each of 8 unique peripheral reach directions for electrode 17.

To do so, we have provided a binning function, `nsp.bin` which takes a spike raster matrix and counts the spikes in non-overlapping windows of some length Δt (to be specified as an input) for each neuron.

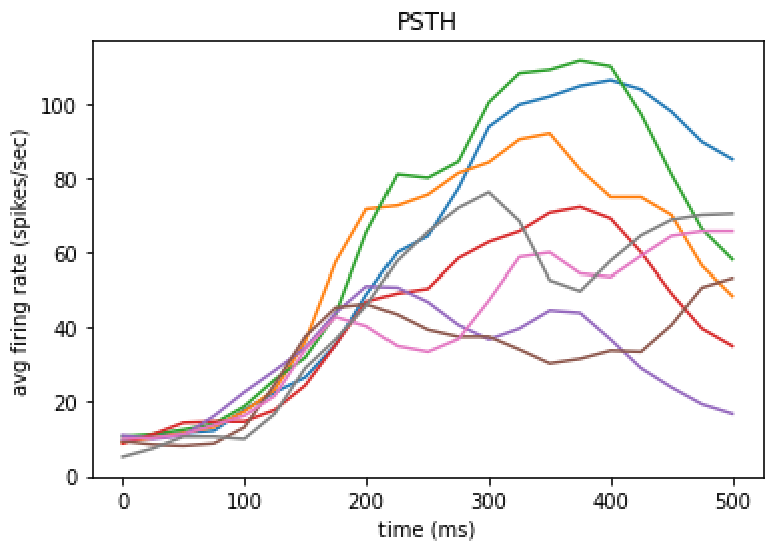
Do the following:

- Call `nsp.bin` in the correct way to bin the spikes in non-overlapping 25 ms bins.
- Then, for each of the 8 unique center-out reach conditions, average the binned spike counts for electrode 17, from $t = 0$ to 500 ms.
- For each of these traces, smooth them by using smooth function in `nsp.smooth` with `window_len = 5`, i.e., if one of these 8 traces was stored as the variable `trace`, you would call `trace = nsp.smooth(trace, window_len = 5)`.
- You should now have 8 traces of averaged binned spike counts for electrode 17.
- Plot these average, smoothed, firing rates through time on the same plot (x-axis time in ms, y-axis firing rate; be sure to get the unit scale correct. Use a different color for each of the 8 reach conditions. Note, these trial-averaged plots of the firing rate are often called peri-stimulus time histograms, or PSTHs. We do this to visualize the data and be sure it makes sense.

In [10]:
```
#====================================================#
```

```
# YOUR CODE HERE:
#   Perform the above described tasks to generate a PSTH.
#====================================================#
time_bins = list(range(0, 525, 25))
num_bins = (len(time_bins))
sum_rates = np.zeros([8,num_bins])
freq_seen = [0, 0, 0, 0, 0, 0, 0, 0]
last_idx = 0
for trial in range(len(R)):
    x = round(R[trial]['target'][1][0])
    y = round(R[trial]['target'][0][0])

    if (x== 0 and y == 0):
        continue
    elif (x== 120 and y == 0):
        idx = 0
    elif (x== 85 and y == 85):
        idx = 1
    elif (x== 0 and y == 120):
        idx = 2
    elif (x== -85 and y == 85):
        idx = 3
    elif (x== -120 and y == 0):
        idx = 4
    elif (x== -85 and y == -85):
        idx = 5
    elif (x== 0 and y == -120):
        idx = 6
    elif (x== 85 and y == -85):
        idx = 7
    spike_raster = R[trial]['spikeRaster'][16,:]
    bins = np.squeeze(nsp.bin(spike_raster, 25, 'sum'))
    bins = np.array(bins[:num_bins])
    sum_rates[idx] = np.add(sum_rates[idx], bins)
    freq_seen[idx] += 1

idx = 0
for rates in sum_rates:
    rates = nsp.smooth(rates/(freq_seen[idx]* .025), window_len = 5)
    plt.plot(time_bins, rates )
    idx += 1

plt.xlabel("time (ms)")
plt.ylabel("avg firing rate (spikes/sec)")
plt.title("PSTH")
#====================================================#
# END YOUR CODE
#====================================================#
```

```
Out[10]: Text(0.5, 1.0, 'PSTH')
```


PSTH
avg firing rate (spikes/sec)
time (ms)
0
20
40
60
80
100
0
100
200
300
400
500

```
In [ ]:
```

Homework 6, Problem 2 on population vector and optimal linear estimator

ECE C143A/C243A, Spring Quarter 2022, Prof. J.C. Kao, TAs T. Monsoor and W. Yu

Total: 30 points. In this notebook, you will implement an optimal linear estimator decoder.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import scipy.special
import scipy.io as sio
import math
import nsp as nsp
import pdb
# Load matplotlib images inline
%matplotlib inline
# Reloading any code written in external .py files.
%load_ext autoreload
%autoreload 2
data = sio.loadmat('JR_2015-12-04_truncated2.mat') # load the .mat file.
R = data['R'][0,:]
```

(a) (4 points) Tuning curve for one neuron

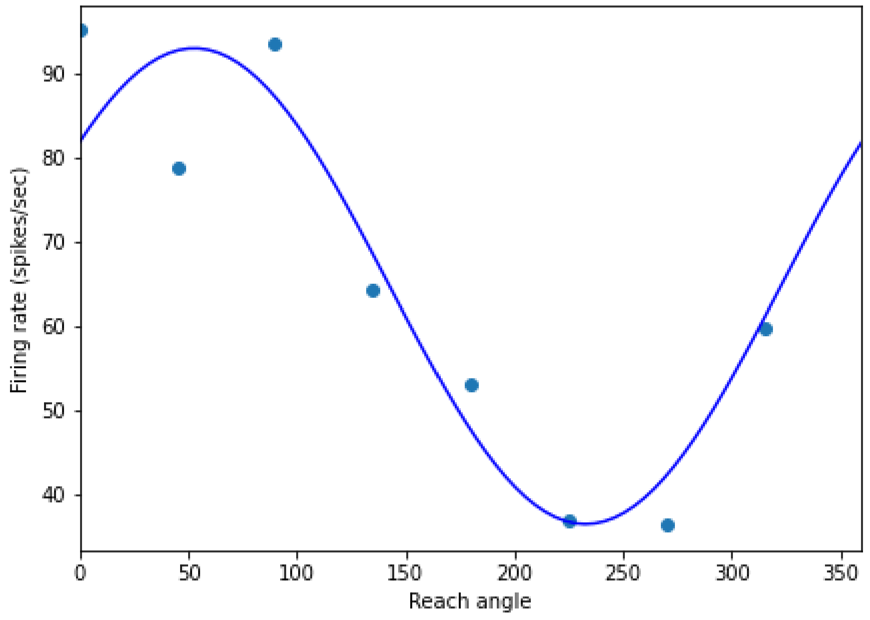
Fit a cosine tuning curve for electrode 17. On the same plot, plot the average firing rate for electrode 17 to each of the 8 directions and the fitted tuning curve. The average firing rate should be computed in a window from [250, 500] ms after trial onset. Reach angle should be on the x-axis and firing rate on the y-axis.

Hint: You may reuse code from HW #2 about tuning curve fitting to do this question. We have provided two functions: calculateFR() and tuning() which ought simplify this question. Take a look at what they do before calling them. The function tuning returns the tuning curve of a neuron (i.e., the parameters of the tuning curve c_0, c_1, θ_0 such that $f(\theta) = c_0 + c_1 cos(\theta - \theta_0)$

In [2]:
```
def calculateFR(i, neuron_idx=16, window=[250,500]):
    # Calculates the firing rate of neuron_idx on trial i in the given window range.
    # Output units are spikes/s
    return float(len(R[i]['spikeRaster'][neuron_idx,window[0]:window[1]].todense().nor

