Module 1: Getting to Know Your Data: Visualization of High-Density Multi-Modal Interactions

With the eight scripts associated with this module you can begin creating visualizations of the most common types of behavioral data collected in psychology research, even with little or no programming experience. The scripts will allow you to make visualizations of various type of data, including event data from annotation of audio or video records, as well as timeseries data, whether from sensors or annotations.

Additionally, the first three scripts are designed to be run one line at a time to educate novice programmers, and the first two scripts contain practice problems with answers. Users who work through these scripts step-by-step will gain skills and confidence to combine and create variations of the provided scripts to accommodate their own data and research questions.

For novice users, we suggest working through the scripts in the order listed below. There are eight total scripts, each described below in detail.

Script 1: *programmingBasics.m*
Script 2: *simpleTimeseriesPlots.m*
Script 3: *simpleEventDataPlots.m*
Script 4: *multiParticipantEventPlotting.m*
Script 5: *annotationImport.m*
Script 6: *convertEvents2Timeseries.m*
Script 7: *plotTimeseriesWithEvents.m*
Script 8: *plotSensorData.m*

For all scripts below you will want to start MATLAB and navigate to the folder where the provided material (.M-functions and the data files) are located. That is, make MATLAB's current folder equal to the folder to which you downloaded this module (e.g. 'C:\...\My Documents'). After this you can call these functions via the command line or the Matlab GUI.

Script 1: *programmingBasics.m*

1. First, open the programmingbasics.M file using the matlab GUI. Run the following two lines to import the data into Matlab

```
filename = strcat(cd, '\data\genericEventData.csv');
data_events = csvread(filename);
```

Hint: To run a line, you can highlight it in the script in Matlab and press F9 (keyboard) or "run selection" on the GUI. You can also copy-paste this line into the command line followed by 'enter'. Note that all lines that begin with a % are "comments" meaning that they are there for communication between coders. Matlab will not process them as part of the script.

You have just loaded an array *csv_events_loc*, into the Workspace. More specifically this is an 87 x 3 double array, meaning it has 87 rows and 3 columns and the values are all numeric. Each event is in its own row. The start times of the events are in column 1, the end times in column 2 and the type of event is in column 3. There are four types of events, each coded as a number between 1 and 4. If you go to your workspace you should now find a variable named *csv_events_loc* in it. If you double click this variable in the workspace Matlab will open a variable window that allows you to view the array.

2. We will now begin accessing the data in this array. For example, to find out the start time of the third event, and store it in a new variable, you would run the line:

```
third_event_onset = data_events(3,1)
```

In general, the syntax for accessing (or referencing) array data in matlab
And then storing it as a new variable, is as follows:
name_of_variable_to_store = array_name(row, column). You can use this to access data from any row or column of data_events. Again, once you create a variable it is stored in your workspace. Also, try changing the numbers in the row and column positions to access other elements in the array, or changing the name of the variable to store another data element.

3. The next lines show you more techniques for accessing multiple array element at once, how to manipulate arrays by adding rows or columns, or by using simple algebra to shift array elements, how to use the size function to determine the size of your arrays, and how to write variables in your workspace to your computer.

Try each and see what it does.

Hint: The comments detail exactly what each line does.

4. For more practice, there are a set of exercises at the bottom of the script. Try to do each on your own, and scroll down for the answers to each.

Script 2: *simpleTimeseriesPlots.m*

1. Next, open the script called *simpleTimeseriesPlots.m*. We will also run this script line by line. Highlight and run the following lines:

```
%get to know the simple plot function
figure(1)
plot (1,1)
plot (1,1, 'rs','MarkerSize',10)
```

The first line of the script will open a new window with a figure object, Figure 1. The second line plots a single datapoint at the position x= 1 y =1 using the syntax *plot(xvalues, yvalues)*. However, as a single datapoint it is not visible on your plot. Therefore, in the third line we replot the datapoint at the same location specifying it as a redsquare (*'rs'*) at size 10 *('MarkerSize',10*). We will provide more information about these stylistic changes further later in this script.

