MATLAB for Mehtalabbers: Week 3

by: David Groppe

For this chapter we'll go over how to make several different types of figures in MATLAB and some techniques for polishing up figures to make them publication quality. For more detailed information on MATLAB graphics check out Holland & Marchand's book *Graphics and GUIs with MATLAB*.

Ione Fine & Geoff Boyton's MATLAB tutorial has several chapters covering graphics and plotting and should be a good introductory resource:
http://faculty.washington.edu/ionefine/MatlabCourseNotes07.html

Topic 1 – A Scatter Plot with Regression Line

1.A Making a scatter plot:

A simple, useful type of graph to start with is a "scatter plot." Let's make a scatter plot of the reaction time data from the oddball tasks we used last week. The data are in the file *odbl_summ.txt*, which should be downloadable from the lab wiki.

First, load the data into MATLAB:

```
>> [easy_rt easy_acc hard_rt hard_acc] ...
=textread('odbl_summ.txt','%f %f %f %f','headerlines',1);
```

To create a new figure in MATLAB we typically need to open a new figure window:

```
>> figure
```

To turn that figure window into a two-dimensional plot, we use the *plot* function:

```
>> plot(easy_rt,hard_rt,'.');
```

The first and second elements of the *plot* function specify the x-axis and y-axis values of the points. The third argument specifies what symbol is used to represent the points (in this case a dot), what type of line (e.g., solid, dashed) should connect the points (if any), and what color the points and lines should be (default=blue). For example, see what changes in our figure when you use the following commands:

```
>> plot(easy_rt,hard_rt,'*');
>> plot(easy_rt,hard_rt,'r*');
>> plot(easy_rt,hard_rt,'r.--');
```

For a list of possible data point symbols, line styles, and colors, see the help documentation for *plot*.

Continuing with our example, it would help greatly to label what means what in our scatter plot. Let's go back to our original plot:

```
>> plot(easy_rt,hard_rt,'.');
```

and label the x-axis:

```
>> xlabel('Easier Oddball Task Reaction Time (in ms)');
```

The x-axis should now be labeled, but the fontsize is a bit too small. We can change that by having xlabel return a "handle" to the x-axis label, which allows us to access the properties of that text.

```
>> h=xlabel('Easier Oddball Task Reaction Time (in ms)');
>> set(h,'fontsize',14);
```

To see a list of all the properties of the text indexed by *h*, enter:

```
>> set(h)
```

You should get the following list:

```
     BackgroundColor
     Color
     DisplayName
     EdgeColor
     Editing: [ on | off ]
     FontAngle: [ {normal} | italic | oblique ]
     FontName
     FontSize
     FontUnits: [ inches | centimeters | normalized |
{points} | pixels ]
     FontWeight: [ light | {normal} | demi | bold ]
     HorizontalAlignment: [ {left} | center | right ]
     LineStyle: [ {-} | -- | : | -. | none ]
     LineWidth
     Margin
     Position
     Rotation
     String
     Units: [ inches | centimeters | normalized | points |
pixels | characters | {data} ]
     Interpreter: [ latex | {tex} | none ]
     VerticalAlignment: [ top | cap | {middle} | baseline |
bottom ]

     ButtonDownFcn: string -or- function handle -or- cell
array
     Children
     Clipping: [ {on} | off ]
     CreateFcn: string -or- function handle -or- cell array
     DeleteFcn: string -or- function handle -or- cell array
     BusyAction: [ {queue} | cancel ]
     HandleVisibility: [ {on} | callback | off ]
     HitTest: [ {on} | off ]
     Interruptible: [ {on} | off ]
```

```
Parent
Selected: [ on | off ]
SelectionHighlight: [ {on} | off ]
Tag
UIContextMenu
UserData
Visible: [ {on} | off ]
```

Thus with the *set* function, we can modify a lot of different aspects of our x-label text (e.g., fontsize, font type, font color).

