Problem Set 3: Exploring Behavior and Neuronal Activity

Problem Set 3 will give you the opportunity to write Python code to look at that includes a combination of behavioral measures as well as physiological recordings from single neurons. Our goal will be to learn how to both organize datasets so that information can be efficiently extracted and to add new information to these datasets.

To complete this problem set, you will

- Load the data and look at some of the basic behavioral measures related to task performance, which in this set is based on manual button presses;
- Explore changes in eye position for each trial, which is more complicated than button presses because these change continuously throughout the entire trial;
- Characterize single neuron activity recording from a part of the brain sensitive to visual images and see how this activity is affected by stimulus and by eye movements.

At the end of this problem set, you will

- Have been introduced to Pandas, a powerful framework for managing data (http://pandas.pydata.org);
- Complete a programming assignment submission which will demonstrate that you know how to load, modify, and do basic analyses on real data;
- Complete a peer assessment submission to confirm that you can properly format graphs to show the results of your analyses. (To receive credit for the peer assessment, you must complete your peer evaluations by the evaluation due date.)

To help you accomplish this task, we will provide you with the following

- A function to load existing data into a DataFrame
- A function to show you how to plot part of the eye signal for a single trial
- Reminders about how you can compute spike rates so that data from different temporal epochs can be analyzed

Combined behavioral and neurophysiological studies of the visual system

One of the most exciting developments in the field of visual neuroscience is the ability of investigators to not only record the activity of individual neurons within specific visual areas of the brain, but to also be able to do so during active, real world behavior such as looking around a scene for familiar objects. The dataset for this problem set come from such an experiment. Recordings were done in a part of the brain called the *inferior temporal cortex*, which is a part of the visual system involved in object recognition. Neurons in this area can have quite complicated response properties. Unlike neurons in the early stages of the visual system, which are activated by simple visual features, cells in this area are often only made more active when presented with complex objects such as faces or pictures of real world objects. In the task, the subject had been trained to look for a target object among distractors (like playing a simplified version of *Where's Waldo*). When the target was found, a button was pressed that had been associated with the found object in in earlier training sessions. Eye movements were tracked on every trial so we can see where the subject was looking during the search. Using python, we will explore the dataset as it relates to recognition behavior, looking patterns, and neuronal activity.

Creating a "dataset" from real data

Data from real experiments generally come from many different sources. Neural activity is picked up by an amplification system that is almost always connected to a computer that stores the recorded signal. Another system may be dedicated to tracking the position of the eyes. Yet another system is in charge of presenting stimuli such as spots of light to reach for or pictures of real objects. And another system may be responsible for coordinating these and ensuring that the timing is accurate. When it's time to actually look at the data from a single session, it is certainly easiest if it is all together, in a single place. For this problem set, we will look at data that integrates basic timing information about when things like button presses and visual events occurred, where the eye movements were, and the spiking activity of a single neuron.

To get started, open up `problem_set3.py` in Spyder (or whatever Python environment you are choosing to use). The first thing you can try after 'running' this file (press the green arrow to load the file in Spyder) and then go to the console. From the console you can read the dataset from the disk by running the `load_data` function (defined in `problem_set3.py`):

```
In [1]: df = load_data('problem_set3_data.npy')
```

Note that we store the result in a variable, as the dataset is large and we need to refer to it by name, not just see it printed. This will create a pandas *DataFrame* which we learned a bit about in a previous video.

The DataFrame will have 10 columns after this step. Note that some columns are simple numbers and others are more complicated, because more than one value can occur per trial:

Name	Description	Datatype	Real Units	Size
targ	Name of target	String	Name	1 per trial
targ_x	Horizontal location of the target	float32 (real)	Degrees visual angle	1 per trial
targ_y	Vertical location of the target	float32 (real)	Degrees visual angle	1 per trial
em_time	Time for each eye movement sample	List of float32's	Timestamp in msec	1 list per trial (of variable length)
em_horiz	Horizontal position for each eye movement sample	List of float32's	Horizontal position in degrees visual angle	1 list per trial (of variable length)
em_vert	Vertical position for each eye movement sample	List of float32's	Vertical position in degrees visual angle	1 list per trial (of variable length)
stimon	What time the stimulus appeared on the screen	int32 (integer)	Timestamp in msec	1 per trial
response	What time the subject responded	int32 (integer)	Timestamp in msec	1 per trial
side	Which button (left or right) was the correct response?	int32 (integer)	0 means left and 1 means right	1 per trial
spk_times	Times of individual action potentials	List of float32's	Spike times (in msec)	1 list per trial (of variable length)