2. Now run lines the next lines to plot your first timeseries!

```
%plot your first timeseries!
figure (1)
plot([1 2 3 4 5], [1 2 3 2 1])
```

Figure 1 should now contain a two lines in the shape of the top of a triangle. Again here we use the syntax *plot(xvals, yvals)*, but now, rather than providing a single datapoint, we provide a sequence of five x values and five y values as arrays of data, specifying five datapoints at " (x1, y1), (x2, y2) and so on, i.e. five points at the (x,y) locations: (1,1) (2,2), (3,3) (4,2) and (5,1). The *plot(xvals, yvals)* function automatically connects the points into a line.

Hint: if you would like to see the five points on the plot, run the lines under the comment `% add each datapoint as a red box.` The first line indicates to Matlab to draw on top of what is already plotted on Figure 1, rather than overwriting it. Line 18 plots the timeseries as a set of red square boxes.

3. Play with this simple timeseries to become familiar with the basic plot (xvals, yvals) syntax. To shift the datapoints so that the xvalues start at x =6 rather than x= 1, edit your script as follows and rerun it*: plot([6 7 8 9 10], [1 2 3 2 1]).* To make the triangle steeper, edit your script as follows and rerun it: *plot([6 7 8 9 10], [1 4 9 4 1]).*

4. Now try a variation on the basic *plot(xvals, yvals)* syntax: *plot (yvals)* by running the following lines.

```
% try the syntax plot(yvals)
figure (2)
plot([1 2 3 2 1], 'bs','MarkerSize',10)
```

You should see the original triangle, now with blue squares. If you leave out the *xvals* array, Matlab assumes that the *xvals* start at 1 and increase by 1 for the length of *yvals*.

5. Now we are ready to plot some actual data from a study! Highlight and run the following lines to open up two data files.

```
dataDir='C:\Users\kdeba\Dropbox\Libraries\Documents\presentations\2018
ICIS\workshop\finalmaterials\Kayamaterials\data\';saveDir='C:\Users\kd
eba\Dropbox\Libraries\Documents\presentations\2018ICIS\workshop\finalm
aterials\Kayamaterials\data\outputs\';

% use string concatenation function strcat to specify exact data files
fnameG=strcat(dataDir,'InfGaze_P6_4mo.csv');
fnameH=strcat(dataDir,'InfHands_P6_4mo.csv');

%read face and hand data using csvread command
gazeData=csvread(fnameG);
handsData=csvread(fnameH);
```

You have just loaded two sets of timeseries data, *gazeData* and *handsData* into the workspace. Double click them in the workspace to view the data in the variable viewer. Each array contains three binary timeseries representing a single infants' gaze

(*gazeData*) or manual contact (*handsData*) with a set of three objects, annotated from a single session with a four-month-old infant. All annotations were completed by human coders labeling all changes at a rate of 10 frames / second (detailed in de Barbaro, Johnson, Forster & Deak, 2016, cited in main text). Each variable is a 3x1864 array (i.e. – three rows, 1864 columns), and each row indicates the precise moment that the infant was making contact (1 = contact, 0= no contact) with one of three objects (row 1 = object 1, row 2 = object two, row 3= object three). Note that time is not explicitly represented in this timeseries: given that datapoints are provided at regular intervals, the timing of the activity can be calculated simply by noting which column it occurs in. That is, each column represents a tenth of a second meaning that each 10 columns represent 1 second.

5. To find out the duration of the coded session (in seconds), copy and paste the following line into your command window and hit return.

```
size (gazeData,2)/10
```

Hint: review *programmingBasics.M* if you don't remember how to use the *size* function, or how to store this value as a new variable in your workspace. Also, you can search for any matlab function online or the mathworks.com website to get more information about it including examples of how to run it.