Let's add a y-axis label and a title:

```
>> h=ylabel('Harder Oddball Task Reaction Time (in ms)');
>> set(h,'fontsize',14);
>> h=title({'Scatter Plot of Participant','Reaction Times
in Experiment 1'});
>> set(h,'fontsize',14,'fontweight','bold');
```

Note how curly brackets and a comma were used to break up the title into two lines of text.

1.B Adding a regression line:

Now to add a regression line. First we need to perform ordinary least squares regression on the line to figure out what the slope and intercept of the line should be. Our y-variable is reaction time in the harder task:

```
>> Y=hard_rt;
```

Our x-variable is a matrix composed of reaction time in the easier task and a column of ones. The column of ones represents the constant y-intercept term of a linear regression equation:

```
>> X=[easy_rt ones(length(easy_rt),1)];
```

We then use the function *regress* to get our coefficients:

```
>> b=regress(Y,X)

b =

    1.0587
   56.8875
```

This gives the equation for our regression line: y=1.0587*x+56.8875. To plot this we first have to tell the figure not to clear the plot each time we add something new to it:

```
>> hold on
```

To define our line, we just the need the beginning and end point. The x-coordinates of the beginning and end points are:

```
>> xpts=[350 550];
```

We use the regression line coefficients to get the y-coordinates of our beginning and end points:

```
>> ypts=b(1)*xpts+b(2);
```

Now to plot the line:

```
>> plot(xpts,ypts,'k-');
```

Ta-da! Wait a minute, what if you think that line is too skinny? Well, just like text in the figure have "handles" that can be used to modify their properties, so do graphical objects like lines. We can get that handle by storing the output of the *plot* command in a variable:

```
>> hln=plot(xpts,ypts,'k-');
>> set(hln,'linewidth',2);
```

Indeed, the axis in the figure even has a handle that you can access to do things like add grid lines. For example:

```
>> set(gca,'xgrid','on','ygrid','on');
```

The *gca* stands for "get current axis" and returns the handle of the axis you're currently working on. Likewise, *gco* can be used to get the handle of the "current object" (e.g., the line you just drew) and *gcf* can be used to get the handle of the "current figure" (in this case Figure Window 1, since you only have one figure open).

1.C Adding finishing touches:

Another nice item to add to this figure would be the correlation coefficient and its statistical significance. Recall from last week how to compute the correlation between variables:

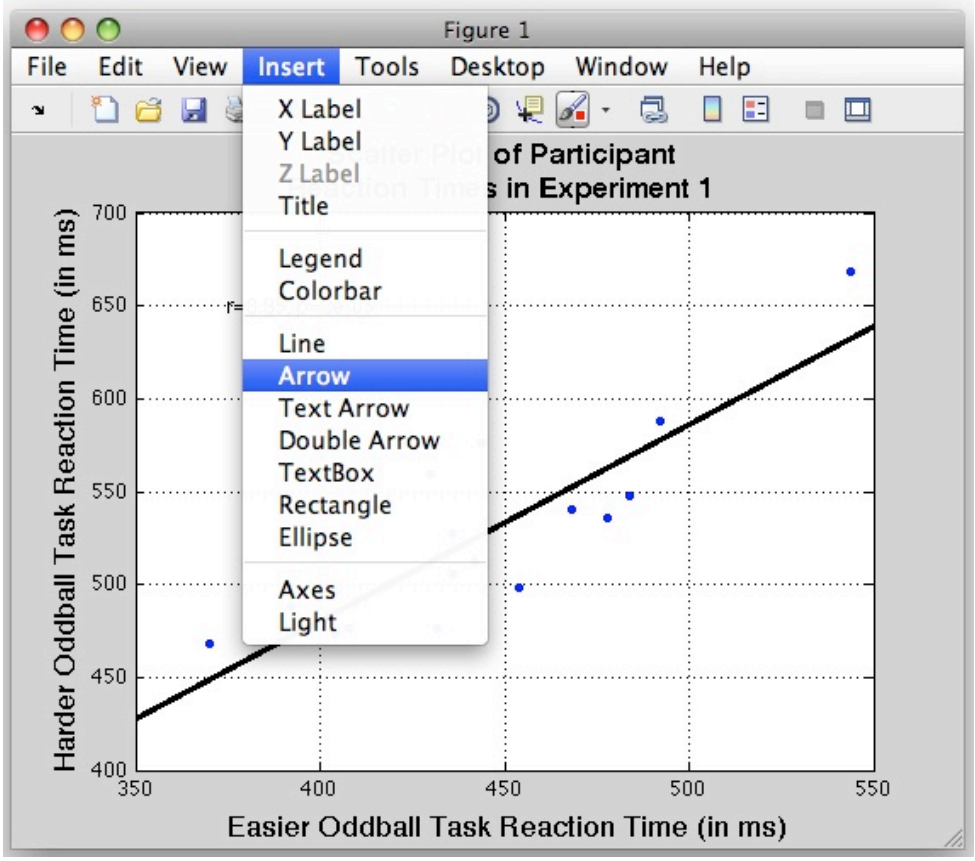
```
>> [r, p]=corr(easy_rt,hard_rt);
```