To better understand this chart it's necessary to understand how the experiment actually ran. On every trial of the experiment, a visual stimulus appeared at a variable time (**stimon**). (Note that each trial starts at its own "time zero".) The display contained a collection of small images displayed on a computer screen. On every trial, the collection included a target image (**targ**) for which the proper response was one of two buttons (**side**). The target on that trial appeared at the coordinates (**targ_x, targ_y**) which are in a unit called degrees visual angle. Think of this as the angle between looking straight ahead and where the target actually appeared. By convention, positive values of x are to the right, and positive values of y are up. The actual response (button press) occurred at a later time in the trial (**response**). Eye position was sampled every 5 msec (**em_time**) and the horizontal and vertical positions are separated into separate columns (**em_horiz**, **em_vert**). Finally, spikes were identified using a spike finder similar in nature to the one we used in Problem Set 1. The times for spikes on every trial are in a separate column (**spk_times**).

If you want to see the name of the columns in the dataset, type:

```
In [2]: df.columns

Out[2]: Index([u'targ', u'targ_x', u'targ_y', u'em_time', u'em_horiz',
u'em_vert', u'stimon', u'response', u'side', u'spk_times'],
dtype='object')
```

This list shows the data columns currently in our dataset. Each is named and has a 'u' in front, which is a special notation for strings that are in "unicode" format (which is not critical for this problem set).

If you want to see a just a portion of the dataset, try `df.head()` which will show (by default) the first five rows of the dataset for all columns. Beware that this dataset includes a fair amount of data, so this will still show a fair amount of data. If you want more info about basic pandas functions you can always ask for help, e.g., by typing `help(pd.DataFrame.head)`.

Exercise #1: Add a column for reaction times ('rts') and target eccentricity ('targ_ecc')

OK, now we will make our first change to the code for this problem set. We start with two additions, which demonstrate how you can add your own information into a dataset. For the first step, we want to add a new column, called 'rts' to our DataFrame by computing how long it took the subject to press the button after the stimulus appeared. This is called the *reaction time*, and it is an important measure of performance. Look into the code for a function called `add_info` and find the first comment section about adding a new column, which should be called `df['rts']`. When you have made the change to the `add_info` function, re-run the code and run the `add_info` command. You will know you are on the right track if `len(df['rts'])` returns `425`.

Now we want to add another column to the data describing how far away the target was from the center of the screen (where the subject's eyes started each trial). This is called "eccentricity", and it can be computed using the Pythagorean theorem using the data from the targ_x and targ_y columns. The simple formula to use is:

$$targ\_ecc = \sqrt{targ\_x^2 + targ\_y^2}$$

Hints: Functions you might want to use include `np.sqrt` (and possibly `np.round`). Convert the formula to valid Python code and add the `df['targ_ecc']` column. Reload your Python file and then run the `add_info(df)` command to include the new column in your data.

Exercise #2: Compute means for the reaction times sorted by the 'targ_ecc' variable

One of the most common requirements for data analysis in science is *aggregating* results of repeated measurements over particular conditions of interest. In many experimental sessions, similar trials are repeated and data are collected for each such trial multiple times. Then, at the

end of the session, an estimate of some measure (reaction time, neural response, etc.) is made for each sub-condition by applying a function (often *mean()*, but not always) for all of the trials belonging to each sub-condition.

Here we want to find the average reaction time for each of the different target eccentricities (distances) that were tested. You might first ask what were the tested eccentricities (perhaps you noticed this when you were adding them, above). You can find this out using the np.unique function directly:

```
In [68]: np.unique(df['targ_ecc'])
Out[68]: array([ 2.,  4.,  6.,  8.], dtype=float32)
```

We see that there are 4 unique values (2,4,6,8) corresponding to distances away (in degrees visual angle) from the center of the screen that the target could appear. What if we wanted to count the number of trials in each condition?

For this we can explicitly group the data and then apply the `count` function:

```
In [81]: df['targ_ecc'].value_counts()
Out[81]:
4    119
2    112
6    108
8     86
dtype: int64dtype: int64
```

This shows that there were about 100 trials for each, although there were a few more at the 4 degree location that at, for example, the 8 degree location. Using the same approach, can you see how many trials were run for each target type (`df['targ']`)?