6. Now let's begin plotting this data using the simpleTimeseries.M script. Run the following lines.

```
%plot gaze to first object
figure(1)
plot (gazeData(1,:))    % plot(yvals)  % if time is regular across
your datastreams you dont need to indicate timestamps/ xvals
title('Infant gaze: Object 1')

axis ([0 size(gazeData,2) 0 1.5 ])
% axis([XMIN XMAX YMIN YMAX])
```

After opening a figure, the second line of the script plots the entire first row of the *gazeData* array (i.e. Gaze to Object 1) onto the figure using the syntax: *plot(yvals)*. The remaining lines add a title and adjust the axis to give the line a bit of white space as background. The syntax is *axis([XMIN XMAX YMIN YMAX]):* an array with four values that specify the edges of the figure space using x-y coordinates. To add 10 seconds of additional buffer around the x axis, you can increase the *XMAX* value by running the line *axis ([0 1964 0 1.5])* or, to if you want this to dynamically buffer based on the actual size of the array, use axis *([0 size(gazeData,2)+100 0 1.5])*.

Looking at the figure itself, we can see a line that is zero until about frame 1000 and then bounces between 0 and 1 until about frame 1800, meaning that the infant was looking to and from object one between approximately 100-180 seconds. Remember, each frame is 1/10 a second and for each frame that the baby was looking to object 1, the timeseries will be 1, when it is not looking to object one, the timeseries will be 0.

7. Run the following lines to plot gaze to all objects on a single plot.

```
% plot gaze to all objects on one plot
```

```
figure (2)
plot (gazeData(1,:), 'r')
hold on
plot (gazeData(2,:),'g')
plot (gazeData(3,:), 'c')
title('Infant Gaze: three objects')
axis ([0 size(gazeData,2)+100 0 1.5 ])
```

We can now see the pattern of the infant's gaze across the entire session: infants first look back and forth at object 2 (in green), then object three (in blue) and lastly, object 1 (in red). This one-at-a-time pattern of gaze is very different from that of older infants, who rapidly alternate their gaze between multiple simultaneously available objects (see de Barbaro, Johnson, Forster & Deak, 2016). You can use the magnifying glass to zoom in on any aspect of the plot (note that you can specify horizontal or vertical zoom only as well). The next five sets of comments show you more options for changing aesthetic aspects of your plotted datapoints/timeseries, known as "line specification".

8. Next, lets run the following lines to plot gaze to all objects on a single plot.

```
% plot gaze and hands data together using two linked subplots
figure (5)
ax1= subplot(2,1,1);    % syntax is subplot(total rows of plots, total
columns of plots, position of current plot to be plotted)
plot(gazeData(1,:),'r')
axis ([0 size(gazeData,2)+100 0 1.5 ])
title('Looking at red toy')

ax2= subplot(2,1,2);
plot(handsData(1,:),'r')
axis ([0 size(gazeData,2)+100 0 1.5 ])
title('Touching red toy')    %-----> try zooming in on one
```

You now have two plots within a single figure: one that indicates when infants look to the red object, and another that indicates when it touches or holds the red object. The *subplot* function is used to create multiple plots within a single figure with the syntax *subplot(total rows of plots, total columns of plots, position of current plot to be plotted)*. Thus, *subplot (2,1,1)* makes a space for two plots stacked on top of each other horizontally, and plots whatever data is specified in the next *plot* function in the first plot grid. The code *subplot(2,1,2)* specifies that the second plot function i.e. *plot(handsData(1,:),'r')*, should be placed in the second position of the 2x1 plot grid. Try plotting another figure where there are two plots side by side by editing the subplot commands as follows: *subplot(1,2,1)* and *subplot(1,2,2)* columns. Finally, try zooming in on the gaze plot. Notice that the x axis of the two plots are not linked, one zooms in and the other stays zoomed out. Run the line *linkaxes([ax1,ax2],'x')*, to link the x axes of the plots.

9. The lines prefaced by the comment `% generate and plot summed gaze and hands data` show you how to easily create and plot new timeseries that represent different ways of summarizing the data in your existing arrays. The lines after the `%save your plots` comment show you how to plot the most recently edited figure as a variety of image types.

10. Finally, try the exercises at the end of the script to practice and extend your new skills! Congratulations, you've graduated to the next script in the module!