def tuning(thetas, meanFRs):
    # Calculates the parameters of a tuning curve.
    F = np.concatenate((np.ones((np.size(thetas), 1)),np.matrix(np.sin(thetas)).T,np.m

    ks = np.linalg.pinv(F)*np.matrix(meanFRs).T
    c0 = ks[0]
    pd = np.arctan2(ks[1],ks[2])
    c1 = ks[1]/np.sin(pd)

    c0 = np.asarray(c0)[0][0]
    c1 = np.asarray(c1)[0][0]
    pd = np.asarray(pd)[0][0]
    return [c0,c1,pd]
```

In [3]:
```
plt.figure(figsize=(7,5))

#======================================================#
# YOUR CODE HERE:
#   Fit the cosine tuning curve for electrode 17 and plot
#     1) the average firing rate for electrode 17 to each of the 8 directions and
#     2) the fitted tuning curve.
#======================================================#
unique_theta = []
target_x = []
target_y = []
thetas_ang = [0, 45, 90, 135, 180, 225, 270, 315]
thetas_rad = [0, np.pi/4, np.pi/2, np.pi*3/4, np.pi, np.pi*5/4, np.pi*3/2, np.pi*7/4]
sums = [0, 0, 0, 0, 0, 0, 0, 0]
counts = [0, 0, 0, 0, 0, 0, 0, 0]
for trial in range(len(R)):

    x = round(R[trial]['target'][0][0])
    y = round(R[trial]['target'][1][0])


    if (x== 0 and y == 0):
        continue
    elif (x== 120 and y == 0):
        idx = 0
    elif (x== 85 and y == 85):
        idx = 1
    elif (x== 0 and y == 120):
        idx = 2
    elif (x== -85 and y == 85):
        idx = 3
    elif (x== -120 and y == 0):
        idx = 4
    elif (x== -85 and y == -85):
        idx = 5
    elif (x== 0 and y == -120):
        idx = 6
    elif (x== 85 and y == -85):
        idx = 7

    counts[idx] += 1
    sums[idx] += calculateFR(i = trial)
avg_rate = []

for num in range(len(sums)):
    avg_rate.append(sums[num]/counts[num])
c0, c1, pd = tuning(thetas_rad, avg_rate)

plt.scatter(thetas_ang, avg_rate)
theta = np.linspace(0, 2*np.pi, num = 80)
plt.plot(theta * 180/np.pi, c0 + c1*np.cos(theta - pd), 'b')


#======================================================#
# END YOUR CODE
#======================================================#
```

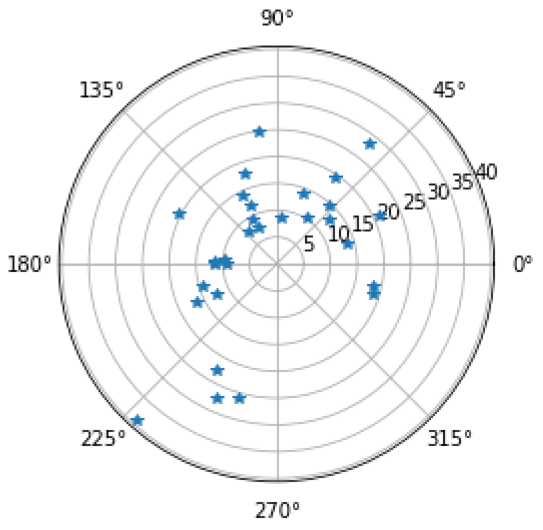
```
plt.xlabel('Reach angle')
plt.ylabel('Firing rate (spikes/sec)')
plt.xlim([0,360])
plt.show()
```



(b) (4 points) Preferred direction and modulation depth

Calculate the tuning parameters for every electrode. Pick the top 30 electrodes having the largest values of $|c_1|$ (where c_1, from homework #2, is from the tuning curve expression θ_0 such that $f(\theta) = c_0 + c_1 cos(\theta - \theta_0)$. The larger $|c_1|$, the greater the "modulation depth" of the electrode. This modulation depth is defined as the difference in firing rate between the preferred and anti-preferred directions. For each of these top 30 electrodes, plot the preferred direction in polar coordinates. The magnitude of each vector should be $|c_1|$. Do these top 30 neurons span the reaching space well?

In [4]:
```
#====================================================#
# YOUR CODE HERE:
#   Generate a polar plot for the top 30 electrodes where
#   the angle is the preferred direction of a neuron and
#   its magnitude is |c1|.
#
#   Hint: call plt.polar(top_30_angles, top_30_c1, '*')
#====================================================
full_electrode_info = []
full_c1 = []
for electrode in range(96):
    sums = [0, 0, 0, 0, 0, 0, 0, 0]
    counts = [0, 0, 0, 0, 0, 0, 0, 0]
    for trial in range(len(R)):
        x = round(R[trial]['target'][0][0])
        y = round(R[trial]['target'][1][0])
        if (x== 0 and y == 0):
            continue
        elif (x== 120 and y == 0):
            idx = 0
        elif (x== 85 and y == 85):
            idx = 1
        elif (x== 0 and y == 120):
            idx = 2
        elif (x== -85 and y == 85):
            idx = 3
        elif (x== -120 and y == 0):
            idx = 4
        elif (x== -85 and y == -85):
            idx = 5
        elif (x== 0 and y == -120):
            idx = 6
        elif (x== 85 and y == -85):
            idx = 7

        counts[idx] += 1
        sums[idx] += calculateFR(i = trial,neuron_idx=electrode )
    avg_rate = []