We can add these statistics to the plot with the *text* function:

```
>> text(375,650,['r=' num2str(r,2) ', p=' num2str(p,2)]);
```

The first two arguments provide the x and y-coordinates for the text. The third argument is the string of text that is added to the figure. We used the *num2str* command we've used before to convert the numeric variables to strings. The second argument of *num2str*, a "2", tells *num2str* to round to the second significant digit.

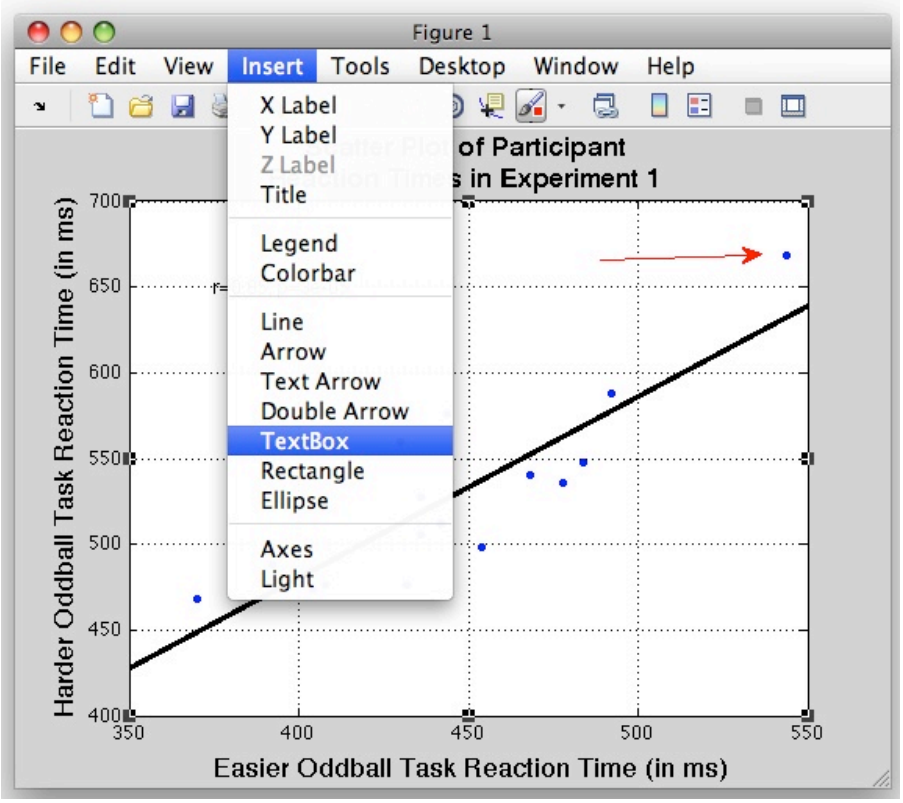
In addition to the command line, MATLAB figure windows provide a menu of options for modifying the figure. For example, say we'd like to add an arrow to the slowest participant's data point to indicate his abnormal age (for an undergraduate), which would explain his relatively slower speed). We can do this via the figure's *Insert* menu:



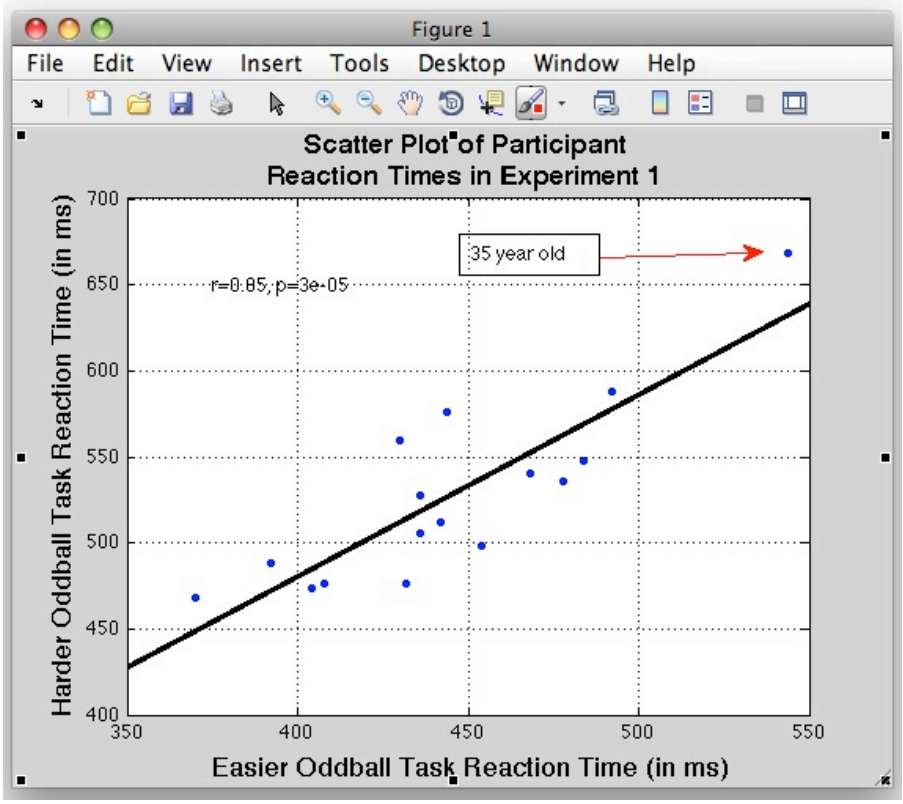
We can set the endpoints of the arrow with the computer's mouse and then change the color to red like so:

```
>> set(gco,'color','r');
```

Finally, we can add text behind the arrow specifying the participant's age also with the *Insert* menu:



Which gives us a final figure that looks something like this:



1.D Exporting/printing/saving a figure:

You can easily save a figure as an enhanced postscript (*eps*) or *jpg* file via the command *print*. For the example the following saves the figure as *scatter_demo.eps* in the current working directory:

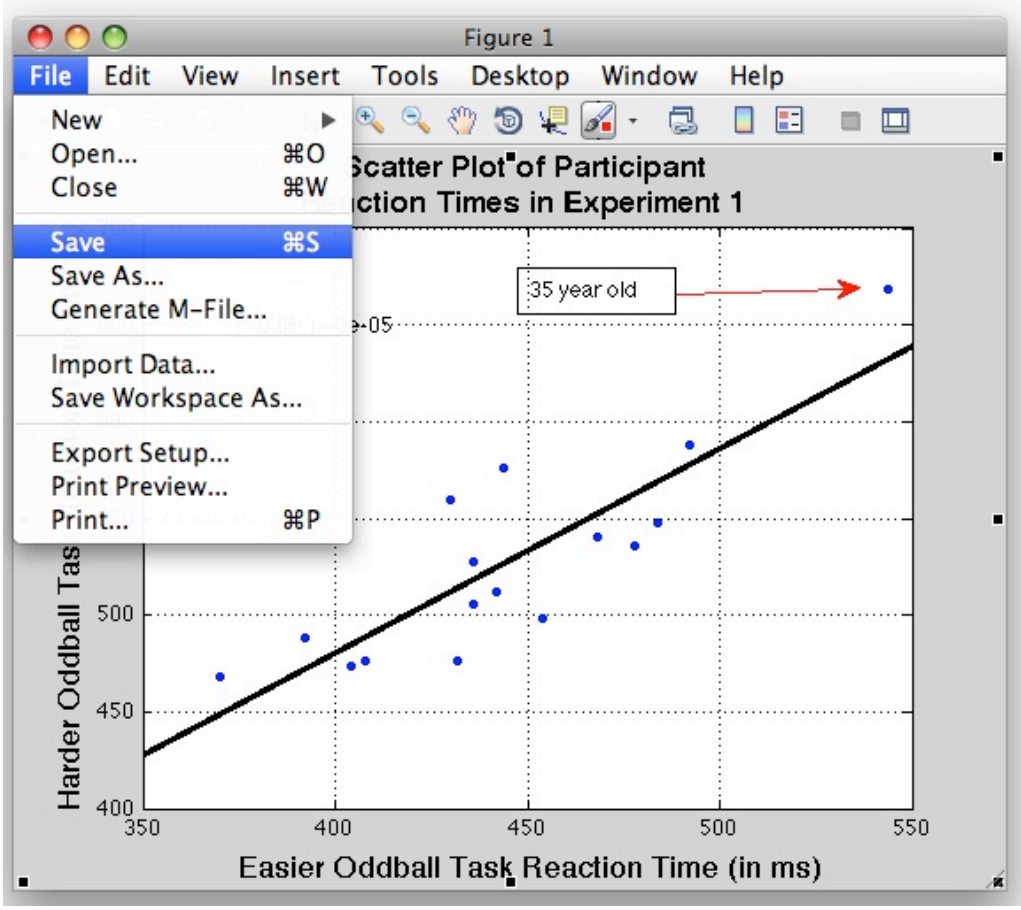
```
>> print -f1 -depsc scatter_demo
```

"-f1" specifies Figure 1 as the figure you want to save. "-depsc" specifies the format of the file (color enhanced postscript). The final string is the filename (minus its extension).

eps files are great in that they can be easily modified with programs like Illustrator and can be easily converted to *pdf*s with the Unix *ps2pdf* command. However, some figures look better as *jpg*s. To save the file as *scatter_demo.jpg* enter:

```
>> print -f1 -djpeg scatter_demo
```

Finally you can save the figure as MATLAB *fig* file via the *File* menu in the MATLAB figure window.



This way you can re-open the figure with MATLAB at a later date and make changes.

EXERCISE

1.1) Produce a scatter plot with a regression line of the accuracy data in the easier and harder oddball tasks.

Topic 2 – Other Types of Graphs

2.A Boxplots

A handy way to summarize data is a "boxplot" (http://en.wikipedia.org/wiki/Box_plot). We can easily make a boxplot of our reaction time data via the *boxplot* function. First, let's clear our scatter plot figure:

```
>> figure(1);clf
```

"figure(1)" makes Figure 1 our current working figure. "clf" clears the current working figure. Now we'll create a boxplot:

```
>>boxplot([easy_rt hard_rt]);
```

The first and second box summarize the easy and hard RT data respectively. The red line indicates the median of the data. The upper and lower edge of each box indicate the 75th and 25th percentiles, respectively. The "whiskers" (dashed lines) indicate the maximum and minimum observations that are not considered "outliers." By convention, an outlier is any point that lies 1.5*(the interquartile range) above the 75th percentile or 1.5*(the interquartile range) below the 25th percentile. Outlier observations are represented by red plus signs.

Lets clarify this plot a bit with some labels:

```
>> h=ylabel('Reaction Time (in ms)');
>> set(h,'fontsize',12);
>> set(gca,'xtick',[1 2],'xticklabel',{'Easier','Harder'});
>> set(gca,'xticklabel',{'Easier','Harder'});
>> h=xlabel({'','Oddball Task'});
>> set(h,'fontsize',12);
```

Note how we changed the labels of the tick marks on the x-axis with the *set* function to indicate what each box corresponded to. Also note how we added a blank filler line the x-axis label to distance it from the x-axis tick mark labels.