Script 3: *simpleEventDataPlots.m*

1. Next, open the script called *simpleEventDataPlots.* We will also run this script line by line. Highlight and run the following lines:

```
clear all

%use the simple plot function to plot a single line representing an
event
figure (1)
plot([ 1 3], [ 2 2], 'c', 'LineWidth',10)

%resize axis
axis([0 10 0 4])
```

The first line of the script cleans up your workspace, the second lines use the *plot* function to plot a single line representing an event. The plot syntax for drawing a single line -- how we will represent each event -- is an extension of the *plot (xvals, yvals)* syntax:

plot([x1 x2], [y1 y2]) , or, in plain English:
plot([event_onset event_offset], [vertical_axis_position vertical_axis_position])

Your first plotted line begins at the point (1,2) and goes to the point (3,2) -- corresponding to a event that begins at second 1 and ends at second 3. It is plotted (somewhat arbitrarily!) in blue, at the vertical axis position of 2.

2. Now let's add some more events. Start by running the first set of three lines below. Observe your results, then run the second set.

```
%add some more "events" to your plot, including some at different
vertical positions
hold on
plot([ 5 6], [ 2 2] , 'c', 'LineWidth',10)
plot([ 2 6], [ 1 1], 'c', 'LineWidth',10 )

%add some more "events" to your plot in another color
plot([ 3.1 4.9], [ 2 2] , 'g', 'LineWidth',10)
plot([ 6.1 9.1], [ 2 2] , 'g', 'LineWidth',10)
plot([ 6.2 9.1 ], [ 1 1], 'g', 'LineWidth',10 )
```

You can imagine that the top line (at the y = 2 position) indicates infant gaze data, and the bottom line (y=1 position) indicates infant touch data, with each color representing gazing at or touching a different object. In a few steps, we will plot real gaze and hand events. Note that when you are plotting events, you need to plot each event by drawing one line one at a time. It would be tedious to specify each line by hand for a large dataset. In the next step, we will practice looping through an array accessing and plotting data for each consecutive event.

3. Now let's run the following set of lines, one section at a time. This time, we will plot the gaze data from a simple array formatted as an event data array.

```
    %create a gaze event array
    gazeEvents  = [
    5,    6,    1 ;
    3.1 , 4.9 , 2 ;
    6.1 , 9.1 , 2 ];

%we can plot all the events in the array using a for loop
figure (2)
hold on
countRows = size(gazeEvents,1)

for currentRow = 1: 1: countRows
    plot(gazeEvents(currentRow,[1 2]), [2 2], 'c', 'LineWidth',10)
end

%resize axis
axis([0 10 0 4])
```

The first set of lines creates an array (*gazeEvents*) from scratch, that has each events in a new row, with start times in first column, stop times in the second column (in seconds) and the type of event in the third column (in this case, gaze events to object 1 or 2).

4. Check your workspace to see that your *countRows* variable has the value '3', representing the number of rows in new *gazeEvents* array. The section beginning with "*for*"…."*end*" is called a "for loop". Here, we create a variable called *currentRow* that begins with the value '1', and in increments of 1 increases until it reaches the value of countRows (3). For each value of currentRow (i.e. 1-3), the script will run all of the lines between for and and – i.e. it will loop through to plot each event as specified in each of the three rows of the *gazeEvents* array!

Hint: To see how this happens, type the line *currentRow=1* into your command window and hit return. Then run the *gazeEvents(currentRow, [1 2])*. This is the equivalent of *gazeEvents(1, [1 2])* – i.e. *[5 6]*! You are telling Matlab to access the first two columns of the first row of the *gazeEvents* array (see your *programmingBasics* script for practice if this doesn't make sense). So, the first time looping through the array, the line:

```
plot(gazeEvents(row,[1 2]), [2 2], 'c', 'LineWidth',10)
```

is read by matlab as:

```
plot([5 6], [2 2], 'c', 'LineWidth',10)
```

the second time, it is read as:

```
plot([3.1 4.9 ], [2 2], 'c', 'LineWidth',10)
```

and the third time, as:

```
plot([6.1 9.1 ], [2 2], 'c', 'LineWidth',10)
```

You can set *currentRow* to the value '2' and then '3' and rerun the command *gazeEvents(currentRow, [1 2])* to see this in action.