    for num in range(len(sums)):
        avg_rate.append(sums[num]/counts[num])
    c0, c1, pd = tuning(thetas_rad, avg_rate)
    full_electrode_info.append([c0, c1, pd])
    full_c1.append(abs(c1))
top_idx = sorted(range(len(full_c1)), key=lambda i: full_c1[i])[-30:]
top_30_c1 = []
top_30_theta = []
for index in top_idx:
    top_30_c1.append(full_electrode_info[index][1])
    top_30_theta.append(full_electrode_info[index][2])
plt.polar(top_30_theta, top_30_c1, '*' )
plt.show()
#=====================================================#
# END YOUR CODE
#=====================================================#
```



Question: Do these top 30 neurons span the reaching space well?

Answer: They seem to be a little biased with no electrodes having prefferered directions betweeen 270-315 degrees.

(c) (4 points) Preparing kinematics for optimal linear estimator. [Code solution provided; please understand it and answer the question.]

Fit the preferred directions by building an optimal linear estimator. We'll use the first 400 trials in the R-struct for training and reserve the last 106 trials for testing our decoder in part (g). Concatenate all the neural data, at 1 ms resolution
(hint: use the command `Y = scipy.sparse.hstack(R[0:400]['spikeRaster'])`).

Next, bin the millisecond resolution neural data, Y (representing the concatenated millisecond activity across the 400 trials), by counting the spikes in non-overlapping 25 ms bins. (Hint: use the `nsp.bin` function we provided.)

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`. Append a row of 1's at the bottom of `Y_bin` via: `Y_bin= np.vstack((Y_bin,np.ones(np.size(Y_bin,1))))` to allow a bias term to be fit.

Similarly, calculate the corresponding hand velocities in 25 ms intervals by using a first order Euler approximation, i.e.,

$$v(t) = \frac{cursorPos[t+25] - cursorPos[t]}{0.025}$$

(Hint: Like the neural data, concatenate all the cursor positions, at 1 ms resolution, by using the command `X =scipy.sparse.hstack(R[0:train_num]['cursorPos'])`, then sample these velocities every 25 ms, and call the resultant matrix of velocities `X_bin`. Discard the last bin that does not have 25 ms worth of data.).

Note that if you have done everything correctly, then `np.size(Y_bin, 1)` should equal `np.size(X_bin, 1)`. What are the dimensions of `Y_bin` and `X_bin`?

(Aside: If you observe the velocities, 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.)

In [5]:
```
#======================================================#
# YOUR CODE HERE:
#   Bin the data.
#======================================================#
dt = 25
binnedR = np.empty((0,0),dtype = list)
train_num = 400
Y = scipy.sparse.hstack(R[0:train_num]['spikeRaster'])
X = scipy.sparse.hstack(R[0:train_num]['cursorPos'])
X = scipy.sparse.csc_matrix(X)
Y_bin = nsp.bin(Y, dt,'sum')
Y_bin = np.vstack((Y_bin, np.ones(np.size(Y_bin,1))))

X_bin = nsp.bin(X, dt,'first')
X_bin = np.diff(X_bin[0:2,:])/dt*1000
```

```
#========================================================#
# END YOUR CODE
#========================================================#
```

In [6]:
```
print("size of Y_bin:", Y_bin.shape)
print("size of X_bin:", X_bin.shape)
```

```
size of Y_bin: (97, 16465)
size of X_bin: (2, 16465)
```

Question: What are the dimensions of `Y_bin` and `X_bin`?

Answer: Dimensions Y_bin = (97, 16465), Dimensions X_bin = (2, 16465)

(d) (6 points) Fitting the parameters in an OLE

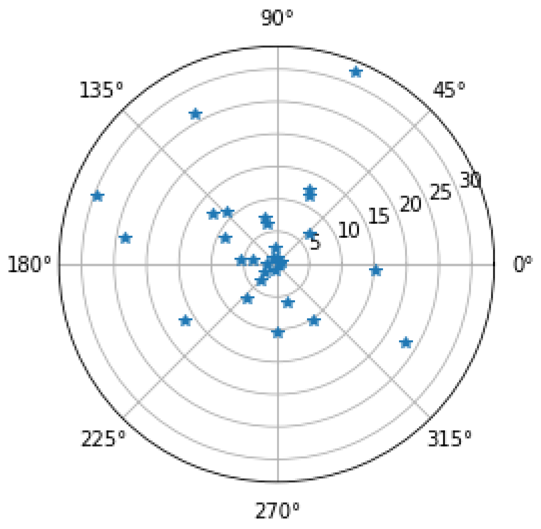
Fit the optimal linear estimator that takes your spike counts binned at 25 ms resolution, `Y_bin`, and your sampled velocities every 25 ms, `X_bin`, and calculates the optimal linear decoder via:

$$L = X_{bin} * Y_{bin}^{\dagger}$$

where $Y_{bin}^{\dagger}$ is the pseudo inverse of Y_{bin}, you can use `scipy.linalg.pinv` in python to calculate the pseudo inverse.
Consider the top 30 electrodes in part (b). For each of these 30 electrodes, we want to visualize their preferred direction and modulation depth (like we did for the cosine fits of preferred direction). If electrode i is one of these 30 electrodes, plot `L[:, i]` in polar coordinates. (Think about why we can think of `L[:,i]` as a preferred direction and magnitude.) Does this plot look similar to the plot you derived in part (b)?

In [7]:
```
#========================================================#
# YOUR CODE HERE:
#   Fit the OLE matrix L via least squares, and visualize
#   L[:,i] for the top 30 electrodes found in part (b) in
#   an analogous polar plot.
#========================================================#
Y_pseudo = scipy.linalg.pinv(Y_bin.astype(float))
L = np.matmul(X_bin, Y_pseudo)
print(L[:, 0])
print(top_idx)
L_angle = []
L_mag = []
for index in top_idx:
    L_angle.append(L[:, index][0])
    L_mag.append(abs(L[:, index][1]))

plt.polar(L_angle,L_mag, '*' )
plt.show()
#========================================================#
# END YOUR CODE
#========================================================#
```