2.B Bar graphs with error bars

Another way we could summarize the reaction time data is with a bar graph. We can do this with the *bar* function:

```
>> figure; bar([1 2],mean([easy_rt hard_rt]));
>> h=ylabel('Reaction Time (in ms)');
>> set(h,'fontsize',12);
>> h=xlabel('Oddball Task');
>> set(h,'fontsize',12);
>> set(gca,'xtick',[1 2],'xticklabel',{'Easier','Harder'});
>> h=title('Mean Reaction Times (Error bars are standard
deviation)');
>> set(h,'fontsize',14);
```

Now lets add error bars to indicate the standard deviation with the function *errorbar*. We can use the function *std* to compute the standard deviation for both sets of observations:

```
>> hold on
>> h=errorbar([1 2],mean([easy_rt hard_rt]),std([easy_rt
```

```
hard_rt]),'k.');
>> set(h,'linewidth',2)
```

We thickened the error bars up a bit with that last line. Note, the mean and standard deviation are not the most appropriate summary statistics for reaction time since reaction time distributions are typically positively skewed. Mean and standard deviation are used here for simplicity.

2.C Histograms

When you have a big enough sample size, histograms are a useful way to visualize the data's distribution. Our sixteen participant oddball data set is too small to construct a decent histogram. But we can simulate some data by randomly sampling from a Gaussian distribution via the *randn* function:

```
>> sim_data=randn(1,200);
```

Now to make the histogram:

```
>> figure; hist(sim_data,20);
```

The last argument of the *hist* function call specifies how many bins to use when making the histogram. Try making histograms with different numbers of bins to see what happens.

2.D Imagesc

The function *imagesc* makes 3D plots by using color as a third dimension. It's particularly useful for visualizing matrices. For example, lets concatenate our reaction time data into a matrix.

```
>> both_rt=[easy_rt hard_rt];
```

Now to visualize that:

```
>> figure;imagesc(both_rt);colorbar;
>> set(gca,'xtick',[1 2],'xticklabel',{'Easier','Harder'});
>> ylabel('Participant');
```

Note, the *colorbar* function adds the colorbar on the right hand side of the figure, which shows you how color maps to reaction time.

EXERCISE

2.1) Reproduce the plots above using the accuracy data instead of reaction time.

2.E Plotting multiple lines on the same axis

Sometimes to contrast multiple sets of data, you might want to plot multiple lines on the same axis. For example, you might want to compare two ERPs. Let's illustrate with some grand average ERPs stored in the MATLAB file *prsentDEMO.gnds* which you can also download this file from the lab wiki.

Load the file into MATLAB:

```
>> load prsentDEMO.gnds -MAT
```

As we saw with EEGLAB *set* files last week, we need the *-MAT* to let MATLAB know *prsentDEMO.gnds* is a MATLAB file even though it doesn't have a *.mat* extension.

You should now have a struct variable *GND* in memory, which contains the grand average ERPs to tones in various conditions and associated information. The grand averages are stored in the 3D matrix *GND.grands*. The first dimension of *GND.grands* corresponds to different electrodes. The second dimension corresponds to the 256 different time points. The third dimension corresponds to four different bins. The 16th electrode is MiCe (you can verify this with GND.chanlocs(16)). To plot the ERP in Bin 1 at MiCe:

```
>> figure; plot(GND.time_pts,GND.grands(16,:,1));
```

Now let's add the ERP in Bin 2 at MiCe in red:

```
>> hold on; plot(GND.time_pts,GND.grands(16,:,2),'r');
```

The x-axis limits are a bit bigger than they need to be. Let's fix that:

```
>> axis([-100 920 -7 3]);
```

And let's make negative up:

```
>> set(gca,'ydir','reverse');
```

And we can add a legend to identify which ERP is which:

```
>> legend('Bin 1','Bin 2','location','northeast');
```

Obviously, this is still a pretty shabby ERP plot (e.g., it's missing axis labels and lines to indicate time=0 and voltage=0). We could spruce it up more for but for the time being it is enough to learn how to overlay multiple lines and add a legend.