5. Now try some variations on this main code by running the lines starting the comment `% lets try some simple variations!` And going through the

complete for loop. Compare your results to those in Figure 2. If you run it as is, only one line that calls the *plot* function is active in the for loop.

6. You can see additional variations if you rerun this for loop lines after adding a % sign to the front of one of the other lines that calls the *plot* function (called "commenting out a line") and removing the % sign from the next line of the script (called "uncommenting" a line). Do this for each of the four lines that call the *plot* function in the for loop.

Hint: This is what you should be running the second time around.

```
clf(3) % clear figure 3
figure (3)
axis([0 15 0 10])
hold on
countRows = size(gazeEvents,1);

for row = 1: 1: countRows

    % this shifts the events vertically up (y axis)
    % plot(gazeEvents(row,[1 2]), [ 7 7] , 'c', 'LineWidth',10)

    % this shift events forward in time by five seconds
     plot(gazeEvents(row,[1 2])+5, [ 2 2], 'c', 'LineWidth',10)

    % this puts the x axis in minutes rather than seconds
    % you will need to zoom in to see the event!
    % plot(gazeEvents(row,[1 2])/60, [ 2 2], 'c', 'LineWidth',10)

     % this plots events in different horizontal positions on the y
     % axis depending on their event type gazes to object 1 are
     % plotted on the y=1 axis, gazes to object 2 are plotted on the
     % y=2 axis,
      % plot(gazeEvents(row,[1 2]), gazeEvents(row,[3 3]) , 'c',
     % 'LineWidth',10)

end
```

Script 4: *multiParticipantEventPlotting.m*

1. This script builds upon the basic event plots from the *simpleEventDataPlotting* script. We will use it to plot a new dataset: moment-by-moment affect data from three mother-infant dyads engaging in free play (contributed by Dr. Sherryl Goodman from Emory University). All mothers in the study have a history of depression. Three states of infant affect are annotated– positive, neutral, and negative (+ 1 to -1) and mother affect is annotated on a 7-point scale, from highly positive to highly negative (+3 to -3). For the purposes of the visualization, we will collapse maternal affect into three states: positive (encompassing +3, +2 and +1) neutral (0) and negative (-1, -2 and -3) to match the infant states. We will use both color and vertical positioning to differentiate each participants' states of affect.

Open the script and check it out. The sections of the script that plot data (i.e. the for loops beginning with `%plot infant affect states` and `%plot mom affect states` should remind you of the for loop in which you ploted from the *gazeEvents* array in the *simpleEventDataPlotting script* . In this script, we embed this plotting code in a for loop that cycles through multiple participants. Let's look at some of the details of what happens in the first for loop, which begins as follows:

```
for pID =  [3414  3367 3532 ]

    % use string concatenation function strcat to specify exact data %
    files
    fname=strcat(cd,'\data\MoInfAffectArray_', num2str(pID), '.csv');

    %read data using csvread command
    pData=csvread(fname);
```

These lines open a data file, similar to what we've seen before in earlier scripts. Note however that in this script, the fname variable is now created using the dynamic value *pID* which changes its value within each of iteration of the top for loops - rather than hard-coding these values as in our prior scripts.

7. To plot mother and infant affect states in different colors and on different axis positions we need to do one more thing. Let's go back to the top of the script to look at the following lines:

```
%first we make a color array, assigning a color and position to each
state of mother and infant affect

infAffectColor= {
      1 ,  'b', .1 ; % +1 -  approach (positive) ; blue
      0 , 'k', 0 ; %    0 - neutral; black
      -1, 'r' , -.1}; % -1  - withdrawal (negative); red
```

In this code we specify a color array, assigning a color and position for each of the three infant affect states. Note the curly brackets used in initializing this array. Because this array contains letters (i.e. "strings" in matlab) it needs to be initialized as a special type of array, called a "cell array".