```
[-7.27640169321345 10.441816540291171]
[49, 6, 41, 34, 73, 35, 93, 59, 58, 37, 33, 55, 76, 19, 64, 70, 25, 46, 40, 78, 72, 2
6, 0, 65, 14, 63, 20, 61, 16, 32]
```



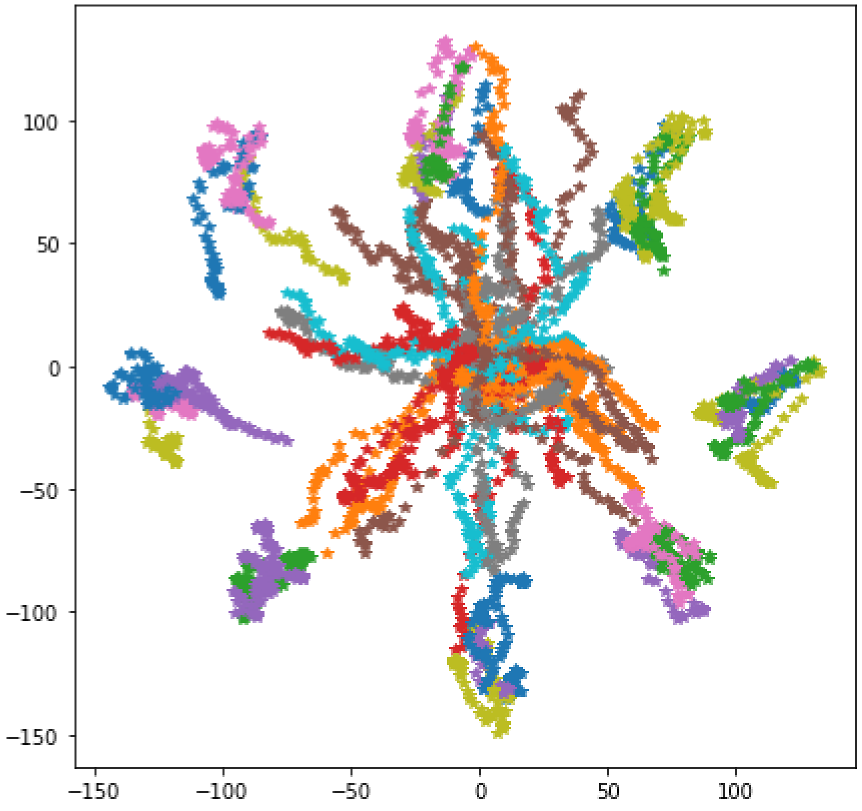
Question: Does this plot have better coverage of the entire workspace than the plot you derived in part (b)?

Abswer: Yes it appears to be more evenly spread than in b

(e) (6 points)

Using the matrix L you derived in part (d), we will now decode new neural data. Using the last 106 trials, we'll evaluate how good our optimal linear estimator is. We are going to operate trial by trial. For each trial in the R-struct, bin the neural data at 25 ms resolution. Remember to append a row of 1's at the bottom of the neural data. Use the matrix L to decode the neural activity and get decoded velocities for each trial. On the same plot, show the decoded trajectory for each trial. Use `R[i]['cursorPos'][0:2,0]` as the starting cursor position on trial i, and find position by integrating the decoded velocity. Do the trajectories appear normal, or are there idiosyncracies in the decoder?

In [8]:
```python
plt.figure(figsize=(7,7))
#=====================================================#
# YOUR CODE HERE:
#   Decode data for the last 106 trials, and plot the
#   decoded positions.  If pos is a 2 x T array holding
#   all the decoded positions, you may plot them via:
#
#     plt.plot(pos[0,:], pos[1,:], '.')
#
#=====================================================#
dt = 25
for trial in range(400, 506, 1):
    Y  = R[trial]['spikeRaster']
    X = R[trial]['cursorPos']
    Y_bin = nsp.bin(Y, dt, 'sum')
    Y_bin = np.vstack((Y_bin, np.ones(np.size(Y_bin, 1))))
    #print(Y_bin)
    rates = np.matmul(L,Y_bin)
    step = np.cumsum(rates, axis = 1)*dt/1000
    step = np.swapaxes(step, 0, 1)
    step[:,0] += X[0,0]
    step[:,1] += X[1,0]
    plt.plot(step[:,0], step[:,1], '*')


#====================================================#
# END YOUR CODE
#====================================================#
```



Question: Do the trajectories appear normal, or are there idiosyncracies in the decoder?

Answer: There are idiosyncracies in the decoder. The plotting does not appear as continuos lines, but it is similar and recognizable shape.

(f) (3 points)

For the decoder in part (e), what is the average mean-square error in position per trial? (For each time bin in a trial, calculate the squared Euclidean distance between the decoded position and the true hand position. Average all these squared Euclidean distance differences across all time in the trial. Finally, average across all trials.)

In [9]:
```
#====================================================#
# YOUR CODE HERE:
#   Calculate the mean-squared error between the decoded
#   hand position and the true hand position.  Average
#   the squared errors across time; then average the squared
#   errors across trials.
#====================================================#
```

```
cum_dis = 0
dt = 25
for trial in range(400, 506, 1):
    dist = np.zeros(X.shape[1])
    Y  = R[trial]['spikeRaster']
    X = R[trial]['cursorPos']
    Y_bin = nsp.bin(Y, dt, 'sum')
    Y_bin = np.vstack((Y_bin, np.ones(np.size(Y_bin, 1))))
    rates = np.matmul(L,Y_bin)
    step = np.cumsum(rates, axis = 1)*dt/1000
    step = np.swapaxes(step, 0, 1)
    step = np.append([[0,0]], step, axis = 0)
    step[:,0] += X[0,0]
    step[:,1] += X[1,0]
    moves = np.arange(0,X.shape[1],dt)
    real_pos = X[0:2,:][:, moves]
    real_pos = np.swapaxes(real_pos, 0, 1)
    for i in range (real_pos.shape[0]):
        dist[i] = (real_pos[i,0] - step[i,0])**2 + (real_pos[i,1] - step[i,1])**2

    mse = np.sum(dist)/len(real_pos)
    cum_dis += mse
print(cum_dis/ 106)
#====================================================#
# END YOUR CODE
#====================================================#
```

```
5325.992074167322
```

Answer: MSE ~ 5326mm^2

```
In [ ]:
```

Homework 6, Problem 3 on Wiener filter

ECE C143A/C243A, Spring Quarter 2022, Prof. J.C. Kao, TAs T. Monsoor and W. Yu

Total: 15 points. In this notebook, you will implement an optimal linear estimator decoder.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import scipy.special
import scipy.io as sio
import math
import nsp as nsp
# Load matplotlib images inline
%matplotlib inline
# Reloading any code written in external .py files.
%load_ext autoreload
%autoreload 2
data = sio.loadmat('JR_2015-12-04_truncated2.mat') # load the .mat file.
R = data['R'][0,:]
```

(a) (4 points) Preparing training data for a Wiener filter. [Code solution provided; please understand it and answer the question.]

Now that we've built an optimal linear estimator, we'll extend it to incorporate history. We will train a Wiener filter, again using just the first 400 trials of the R-struct. We will use 100 ms worth of history in our Wiener filter. As each bin is 25 ms long, this corresponds to using data up to and including 4 bins in the past, i.e. $P = 3$. Make the large matrix Y_W discussed in Lecture. To be clear, we are talking about the matrix:

$$\begin{bmatrix} y_{P+1} & y_{P+2} & y_{P+3} & \cdots & y_K \\ y_P & y_{P+1} & y_{P+2} & \cdots & y_{K-1} \\ \cdots & & & & \\ y_1 & y_2 & y_3 & \cdots & y_{K-P} \\ 1 & 1 & 1 & \cdots & 1 \end{bmatrix}$$

Calculate this matrix and output its dimension.