Now let's return to our question for this exercise. What was the mean reaction time for each tested target eccentricity? Instead of looping over all the reaction times and keeping track of the conditions ourselves, we will take advantage of the DataFrame framework in pandas to make this very easy. In particular, we'll employ a very useful function called `pivot_table()`. You can see the official documentation for this function by typing `help(pd.pivot_table)`. At first it may seem complicated to specify the correct arguments, but if you understand what it does, the parameters should make more sense.

You call the pivot_table function as:

```
df.pivot_table(values='VALUE_COL', index='SORTING_COL')
```

Try this at the command line (substituting VALUE_COL and SORTING_COL with actual names of columns in our dataset!). When you have the proper syntax for computing mean rts by target eccentricity, copy this code in the function called `rts_by_targ_ecc()`. Note that in the problem set, the result of this function call is assigned to a variable that is returned. You should do the same.

Exercise #3: Plot behavioral variables that are sorted and grouped

Graphs are essential for presenting data, and in this exercise we'll build on the pivot table to create a graph of reaction times indexed by target eccentricity and grouped by the "side" variable, which specifies which button press was required to successfully complete the trial. If you are confused by the idea of both indexing and grouping, consider measuring the heights of males and females at 4 different ages. The age condition can serve as an index variable, and for each age, we have a "grouping" by gender (note that how you sort and group is often a matter of preference or a matter of highlighting important trends in data). Pandas makes it quite easy to manage sorting and grouping like this and to turn these data into nicely formatted graphs.

Before we actually create the graph, we want to add a new column into the DataFrame that converts values from the column called 'side' (which are 0's and 1's) into more meaningful values ('left' and 'right'). This is a very common requirement when managing data. In Python, one way to do this is using the NumPy function called `np.choose()`.

First, try the following:

```
np.choose([2,1,0,1,2],['zero','one','two','three'])
```

Do you see how this works? Think of the first argument to `np.choose` as the indices to use to 'choose' items from the list specified by the second argument. Note that indices can be repeated, and not all of the available ones have to be used. This is called doing a "table lookup", and it's a remarkably useful technique to understand how to use.

So to convert from the 0's and 1's in the `df['side']` column try the following code:

```
df['side_name'] = np.choose(CHOOSERS,LIST_TO_CHOOSE_FROM)
```

If you substitute `CHOOSERS` and `LIST_TO_CHOOSE_FROM` with the proper arguments, that one line can use the 0's and 1's in the side data column (the chooser) to choose from a Python list containing the strings 'left' and 'right' (the list to choose from). Where there is a 0 in the side data, it should choose 'left', and where there is a 1 it should choose 'right'. This is much more efficient than having to write a for-loop to do the conversion, and much, much faster. As a first step for this exercise, find the function called `plot_rts(df)` in the problem set and add this line.

Now we will create a bar chart. To do this we will create a pivot table as above, but we'll now include a grouping variable (the side_name variable) using the additional keyword `columns='GROUPING_COL'.` If you look at the results of this call to `pivot_table()` (try it at the command line) you will see that it results in a 2D collection of values. The rows should correspond to the target eccentricity, as before, but now we have columns for each of the sides (which should be labeled 'left' and 'right').

Instead of returning the result of the call to `pivot_table()` we will tell Python to plot it. For example, if we had:

```
results = df.pivot_table(...)  # (You will fill in the ...)
```

then you can plot the result by adding:

```
results.plot()
```

Note, however, that the default plot is a line plot, not a bar chart. To get a bar chart, add the parameter `kind='bar'` to the plot() function call: `results.plot(kind='bar')`. Add the plotting call to the `plot_rts(df)` function.

The plotting routines in pandas know how to plot grouped bar charts automatically, which is very helpful. The way this grouping occurs is controlled by the different selection of the "index" and "columns" choices. Here, we asked you to choose index to correspond to target eccentricity and columns to group the data into pairs based on side. You can see examples below of how this should look.

Because any graph you create should be labeled properly, complete this function by adding a proper title, xlabel, and ylabel. You can also adjust the legend to clean it up and make sure it doesn't obscure any data (check out the `title=` and `loc=` keywords). Each of the plotting adjustments can be made by calling the corresponding `plt` function, e.g.: `plt.title()` and `plt.ylabel()`.

Once you get this plot to look good, it should be saved as an image that can be uploaded as part of the peer assessment (see below).