8. Now let's take a look at the for loop where we will plot infant affect states. The code in this section is as follows:

```
%plot infant affect states
for curAffectState= [ 1:3 ]
        %get label, color and position info for the current infant
        affect state
        curAffColor = infAffectColor{curAffectState,2};
        curAffValue = infAffectColor{curAffectState,1};
        curAxisPos = infAffectColor{curAffectState,3};

        %plot infant affect for this state
        %grab all the event rows for this state of infant affect
        indices = find(infantData(:, 4)== curAffValue);
        dataRows = infantData(indices,:);
```

```
        %count the number of affect events
        countRows = size(dataRows,1);

       %for loop to plot all infant affect events of the current
    state
        for row = 1: countRows
           hold on
           plot(dataRows (row,[1 2]), [ curAxisPos curAxisPos],
          'Color',curAffColor, 'LineWidth',10 )
        end
    end
```

This code contains a nested for loop structure. The main for loop cycles through the three affect states. For each type affect state (positive, negative, and neutral) there is a for loop to plot all the events that match the current affect state. The line:

```
curAffColor = infAffectColor{curAffectState,2};
```

grabs the color information for the associated infant affect state and saves it in the variable *curAffColor*. To see what color it will set for *curAffectState =1*, run the line: *infAffectColor{1,2}* from the command window. You should get *'b'* for black, the value in the first row, second column of the *curAffColor* variable.

9. Now let's take look at the plot code itself within one of the for loops

```
plot(dataRows (row,[1 2]), [ curAxisPos curAxisPos],
'Color',curAffColor, 'LineWidth',10 )
```

Note: The plotting function now uses the dynamic value *curAffColor* – which changes its value within each loop – rather than hard-coding a single color as in figure 5. Similarly, curAxisPosition will change with each loop through the three affect states.

10. Now lets return to the top of the for loop looping through each participants data. Check out the following lines

```
infantData = pData(~(isnan(pData(:,4))),:); %inf data located in col 4
momData = pData(~(isnan(pData(:,3))),:); %mat data located in column 3
```

Lots of data coming out of common annotation software has multiple types of annotations in the same file. These are often stored in separate columns of the data. If we want to access some of this data (e.g. infant vs. mom affect annotations) but not all of it (e.g. infant and mom problem data) we can find and store those data in a new variable.

There are multiple functions in the first line – lets break them down. *isnan(pData(:,4))* gives you the row numbers of all rows that have no values in the fourth column. The ~ symbol means "not", so ~ *isnan(pData(:,4))* – gives you the row numbers of all rows that have values in the fourth column: ie those rows that have any infant affect data, and stores them in the variable *infIndices*. Finally, the next line creates a new variable that takes the stored Row numbers (indices) and provides the actual values in those rows (i.e. the infant affect events themselves). The line finding maternal affect data combines these two lines into a single line, but the outcome is the same.

To see these in action, you can set a "breakpoint" in your file. That is, point your mouse to the left of the line you are interested in, just to the right of the line number. If you right click your mouse here, you will see a dot appear –a breakpoint. When we run the script in the next step this breakpoint will "stop" the script at this location, allowing you to observe all variables present in the workspace and their values.

11. Now we are ready to run the full script. This is a complete file so you can run it by either pressing the run button in the GUI, or copy-pasting the script title into the command window and hitting return.

    If you set breakpoints, you will see a green arrow stopped at your first breakpoint. This indicates how much of the script has already been run. In this "debugging" mode you can advance (step) through the script one line at a time by pressing the *Step* button in the GUI. The workspace shows only those variables that have already been created in the script up to that point, and the variables will dynamically update as you step through the function. You can advance to the next breakpoint by pressing the *Continue* button. Or you can remove any individual breakpoint by clicking the red dots and hitting 'run" again to take you to the end of the script.

    If you haven't set any breakpoints, you will get three plots showing mother and infant affect for three different participants. The x-axis shows time (in seconds) and the y axis distinguishes different dimensions of affect. Maternal affect is represented in three rows at the top, infant affect is represented in three lower rows. For both mothers and infants, the highest of the three rows (in blue) represent positive affect, the middle row (in black) represents neutral affect, and the bottom row (in red) represents negative affect. We discuss these figures in the main manuscript text.