In [2]:
```
#======================================================#
# YOUR CODE HERE:
#   Create the Y_W matrix and print its dimensions.
#======================================================#
dt = 25
train_num = 400
Y =scipy.sparse.hstack(R[0:train_num]['spikeRaster'])
X =scipy.sparse.hstack(R[0:train_num]['cursorPos'])
X= scipy.sparse.csc_matrix(X)
Y_bin = nsp.bin(Y, dt,'sum')

X_bin = nsp.bin(X, dt,'first')
X_bin = np.diff(X_bin[0:2,:])/dt*1000

X_small_bin = np.matrix(X_bin[:,3:],dtype=float)
last_index = np.size(X_bin,1)
Y1= np.matrix(Y_bin[:,3:],dtype= float)
Y2 =np.matrix(Y_bin[:,2:-1],dtype= float)
Y3 =np.matrix(Y_bin[:,1:-2],dtype= float)
Y4 =np.matrix(Y_bin[:,0:-3],dtype= float)
Y_large = np.vstack((Y1,Y2,Y3,Y4))

Y_w_bin = np.vstack((Y_large,np.ones(np.size(Y_large,1))))

print("The dimension for the large matrix",np.size(Y_w_bin,0),np.size(Y_w_bin,1))
#=====================================================#
# END YOUR CODE
#=====================================================#
```

```
The dimension for the large matrix 385 16462
```

Question: What are the dimensions of Y_W?

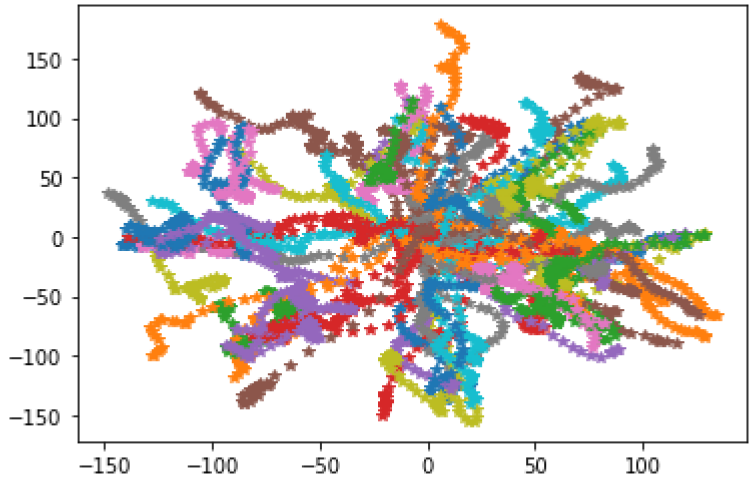
Answer: 385 x 16462

(b) (7 points) Fit a Wiener filter and decode activity.

Fit the Wiener filter using the data matrix you generated in part(a). Plot, as you did in question 2(g), the decoded trajectories for each test trial. (Hint: to decode the first 3 velocities of trial i at times 25 ms, 50 ms, and 75 ms, since 100 ms have not yet occurred in the trial, you will have to use the last bins of spiking activity in trial $i-1$.)

In [3]:

```
#=====================================================#
# YOUR CODE HERE:
#   Fit the Wiener filter and decode the trajectories for
#   each test trial.
#=====================================================#
dt = 25
Y_pseudo = scipy.linalg.pinv(Y_w_bin.astype(float))
L = np.matmul(X_small_bin, Y_pseudo)
for trial in range(400, 506, 1):
    Y = R[trial]['spikeRaster']
    X = R[trial]['cursorPos']

    Y_bin = nsp.bin(Y, dt,'sum')

    Y1= np.matrix(Y_bin[:,3:],dtype= float)
    Y2 =np.matrix(Y_bin[:,2:-1],dtype= float)
    Y3 =np.matrix(Y_bin[:,1:-2],dtype= float)
    Y4 =np.matrix(Y_bin[:,0:-3],dtype= float)
    Y_large = np.vstack((Y1,Y2,Y3,Y4))

    Y_w_bin_0 = np.vstack((Y_large,np.ones(np.size(Y_large,1))))

    rates = np.matmul(L,Y_w_bin_0)
    step = np.cumsum(rates, axis = 1)*dt/1000
    step = np.swapaxes(step, 0, 1)
    step[:,0] += X[0,0]
    step[:,1] += X[1,0]
    plt.plot(step[:,0], step[:,1], '*')



#=====================================================#
# END YOUR CODE
#=====================================================#
```



(c) (4 points) Calculate the mean-square error for the Wiener filter.

What is the average mean-square error in position per trial? Did the Wiener filter perform better or worse than the optimal linear estimator?

In [7]:
```
#=====================================================#
# YOUR CODE HERE:
#   Calculate the mean-squared error between the decoded
#   hand position and the true hand position.  Average
#   the squared errors across time; then average the squared
#   errors across trials.
#=====================================================#
cum_dis = 0
for trial in range(400, 506, 1):
    Y = R[trial]['spikeRaster']
    X = R[trial]['cursorPos']
    dist = np.zeros(X.shape[1])

    Y_bin = nsp.bin(Y, dt,'sum')

    Y1= np.matrix(Y_bin[:,3:],dtype= float)
    Y2 =np.matrix(Y_bin[:,2:-1],dtype= float)
    Y3 =np.matrix(Y_bin[:,1:-2],dtype= float)
    Y4 =np.matrix(Y_bin[:,0:-3],dtype= float)
    Y_large = np.vstack((Y1,Y2,Y3,Y4))
    Y_w_bin_0 = np.vstack((Y_large,np.ones(np.size(Y_large,1))))

    rates = np.matmul(L,Y_w_bin_0)
    step = np.cumsum(rates, axis = 1)*dt/1000
    step = np.swapaxes(step, 0, 1)
    step[:,0] += X[0,0]
    step[:,1] += X[1,0]
    moves = np.arange(0,X.shape[1],dt)
    real_pos = X[0:2,:][:, moves]
    real_pos = np.swapaxes(real_pos, 0, 1)
    dist = np.zeros(X.shape[1])
    for i in range(len(real_pos)-4):
        dist[i] = (real_pos[i,0] - step[i,0])**2 + (real_pos[i,1] - step[i,1])**2


    mse = np.sum(dist)/(len(real_pos)-4)
    cum_dis += mse
print(f'Mean square error is {cum_dis/ 106}mm^2')


#======================================================#
# END YOUR CODE
#======================================================#
```

```
Mean square error is 3240.8007003265834mm^2
```

Question: Does the WF do better or worse than the OLE?

Answer: It performs better as it has a smaller MSE

Homework 6, Problem 4 on Kalman filter

ECE C143A/C243A, Spring Quarter 2022, Prof. J.C. Kao, TAs T. Monsoor and W. Yu

Total: 35 points

In this question, we'll implement a Kalman filter for decoding neural activity. This will specifically be a velocity Kalman filter. We will be working with the same dataset that we used in prior questions. To begin, we'll first learn the dynamical system. Use the first 400 trials as training data and the remaining 106 trials as testing data.