EXERCISE

2.2) Contrast the ERPs at MiPa (electrode 17) in Bins 1, 2, & 3 on the same plot with a legend.

2.F Plotting ERP topographies

As a final illustration of MATLAB graphics, we'll use the EEGLAB function *topoplot* to plot the topography of the auditory N1 (first you'll need to download EEGLAB: http://sccn.ucsd.edu/eeglab/install.html). First let's the find the time point in our grand averages that corresponds to 100 ms. The field *GND.time_pts* stores the value of each time point in ms. According to the *find* function:

```
>> find(GND.time_pts==100)

ans =

    51
```

The 51st time point is 100 ms. Now to plot the topography at 100 ms of the ERPs in Bin 1:

```
>> figure; topoplot(GND.grands(:,51,1),GND.chanlocs);
>> h=title('Bin 1, 100 ms');
>> set(h,'fontsize',12,'fontweight','bold');
```

Topic 3 – Multiple Plots on the Same Figure

3.A Subplots

Sometimes it is useful to create multiple plots in the same figure. This can be done with the MATLAB function *subplot*. For example, lets plot the topographies of the ERPs in Bin 1 at four different points in time (100, 192, 300, and 472 ms). First, lets find out which time points correspond to those four times. We already know that 100 ms corresponds to the 51st time point. As for the rest:

```
>> find(GND.time_pts==192)
ans =

    74

>> find(GND.time_pts==300)
ans =

   101

>> find(GND.time_pts==472)
ans =

   144
```

Now let's create a figure and carve it up into four axes:

```
>> figure; subplot(2,2,1);
>> topoplot(GND.grands(:,51,1),GND.chanlocs);
>> h=title('Bin 1, 100 ms'); set(h,'fontsize',12);
```

The first argument of subplot tells it to parse the figure such that there are two rows of axes. The second argument of subplot tells it to parse the figure such that there are two columns of axes. The third argument says to make the first axis (the topmost and leftmost axis) the current working axis. The subsequent topoplot command draws the topography there.

Now for the rest of the topographies:

```
>> subplot(2,2,2);
topoplot(GND.grands(:,74,1),GND.chanlocs);
>> h=title('Bin 1, 192 ms'); set(h,'fontsize',12);
>> subplot(2,2,3);
topoplot(GND.grands(:,101,1),GND.chanlocs);
>> h=title('Bin 1, 300 ms'); set(h,'fontsize',12);
>> subplot(2,2,4);
topoplot(GND.grands(:,144,1),GND.chanlocs);
>> h=title('Bin 1, 472 ms'); set(h,'fontsize',12);
```

Note that we didn't have to use the *hold on* command. That's because each plot command operated on a different axis (i.e., we never plot more than one thing on the same axis).

We can add a title to the entire figure using *textsc*:

```
>> h=textsc('Grand Average Topographies','title');
```

```
set(h,'fontsize',12','fontweight','bold');
```

EXERCISE

3.1) Make a figure with two axes. On the left axis, make a scatter plot of reaction time in the two oddball tasks. On the right axis, make a scatter plot of accuracy in the two tasks. Be sure to include axis labels and titles. You do not need to include regression lines.

Summary

You should have learned the following in this unit:

- How to create figures with one or more axes
- How to save figures and to export them to postscript and jpeg formats using the *print* command
- How to create various types of plots in an axis
- How to overlay multiple plots on the same axis using the *hold* command
- How to manipulate the properties of axes, figures, and graphical objects using handles and the *set* function
- How to manipulate and annotate a figure/axis using the menus on a figure window

You should have also rudimentary understanding of the following functions:

- figure
- plot
- axis
- xlabel
- ylabel
- title
- set
- text
- print
- boxplot
- bar
- errorbar
- imagesc
- legend
- find
- topoplot
- subplot
- textsc
- regress
- randn