Hint: Note that these colors and horizontal positions are set in *infAffectColor* and *momAffectColor*. These parameters are arbitrary in that one could change them to any other color or positioning- however, setting these values in an intuitive way (higher affect higher, red = distress, same colors for same states across mother and infant) can greatly facilitate the comprehension of your plots. Additionally the simple black lines ("kebab lines") plotted before the main mother and infant plot loops help to orient the observer to where data will be present. Try removing them (or changing their positioning) to see how it affects the overall plot. Also, try adjusting the height and width of these plots and see how that affects the spacing between the dimensions of affect. The plots shown in the manuscript text were adjusted to be long and thin (hotdog shape) such that each dimension of mother and infant affect was touching as this highlights the patterns of contingency within and between mother and infant affect.

Script 5: *annotationImport.m*

1. Open up your script in matlab. This script transforms the outputs of annotation software commonly used in psychology, such as Elan, DataVyu, Noldus, or Mangold Interact into a more usable format that can be easily worked with within Matlab. It saves the transformed data as a csv file that can be imported by other scripts in this module or otherwise. We'll come back to the script in a moment.

2. Open the file *3414goodman.csv* in your favorite spreadsheet viewer. This the raw exported annotation file from which we derived the data for Script 4. It contains labels for moment-to-moment changes in mother and infant affect. It comes from the software Mangold Interact. As with exports from other annotation software, this file contains many rows of data. Each row corresponds to a single event, and includes: onset and offset times as well as coding dimensions—in this case, MaternalAffect, InfantAffect, Maternal Problem Data, and Infant Problem data. In this file, maternal affect data is coded on a continuous numerical scale (ranging from -3 to +3) and infant problem data is coded categorically (withdrawal, neutral or positive, or W, N and P). Data exported from Annotation software typically contains column headers, which we will use to identify the annotations. this code is written for outputs where each participant's data is saved in a separate csv file.

3. Lets take a look at a few aspects of the script in detail. Set a breakpoint in your script at the following line and then advance one step forward so that the script runs the line:

```
pDataTable = readtable(fname,'Delimiter',',');
```

Here, we use the import function *readtable* (vs the more common csvread) as annotation files typically contain both numbers and strings (ie letters and words).

4. Click the variable in the workspace. Notice it is different than the arrays we have been working with: it contains headers. It can also contain a much wider variety of data (both numerical and text data). However, it is an unwieldy data format and many functions that we can use with arrays do not work with tables.

5. The goal of the rest of the code is to translate the data from the table int an array format, and then store the array as a csv for later processing. Given the large variety of types of data stored in table we will use different strategies to translate different types of data, which we will cover in the next steps.

6. For data that are numeric and will always be in the same order across datafiles – like our onset and offset times (columns 4&5), you can use the function *table2array* to simply specify the column numbers that you want to put in the new array, as follows.

```
pData = table2array(pDataTable(:,[4,5]));
```

7. For data that are numeric but might not be in the same order (such as the MaternalAffect data), use the column header name to simply put that column into *pData*. Note that before doing this we will add four empty columns to *pData* to make space for the new data. We could have used the same technique for onset and offset data as well.

```
pData = [ pData , NaN(size(pData,1), 4)];
pData(:,3) = pDataTable.MaternalAffect;
```

8. Data that are text-based, i.e. those whose event data contain any letters or non-numeric symbols (such as Infant Affect and both Mother and Infant Problem data) are the most complex to translate. As arrays can contain numeric information only we must convert these to numeric values before transferring them to the new *pData* array. For each annotation in *pDataTable* we will replace it in *pData* (array) with its corresponding numeric code, as follows:

```
% 'A' (infant approach/ positive affect) -> +1
% 'N' (infant neutral affect) -> 0
% 'W' ( infant withdrawal/ negative affect)  -> -1
```

9. To do this we first define a function that can search within each cell of the table to find specific annotations within their contents.

```
cellfind = @(string)(@(cell_contents)(strcmp(string,cell_contents)));
```