Quiz Question: Look at your graph. Do you notice any consistent difference between the right and left responses? Usually subjects have a preferred hand and are quicker to react with that hand (i.e., lower RTs). Does this subject look to have a preference? Which side?

Exercise #4: Looking at eye movements

For this part of the problem set, we want to explore the eye movement patterns that occur on every trial. To get started with visualizing these, we will first complete a short function to extract the eye positions only during the time that the visual stimulus was on the screen. Before we do that, let's see how to plot the eye movement data for any trial. Try:

```
plot(df['em_time'][0], df['em_horiz'][0])
```

What does this show? On the x-axis we have time (in msec) and on the y-axis we have horizontal eye position (in degrees visual angle). You can, of course, do the same for the vertical eye position:

```
plot(df['em_time'][0], df['em_vert'][0])
```

These plots show the eye position for the whole of trial 0 – even times when there was no stimulus on the screen. To focus our analysis a bit more, our first step is to extract only the eye positions corresponding to times when the stimulus was on the screen. The function

`get_ems(df, trial)` is intended to get the actual eye positions for the period between "stimon" and "response". Edit that function to take the dataframe and a desired trial number (where 0 corresponds to the first trial) and have it return the times, the horizontal, and the vertical eye samples for just the period between "stimon" and "response". The key to this function is finding the proper indices for any given trial. Keep in mind that you need to account for the sampling rate of the eye position signal (how can you find this?).

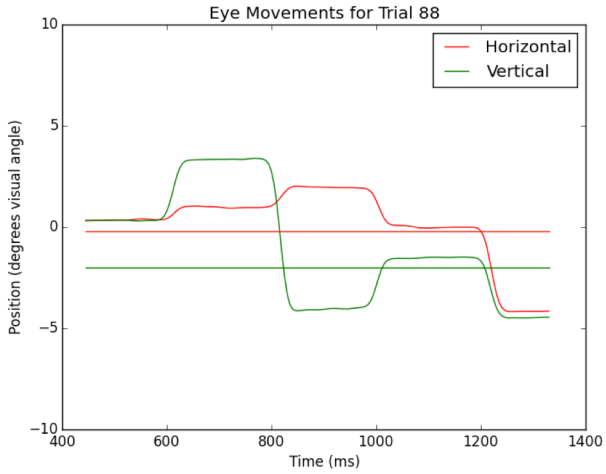
Quiz Question: What is the sampling rate for the eye movement signals? (Hint: Think about what is actually contained in **`df['em_time']`**)

With `get_ems(df, trial)` working, we want to now create an eye movement plot that shows the horizontal and vertical eye positions, in addition to information about where the target actually appeared on that trial. To do this, we will edit the function called `plot_ems_and_target(df, trial)`. This function will call `get_ems(df, trial)` to actually retrieve the eye position information, and we will pass this to the usual `plt.plot()` routine. Note that `plt.plot()` can take multiple series, so we can say, for example:

```
plt.plot(x0, y0, 'r', x1, y1, 'g')
```

and this will create a plot with two lines, the first colored red and the second colored green. For consistency, we want this plot to have the following characteristics:

- X axis is time, in msec, from start of trial
- Y axis is position, in degrees visual angle
- Trace of horizontal position against time (only during stimon period) in red
- Trace of vertical position against time (only during stimon period) in green
- Horizontal line across graph with y-value denoting target x position in red
- Horizontal line across graph with y-value denoting target y position in green
- Label x and y axes (with units)
- Title graph (include trial number in label)
- Include legend to show that red is for horizontal and green for vertical
- Set the y-axis to always be between -10 and 10



Example plot that shows all the elements described above for Exercise 4, which should be the plot generated by calling:

```
plot_ems_and_target()
```

for trial 88.

Once you get this plot to look good, call your plotting function for **Trial 112** and save as an image that can be uploaded as part of the peer assessment (see below).

Quiz Question: Use your **`plot_ems_and_target`** function to look at trial 213. Make a note of when (in ms within the trial) the eyes first get very close to the target (you don't have have an exact value)?

Exercise #5: Combining eye movements and neural data

So far we have learned how to organize and sort our data by conditions and groups and we have seen how eye position changes from trial to trial. In this part of the problem set, we will ask how neurons that respond to visual objects are affected by a target object's location and the position of the eyes.