In [1]:
```
# Importing the necessary packages and the data
import numpy as np
import matplotlib.pyplot as plt
import scipy.special
import scipy.io as sio
import math
import nsp as nsp
import pdb
# Load matplotlib images inline
%matplotlib inline
# Reloading any code written in external .py files.
%load_ext autoreload
%autoreload 2
data = sio.loadmat('JR_2015-12-04_truncated2.mat') # load the .mat file.
R = data['R'][0,:]
```

(a) (4 points) The A-matrix.

We will first learn the parameters of a linear dynamical system. We'll begin with the $\mathbf{A}$ matrix, which should obey laws of physics. Our linear dynamical system state at time k is be:

$$\mathbf{x}_k = \begin{bmatrix} p_x(k) \\ p_y(k) \\ v_x(k) \\ v_y(k) \\ 1 \end{bmatrix}$$

where $p_x(k), p_y(k), v_x(k)$, and $v_y(k)$ are the x-position, y-position, x-velocity, and y-velocity, respectively, at time k. We'll worry about only deriving an update law for the velocities.

Write what the $\mathbf{A}$ matrix looks like below if if $v_{xx} = 0.7$, $v_{yy} = 0.7$, $v_{yx} = 0$, and $v_{xy} = 0$ and we are using a bin width of 25 ms. Recall that the units of position are mm and assume that the velocities you are calculating are in m/s or equivalently mm/ms.

Answer:

A = [[1, 0, 25, 0, 0], [0, 1, 0, 25, 0], [0, 0, .7, 0, 0], [0, 0, 0, .7, 0], [0, 0, 0, 0, 1]]

(b) (4 points) Fit the A-matrix.

Calculate the hand velocities in 25 ms intervals by using a first order Euler approximation, i.e.,

$$v(t) = \frac{cursorPos[t+25] - cursorPos[t]}{25}$$

Find and report the values in the $\mathbf{A}$ matrix. To be clear, you should only be finding a matrix

$$\mathbf{A}_s = \begin{bmatrix} v_{xx} & v_{xy} \\ v_{yx} & v_{yy} \end{bmatrix}$$

and imputing those values into an $\mathbf{A}$ matrix that obeys physics.

In [2]:

```python
#=====================================================#
# YOUR CODE HERE:
#   Fit and report the 5x5 matrix A.
#=====================================================#
dt = 25
A = np.zeros((5,5))
X = scipy.sparse.hstack(R[0:400]['cursorPos'])
X = scipy.sparse.csc_matrix(X)
X_bin = nsp.bin(X,dt,'first')
X_bin = np.diff((X_bin[0:2, :])/dt)


A[0,0], A[1,1], A[4,4] = 1, 1, 1
A[0,2], A[1,3] = 25, 25

A_s = np.matrix(X_bin[:, 1:X_bin.shape[1]], dtype = float) * scipy.linalg.pinv(np.matr
A[2:4, 2:4] = A_s
print(A)

#=====================================================#
# END YOUR CODE
#=====================================================#
```

```
[[ 1.00000000e+00  0.00000000e+00  2.50000000e+01  0.00000000e+00
   0.00000000e+00]
 [ 0.00000000e+00  1.00000000e+00  0.00000000e+00  2.50000000e+01
   0.00000000e+00]
 [ 0.00000000e+00  0.00000000e+00  7.79769072e-01 -7.41686029e-03
   0.00000000e+00]
 [ 0.00000000e+00  0.00000000e+00  9.61320498e-03  7.80798314e-01
   0.00000000e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
   1.00000000e+00]]
```

Answer: See above print Matrix for A

(c) (4 points) Fit the C matrix.

For this question, we will only be using `R[i]['spikeRaster']` for the neural data (i.e., ignore `R[i].['spikeRaster2']`). Calculate the $\mathbf{C}$ matrix by only finding the coefficients mapping velocity (and the constant 1) to the neural data. (We do not calculate the coefficients corresponding to the mapping of position to neural data, since we are only decoding velocity. In a position-velocity Kalman filter that decodes both position and velocity, we would fit these coefficients.) Concretely, find a matrix $\mathbf{C}_s$, which is 96×3 and is the least-squares optimal mapping from:

$$\mathbf{y}_k = \mathbf{C}_s \begin{bmatrix} v_k^x \\ v_k^y \\ 1 \end{bmatrix}$$

Then, impute the values of $\mathbf{C_s}$ into the matrix $\mathbf{C}$, which initialized to be a matrix of zeros of size 96×5.
Thus, the first two columns of $\mathbf{C}$ should be all zeros, but the last three columns of $\mathbf{C}$ will be populated with the values in $\mathbf{C}_s$. Bin the neural data in non-overlapping 25 ms bins. Find the $\mathbf{C}$ matrix and report the value of `np.sum(C,0)`.

In [3]:
```
## Part c
#====================================================#
# YOUR CODE HERE:
#   Fit the C matrix, and report np.sum(C, 0)
#====================================================#
X_bin = np.vstack((X_bin, np.ones(np.size(X_bin,1))))
C = np.zeros((96,5))
Y = scipy.sparse.hstack(R[0:400]['spikeRaster'])
Y_bin = nsp.bin(Y,dt,'sum')

C_s = np.matrix(Y_bin, dtype = float)* scipy.linalg.pinv(np.matrix(X_bin, dtype = floa
C[:, 2:5] = C_s


print(np.sum(C,0))

#====================================================#
# END YOUR CODE
#====================================================#
```

```
[ 0.          0.         -4.49938145  1.70531105 40.19265294]
```

Answer: See above answer

(d) (4 points) Fit the W matrix.

Find the $\mathbf{W}$ using the $\mathbf{A}$ matrix calculated in part (b). We will only want to calculate an uncertainty over the velocity, and not on the positions. Thus, you will perform the covariance calculation over the velocities, resulting in a 2×2 matrix $\mathbf{W}_s$.
You will insert these values into the correct location in the $\mathbf{W}$ matrix, which is everywhere else 0.
Report the $\mathbf{W}$ matrix.