10. Next, we use this function to search for specific annotations within specific columns of the data. Given that 'A' is the annotation for infant positive affect, we use the functions find, *cellfun* and *cellfind* to search for this value in the Infant Affect column. The first of the following lines stores the numbers of all the rows in this column that contain an 'A' in a variable called *indicesPA*. The next line puts our selected numeric code (+1 for 'A') into the corresponding rows of column 4 of *pData*. We will put them into column four since that is the next empty column.

```
indicesPA  = find(cellfun(cellfind('A'),pDataTable.InfantAffect));
pData(indicesPA, 4)=1;
```

11. The remainder of the script transfers additional text-based annotations from *pDataTable* to *pData* to create an array with six columns from the original export file: Onset, Offset, MaternalAffect, InfantAffect, Maternal Problem data, Infant Problem Data. Finally, we save *pData* into a .csv file. Note that *pData* no longer has headers so you must carefully note which column corresponds to which dimension of coded data. Additionally, you must carefully note what annotation in each value in each column corresponds to.

Script 6: *convertEvents2Timeseries.m*

1. This script converts event data into time series data and stores the output in the data folder. This is useful as some analyses require timeseries format. Note that this script will not work on raw /unprocessed event data directly exported from common annotation software. This script is written to work with annotation software exports that have been processed via Script 5 (*annotationImport)*. Alternatively, they will work with any all-numeric event-data where individual events are stored in consecutive rows, and that include an onset and offset column and any coded dimensions in other columns.

2. The script is set up to transform the infant affect events we worked with in script 5 into timeseries data.

```
dataCol = 4; % indicate the column that has the data you want to
convert
dataLabel = 'Infant Affect';
eventOnsetCol = 1; % indicate the event onset column
eventOffsetCol =2;% indicate the event offset column

% set sample_rate (the interval between two consecutive time stamps
of converted timeseries data)
%.1 corresponds to 10fps
% 0.034 corresponds to 29.97fps (common in US)
%.04 corresponds to 25fps

sample_rate = .04;
```

Note: To modify the script to work with other data, change the data parameters in the indicated lines and the particulars of the filename details within the for loop looping through each participant.

3. Now run the script and open one of the new timeseries csvs from the data folder. The first column is time in seconds, with the interval between timestamps determined by the *sample_rate* variable. Between the onset and the offset times for each event in the original datafile you should see the value corresponding to that affect state. Congratulations, you now have a timeseries!

Script 7: *plotTimeseriesWithEvents.m*

1. Take a look at the script. This script plots infant participants' heart rate data (in timeseries format) as it changes over the course of a session that includes various types of tasks (in event-data format), including a habituation task (visual paired comparison; VPC) and watching various video clips (smiling baby, crying baby, channel hopping; chan hop) task. It plots multiple participants' data at once. It also introduces how to add text labels to plots.

2. See if you can figure out how the script works. If you have gone through the earlier scripts in this module, you will be familiar with most of the code here. Add some breakpoints if you are unsure of what a line or part of a line does.

3. You haven't yet seen the *text* function before. It allows you to insert a label into your plot. The syntax is, *text(x_axis_position_for_start_of_text, y_axis_position_for_start_of_text, the_text_itself, 'Rotation', degrees_to_rotate).* In this script it is in the loop that plots event data, as follows:

```
text( startTime/1000, 185, eventName, 'Rotation', 45)
```

In this case, the text itself is stored in the variable *eventName*, which changes dynamically on each loop through the list of events to match the task name, as specified in the *eventTypesCell*.

4. Run the script by either pressing the run button in the gui, or copy-pasting script title into the command window and hitting return. You should get two new plots, one each from two different participants. We discuss these figures in the main manuscript text.

Script 8: *plotSensorData.m*

1. This script plots data files from common mobile sensor outputs, in this example, from the Empatica E4 device. It can plot multiple participants of data. It translates a common format of time data for sensors (unix time format; see https://en.wikipedia.org/wiki/Unix_time) into easier to work with matlab time formats. It uses subplots to plot three different physiological markers (heart rate, electrodermal activity (EDA) and motion (ACC) into a single plot, and it uses link axes to connect the subplots when zooming or panning.

2. Run the script by either pressing the run button in the gui, or copy-pasting script the title *plotSensorData* into the command window and hitting return. You should get a new plot!