To get started, let's revisit a problem we encountered in Problem Set 2. How do we turn a list of spike times into rates? Recall that in that problem set we selected a 'bin' or 'window' and counted spikes that occurred over a certain interval. We then turned that count into spikes/second, or rate. We will complete a function in this problem set that does something very similar here.

In your Python code, find the function `get_rate(spk_times, start, stop)`. We want this function to return the spike rates for a given collection of spike times, using start and stop as the counting window (the times for this problem all have the units msec). For this function, the spike times are just the spikes from single trial. The returned value should be in the units of spikes/sec. Add the necessary steps to this function to make the function work properly. Think about finding spike times between start and stop, counting these up, and turning that count into a rate (spikes/second). As `start` and `stop` are in msec, be careful with your units. Remember that normal rates range from a few spikes per second to no more that 100-200 spikes/sec. If your numbers are not in this range, then there is likely a unit problem.

As a simple test of your `get_rate()` you can try:

```
get_rate(np.arange(1000,step=10),100,200)
```

This creates a list of spike times between 0 and 1000 stepping by 10's and asks what is the rate (in spikes/sec) between time 100ms and 200ms. If the spike times occur every 10ms, what should the answer be? Regardless of the counting window, the answer should always return `100.0 spikes/sec`.

With a working `get_rate()` function, we want to actually add some rates for all the trials in the dataset. To do this, you can use the `add_aligned_spikes()` function, which has already been defined for you. What does this function do? It uses your `get_rate()` function to add the rate

for every trial, but it also helps you compute a reasonable start and stop value. In particular, it allows you to choose a window that is temporally related to a specific event that happened for that trial. Recall that one of the columns in our DataFrame is called **stimon**. This column contains the time (in ms) at which the visual image appeared on the screen. We expect changes in rate to occur sometime after this particular event. Given that the neuron begin recorded from in this dataset was in a part of the brain where signals arrive approximately 100ms after light reaches the eye, start with a counting window of 100ms-200ms *after* the stimulus came on. (Keep in mind that in our real experiment, the stimulus came on at different times for each trial.) To do this, type:

```
add_aligned_rates(df, 'stimon' ,100, 200)
```

If everything worked correctly, then nothing should be returned, but a new column in the DataFrame df should be added called `df['rates_stimon_100_200'].`

So now we can actually look at how neural activity, as measure in our rate variable, depended on things like which target was shown and where the target actually appeared. This problem is just like our reaction time problem above, except that instead of use the reaction times (rts) as our values, we will use the rates. Recall our use of the pivot table command:

```
df.pivot_table(values='VALUE_COL', index='SORT', columns='GROUP')
```

We can do use this same format to analyze spike rates. Try:

```
df.pivot_table(values='rates_stimon_100_200', index='targ',
               columns='targ_ecc')
```

Now we have a table showing how this neuron responded to each different target depending on how far away that target was from the center of the screen.

Quiz Questions: Which target was the most effective (caused the highest spike rate)?

OK, we are almost done! The last thing we want to do is combine some of the eye movement data with our neural analysis. In Exercise #4, above, we created a plot of the eye position for a single trial along with markers showing the target location.

From the command line, call that function for trial 303:

```
plot_ems_and_target(df, 303)
```

What do you see? The target seems to have appeared 4 degrees straight down (0, -4.0) starting around 520ms after the trial began. Notice the patterns of the eye position traces. They seem to take four stable positions throughout the trial, and around 1100ms, the eye position seems to get very close to the target position. This is a normal search pattern, where the eyes look around and after two searching eye movements, the subject finally finds ('acquires') the target on the third

one. Here, we would like to use the time that the subject finds the target as different alignment event, to compare with our analysis using the time of stimulus onset.

Identifying the time that the eyes find the target is a bit tricky, so we have supplied a function you can use to do this automatically. (The code is included at the bottom of `problem_set3.py`, though, so feel free to look it over or try to come up with a version yourself! Just make sure that you don't break the version we supply, or your submission will not work as expected.) At the end of the `add_info()` function template, you should see a line that reads:

```
# Leave this here - it adds information used for Exercise 5
add_acq_time(df)
```

When you called `add_info(df)` before, this information was added to your DataFrame as a column called `'targ_acq'`. So let's now compute spike rates for the 100ms-200ms interval after the target is acquired by an eye movement. To do this, we just need to call the same `add_aligned_rates()` function we used above, changing the alignment event:

```
add_aligned_rates(df, 'targ_acq' ,100, 200)
```

And now we've added those rates to our DataFrame. To see the results, re-run the pivot table command, but where you had `values='rates_stimon_100_200'` before (aligning to the moment the stimulus appeared) we now want to use `values='rates_targ_acq_100_200'` (aligning to the moment the eyes land on the target):

```
df.pivot_table(values='rates_targ_acq_100_200', index='targ',
      columns='targ_ecc')
```

Based on what we learned in Exercise #3, how do you think you could easily make a plot of these data? Give it a try!

Assignment problems:

You will submit results of your edits to `problem_set3.py` using the `problem_set3_submit.py` code (see **Submitting your code** below) and upload the results of your plotting routines using Peer Assignment tab in Coursera (see **Submitting your figures for Peer Assessment** below).

Submitting your code:

Once you have updated `problem_set3.py` as described above, you can submit this part of your problem set by opening the file `problem_set3_submit.py` in Spyder. With that file open, you can run it (press the green play button or hit F5) and it will prompt you for your email

address and a “one time password”. This is the same password you’ve used for the previous assignments (unless you requested that it be changed). This is NOT your Coursera password.

Submitting your figures for Peer Assessment

To complete the problem set, you will also need to complete the **Problem Set 3 Peer Assessment**. Please read the question prompts carefully to ensure that you submit the correct figure in the correct locations! You will need to save the figures (for inline figures, right click on the figure and select “Save As”, for figures in their own window, just choose the save icon). You can save the files as png, jpg, gif or pdf for peer assessment.

The purpose of the peer assessment is to ensure that you are presenting your data in a meaningful way that can be easily interpreted by others. Therefore, a lot of the assessment is based on the format of your figure, including axes labels and the figure title. Each figure will be assessed in the following categories (2pts possible per category): *Please note, there are multiple ways to receive full credit – we provide guidelines and examples below:*

Exercise 3 - Bar Graph Rubric:

The x-axis is properly labeled. The x-axis label should be “Target Eccentricity” or something similar. Units should be “degrees visual angle” or something similar.

0: The x-axis is unlabeled; or the labeling and units are both incorrect.

1: The x-axis is labeled but either the label name or the units are incorrect or not provided.

2: The x-axis is labeled properly (e.g. "Target Eccentricity") and units are correctly provided (e.g. “degrees visual angle”).

The y-axis is properly labeled. The y-axis should be labeled as "Reaction Time" or something similar. The units should be in either milliseconds (msec or ms are also acceptable) or seconds (also sec or s), and the units given should match the numeric labels provided.

0: The y-axis is unlabeled, or the labeling and units are both incorrect.

1: The y-axis is labeled but either the label name or the units are incorrect or not provided.

2: The y-axis is labeled properly (e.g. "Reaction Time") and units are correctly provided (e.g. msec).

The figure is properly titled. There are lots of options for the figure title - it just needs to make sense. For example, “Reaction Times by Eccentricity and Side”

0: The figure is untitled, or the title provided is irrelevant.

1: The figure has a descriptive title.

The legend is clearly labeled and does not obscure the plot. The legend should indicate that the two bars represent “left” and “right” button presses.

0: There is no legend, or the labels are irrelevant.

1: There is a properly labeled legend, but it significantly obscures the plot.

2: The figure has a clearly labeled legend that does not obscure the plot.

The figure consists of four groups of two bars. There should be one pair of bars for each eccentricity (2,4,6 and 8).

0: The figure is not a bar graph.

1: The figure is a bar graph, but the bars are not grouped properly to show left and right reaction times for each eccentricity.

2: The figure clearly displays pairs of bars for each eccentricity (2, 4, 6, 8) and the bars represent the reaction times for left and right button presses.

9 Total Points.

Here are some examples! (NOTE: These example figures have made up data so that we do not give away the solution!)

This figure would receive all 9 points. Everything looks good!

This figure would receive 5/9 points.

x-axis:	0 pts.
y-axis:	1 pt.
title:	1 pt.
legend:	1 pt.
bars:	2 pts.
Total:	5 points

Exercise 4 – Eye Position Rubric:

The x-axis is properly labeled. The x-axis label should be “Time.” Units should be “msec.”

0: The x-axis is unlabeled, tor the labeling and units are both incorrect.

1: The x-axis is labeled but either the label name or the units are incorrect or not provided.

2: The x-axis is labeled properly (e.g. "Time”) and units are correctly provided (e.g. msec).