In [4]:
```
## Part d
#====================================================#
# YOUR CODE HERE:
#   Fit and report the W matrix.
```

```
#========================================================#
dt = 25
Y = scipy.sparse.hstack(R[0:400]['spikeRaster'])
Y_bin = nsp.bin(Y,dt,'sum')
K = np.size(Y_bin, 1)

X = scipy.sparse.hstack(R[0:400]['cursorPos'])
X = scipy.sparse.csc_matrix(X)
X_first_bin = nsp.bin(X[0:2],dt,'first')
X_k = np.diff(X_first_bin/dt)
#X_k = np.vstack((X_first_bin[:, 0:-1], X_bin, np.ones(np.size(X_bin,1))))


X = scipy.sparse.hstack(R[0:400]['cursorPos'])
X = scipy.sparse.csc_matrix(X)
X_bin = nsp.bin(X,dt,'last')
X_bin = np.diff((X_bin[0:2, :])/dt)
#Ax_k1 = np.matmul(A_s, np.matrix(X_bin[:,0:X_bin.shape[1]-1], dtype = float))

train_num = 400
dt = 25
X =scipy.sparse.hstack(R[0:train_num]['cursorPos'])
X = scipy.sparse.csc_matrix(X)
X_bin = nsp.bin(X, dt,'first')
X_bin = np.diff(X_bin[0:2,:])/dt
#print(X_bin.shape)
Xk = X_bin[:,1:]
Xk_1 = X_bin[:,:-1]
Ax_k1 = np.matmul(A_s, np.matrix(Xk_1, dtype = float))

term_1 = (Xk - Ax_k1)
term_2 = np.transpose(term_1)
Ws = np.multiply(1/(K-1), np.matmul(term_1, term_2))


W = np.zeros((5,5))
W[2:4, 2:4] = Ws
print(f'W is {W}')
print(f'Ws is {Ws}')

#========================================================#
# END YOUR CODE
#========================================================#
```

```
W is [[ 0.          0.          0.          0.          0.        ]
 [ 0.          0.          0.          0.          0.        ]
 [ 0.          0.          0.01151216 -0.00079399  0.        ]
 [ 0.          0.         -0.00079399  0.01212474  0.        ]
 [ 0.          0.          0.          0.          0.        ]]
Ws is [[0.011512161633165508 -0.0007939896703401133]
 [-0.0007939896703401133 0.012124738898787166]]
```

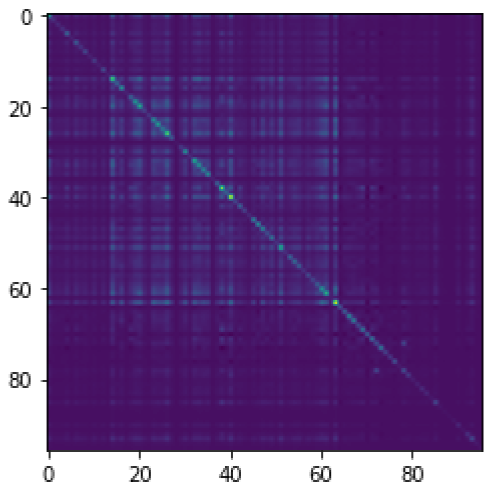
Answer: W (see above printed) is 5x5 matrix with all zeroes except Ws

(e) (4 points) Fit the Q matrix.

Find the $\mathbf{Q}$ matrix using the $\mathbf{C}$ matrix calculated in part (c). Submit a plot of Q using `plt.imshow(Q)`.

In [5]:
```
## Part e
#========================================================#
# YOUR CODE HERE:
#   Fit the Q matrix, and visuailze it using plt.imshow(Q)
#========================================================#
dt = 25
X = scipy.sparse.hstack(R[0:400]['cursorPos'])
X = scipy.sparse.csc_matrix(X)
X_first_bin = nsp.bin(X[0:2],dt,'first')
X_bin = np.diff(X_first_bin/dt)
X_first_bin = np.vstack((X_first_bin[:, 0:-1], X_bin, np.ones(np.size(X_bin,1))))
Y = scipy.sparse.hstack(R[0:400]['spikeRaster'])
Y_bin = nsp.bin(Y,dt,'sum')


K = np.size(Y_bin, 1)
Y_k = np.matrix(Y_bin, dtype = float)
CX_k = np.matmul(C, np.matrix(X_first_bin, dtype = float))
term1 = (Y_k - CX_k)
term2 = np.transpose(term1)
Q = np.multiply(1/K, np.matmul(term1, term2))
plt.imshow(Q)
plt.show
print(Q)
#========================================================#
# END YOUR CODE
#========================================================#
```

```
[[1.13424397 0.09171865 0.08359115 ... 0.12934506 0.021602   0.01461902]
 [0.09171865 0.1960104  0.02184649 ... 0.03148632 0.00653313 0.00338361]
 [0.08359115 0.02184649 0.16839519 ... 0.03247488 0.00860682 0.0037054 ]
 ...
 [0.12934506 0.03148632 0.03247488 ... 0.37556598 0.01733965 0.00648853]
 [0.021602   0.00653313 0.00860682 ... 0.01733965 0.06649927 0.00222185]
 [0.01461902 0.00338361 0.0037054  ... 0.00648853 0.00222185 0.03853453]]
```



Answer:See above Q matrix and plot

(f) (9 points) Write the KF recursion.

Write a function, `KalmanSteadyState.m` that accepts as input the $\mathbf{A}, \mathbf{W}, \mathbf{C}$, and $\mathbf{Q}$ matrices and returns Σ_∞, $\mathbf{K}_\infty$, $\mathbf{M}_1$ and $\mathbf{M}_2$. We are going to use an assumption made in Gilja, *Nuyujukian* et al., Nature Neuroscience 2012, which is that the monkey sees the cursor whenever it is updated and therefore has no uncertainty in its position. Thus, in your recursion, make the following modification in the recursion: If `S` denotes $\Sigma_{k|1:k-1}$, then immediately after calculating `S`, set:

$$\mathbf{S}[\mathbf{0}:\mathbf{2},:] = \mathbf{0}$$
$$\mathbf{S}[:,\mathbf{0}:\mathbf{2}] = \mathbf{0}$$

This removes all uncertainty in the cursor's position. Use a while loop with the following convergence criterion: `np.max(np.abs(it_d)) > tol` where `it_d` measures the difference between M1 and M2's entries' between current value and updated value after iteration. and $\mathbf{tol} = \mathbf{10^{-13}}$. Submit the values of the $\mathbf{M1}$ matrix and the value of `np.sum(M2,1)`.