The y-axis is properly labeled. The y-axis should be labeled as "Position.” The units should be in “degrees visual angle.” The y-axis goes from -10 to 10.

0: The y-axis is unlabeled, or the labeling and units are both incorrect.

1: The y-axis is labeled but either the label name or the units are incorrect or not provided.

2: The y-axis is labeled properly (e.g. "Position") and units are correctly provided (e.g. degrees visual angle), but the y-axis does not go from -10 to 10.

3: The y-axis is labeled properly (e.g. "Position") and units are correctly provided (e.g. degrees visual angle). The y-axis goes from -10 to 10.

The figure is properly titled. There are lots of options for the figure title - it just needs to make sense and should include the trial number (112).

0: The figure is untitled, or the title provided is irrelevant.

1: The figure has a descriptive title, but does not include the trial number.

2: The figure has a descriptive title including the trial number (112).

The legend is clearly labeled and does not obscure the plot. The legend should indicate that the two line colors represent “horizontal” (red) and “vertical” (green) eye movements.

0: There is no legend, or the labels are irrelevant.

1: There is a properly labeled legend, but it significantly obscures the plot.

2: The figure has a clearly labeled legend that does not obscure the plot.

The figure consists of four clear lines including the trace of the horizontal position against time (red), the vertical position against time (green) and horizontal lines representing the target x- (red) and y- (green) positions.

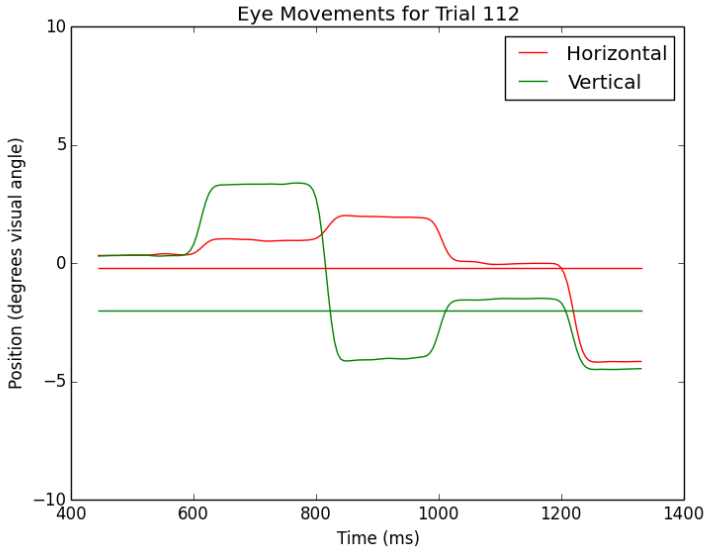
0: All of the lines are missing or are not clearly presented.

1: Only two of the four lines are clearly presented.

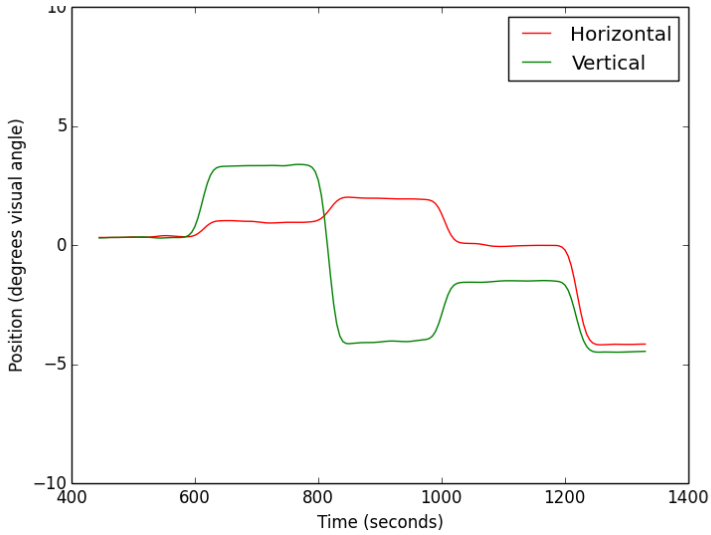
2: All four lines are clearly presented.

Total: 11 pts.

Here are some examples! (NOTE: These example figures use the wrong data so that we do not give away the solution!)



This figure would receive all 11 points.



This figure would receive 8/11 points.

x-axis:	1 pt.
y-axis:	3 pts.
title:	1 pt.
legend:	2 pts.
lines:	1 pt.
Total:	8 points.