In [6]:
```
def KalmanSteadyState(A, W, C, Q):

    S_p = np.zeros((5,5)) # previous state estimate cov
    S_c = np.zeros((5,5)) # current state estimate cov
    M1_p = np.zeros((5,5)) # previous M1
    M1_c = np.ones((5,5)) # current M1
    M2_p = np.zeros((5,96)) # previous M2
    M2_c = np.ones((5,96)) # current M2
    tol = math.pow(10, -13)
    count = 0

    # Stopping criterion.
    it_d = np.hstack((M1_c.flatten() - M1_p.flatten(),  M2_c.flatten()-M2_p.flatten()

    while(np.max(np.abs(it_d)) > tol):

        #========================================================#
        # YOUR CODE HERE:
        #   Implement the Kalman filter recursion.
        #========================================================#

        S_c = np.matmul(np.matmul(A, S_p),np.transpose(A)) + W
        S_c[0:2,:] = 0
        S_c[:,0:2] = 0

        t1 = np.matmul(S_c, np.transpose(C))
        t2 = np.matmul(C, S_c)
        t3 = np.linalg.inv(np.matmul(t2, np.transpose(C)) + Q)
        t4 = np.matmul(t1, t3)

        M1_p = M1_c
        M2_p = M2_c

        S_c = S_c - np.matmul(t4, t2)
        K_st = np.matmul(t1, t3)
        S_p = S_c
        M1_c = A - np.matmul(np.matmul(K_st, C), A)
        M2_c = K_st


        #========================================================#
        # END YOUR CODE
        #========================================================#
```

```
            count = count + 1
            it_d = np.hstack((M1_c.flatten() - M1_p.flatten(),  M2_c.flatten()-M2_p.flatte

    S_st = S_c # steady state covariance
    K_st = M2_c # steady state Kalman gain

    return (M1_c,M2_c,S_st,K_st)
```

After writing this function, run the following code to calculate the steady state parameters.

In [7]:
```
m1,m2,_,_ = KalmanSteadyState(A, W, C, Q)

print(m1)
print(np.sum(m2,1))
```

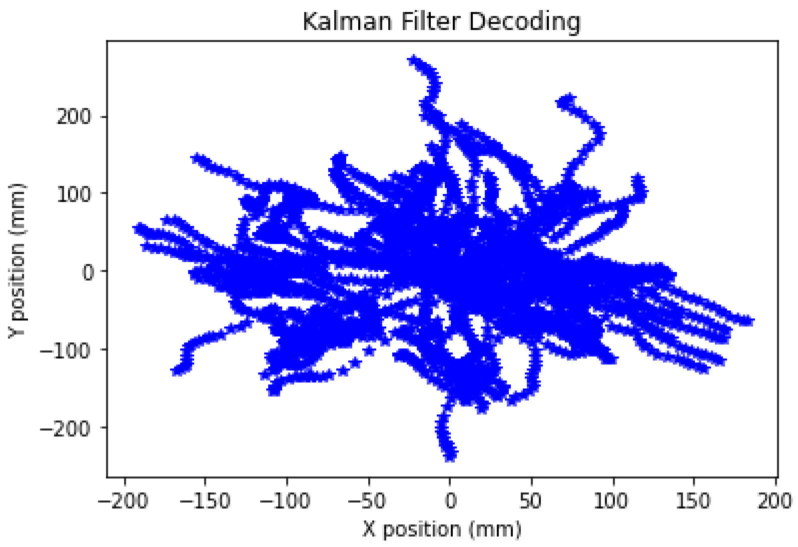
```
[[ 1.00000000e+00  0.00000000e+00  2.50000000e+01  0.00000000e+00
   0.00000000e+00]
 [ 0.00000000e+00  1.00000000e+00  0.00000000e+00  2.50000000e+01
   0.00000000e+00]
 [ 0.00000000e+00  0.00000000e+00  6.77629013e-01 -2.53787898e-02
   1.23314734e-02]
 [ 0.00000000e+00  0.00000000e+00 -1.54426706e-02  6.27592684e-01
  -5.85754563e-02]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
   1.00000000e+00]]
[[ 0.        ]
 [ 0.        ]
 [-0.05834287]
 [ 0.12670082]
 [ 0.        ]]
```

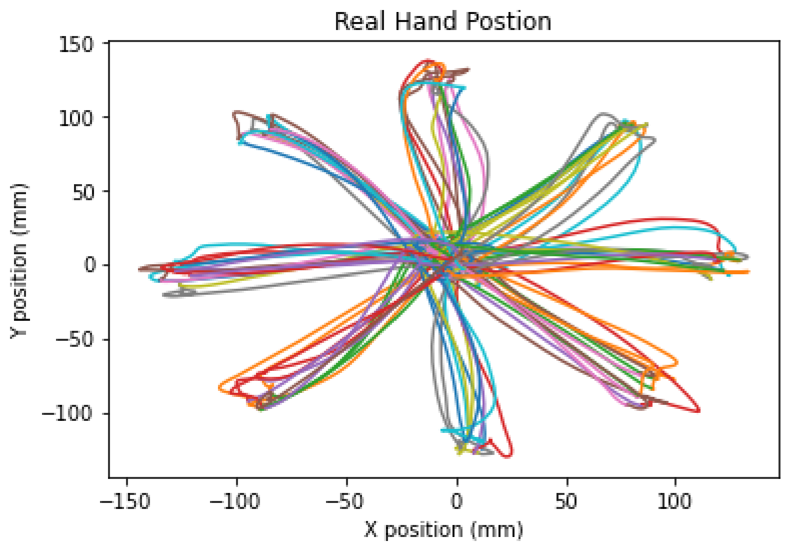
Answer:

(g) (6 points) Decode using the KF.

Using the $\mathbf{M}_1$ and $\mathbf{M}_2$ matrix found in part (f), decode the neural activity for each trial in the testing data. Initialize $\mathbf{x}_0$ on each trial to be the starting position on the trial, and a velocity of $\mathbf{0}$. On one plot, show the true hand positions. On a separate plot, show the positions decoded by the Kalman filter.

In [8]:
```
#====================================================#
# YOUR CODE HERE:
#   Decode the activity using a Kalman filter and
#   plot the decoded positions.
#====================================================#

init_vel = 0
for trial in np.arange(400, 506):
    hand_pos = R[trial]['cursorPos']
    Xk_p = np.matrix([[hand_pos[0, 0]], [hand_pos[1,0]], [0], [0], [1]], dtype = float


    Y = R[trial]['spikeRaster']
    Y_bin = nsp.bin(Y, dt,'sum')
    for time in range(len(Y_bin[0])):

        Y = np.matrix(Y_bin[:,time], dtype = float)

        Y = np.swapaxes(Y, 0, 1)


        Xk = np.matmul(m1, Xk_p) + np.matmul(np.matrix(m2,dtype= float), np.matrix(Y,d

        plt.plot(Xk[0], Xk[1], '*', color = 'blue')
        Xk_p = Xk


plt.xlabel("X position (mm)")
plt.ylabel('Y position (mm)')
plt.title("Kalman Filter Decoding")
plt.show()

for trial in np.arange(400, 506):
    hand_pos = R[trial]['cursorPos'][0:2, :]
    plt.plot(hand_pos[0,:], hand_pos[1,:])
plt.xlabel("X position (mm)")
plt.ylabel('Y position (mm)')
plt.title("Real Hand Postion")
plt.show()
#======================================================#
# END YOUR CODE
#======================================================#
```




Real Hand Postion
Y position (mm)
X position (mm)
150
100
50
0
−50
−100
−150
−100
−50
0
50
100

Answer: