Behavior Analysis in R

Version 1.0 | Last updated 28 Aug 2015

Introduction

This is basically a walkthrough as to how to use the behavior code. For the sake of not getting too bogged down in minute details, I haven't described every option or every function here. If there's something you want to do, especially something you've seen done before, just shoot me an email to ask how to do it.
Also, if you're having trouble, try downloading the latest version of this file from https://www.dropbox.com/sh/xaxyn0cndwt7353/AAA_f1MJHI8bB1TkPwiAyi7Oa?dl=0 and making sure you have the most recent version of the code files before you send a bug report. For more on bug reports, see the end of this file.

Step 1: Download code files

Go to the GitHub repository (https://github.com/FernaldLab/behaviorcode) and click "Download ZIP" on the right to get all the code files (unfortunately there is no way to download individual files). You will need the following files:

```
behavior.R
bootstrap_rewrite2.R
clusterycode.R
powerBootstrap2Independent_fast.R
```

Now, open R and `source()` in all of these files. For example, if you saved them to a folder called "code" on the desktop, you would type:

```
> source('~/Desktop/code/behavior.R')
> source('~/Desktop/code/bootstrap_rewrite2.R')
> source('~/Desktop/code/clusterycode.R')
> source('~/Desktop/code/powerBootstrap2Independent_fast.R')
```

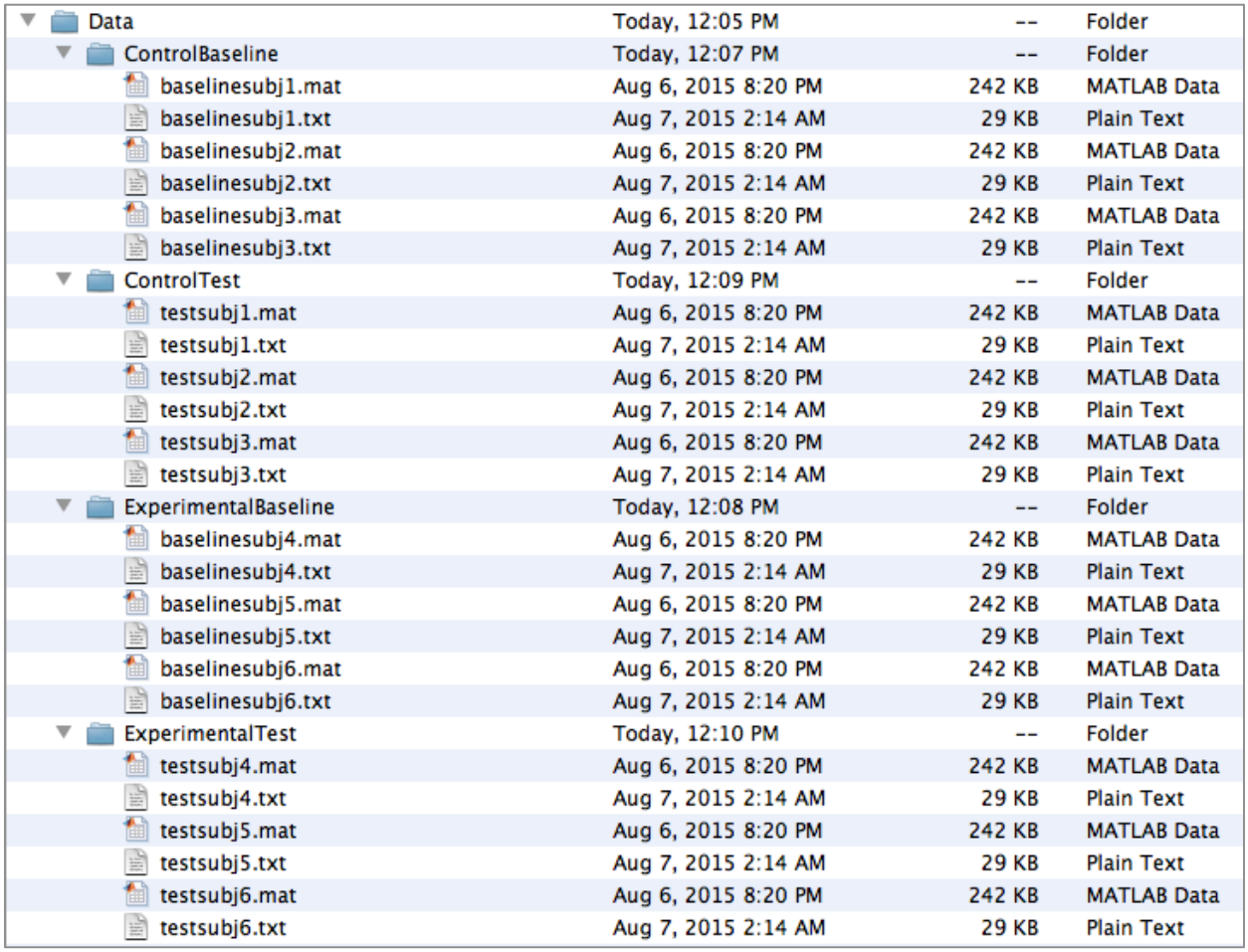
You'll also need to install some R packages used in the code:

```
> library(WGCNA)
> library(stringr)
> library(survival)
```

If any of these fail, you might need to use `install.packages()` instead of `library()`.

Step 2: Get your data into R

Make a folder for each group and each timepoint in your experiment. For the rest of this guide, let's say that you have two experimental groups, "Experimental" and "Control", and that you have logs from two different timepoints, "Baseline" and "Test". In this case, you'd want four folders: "ExperimentalBaseline", "ExperimentalTest", "ControlBaseline", and "ControlTest". It doesn't matter what you name them as long as you can keep track of which is which. Put the appropriate scorevideo logs in each folder, then put all the folders in one big folder (let's call it "Data" and say it's on the desktop). You should have something that looks like this:



Data	Today, 12:05 PM	--	Folder
ControlBaseline	Today, 12:07 PM	--	Folder
baselinesubj1.mat	Aug 6, 2015 8:20 PM	242 KB	MATLAB Data
baselinesubj1.txt	Aug 7, 2015 2:14 AM	29 KB	Plain Text
baselinesubj2.mat	Aug 6, 2015 8:20 PM	242 KB	MATLAB Data
baselinesubj2.txt	Aug 7, 2015 2:14 AM	29 KB	Plain Text
baselinesubj3.mat	Aug 6, 2015 8:20 PM	242 KB	MATLAB Data
baselinesubj3.txt	Aug 7, 2015 2:14 AM	29 KB	Plain Text
ControlTest	Today, 12:09 PM	--	Folder
testsubj1.mat	Aug 6, 2015 8:20 PM	242 KB	MATLAB Data
testsubj1.txt	Aug 7, 2015 2:14 AM	29 KB	Plain Text
testsubj2.mat	Aug 6, 2015 8:20 PM	242 KB	MATLAB Data
testsubj2.txt	Aug 7, 2015 2:14 AM	29 KB	Plain Text
testsubj3.mat	Aug 6, 2015 8:20 PM	242 KB	MATLAB Data
testsubj3.txt	Aug 7, 2015 2:14 AM	29 KB	Plain Text
ExperimentalBaseline	Today, 12:08 PM	--	Folder
baselinesubj4.mat	Aug 6, 2015 8:20 PM	242 KB	MATLAB Data
baselinesubj4.txt	Aug 7, 2015 2:14 AM	29 KB	Plain Text
baselinesubj5.mat	Aug 6, 2015 8:20 PM	242 KB	MATLAB Data
baselinesubj5.txt	Aug 7, 2015 2:14 AM	29 KB	Plain Text
baselinesubj6.mat	Aug 6, 2015 8:20 PM	242 KB	MATLAB Data
baselinesubj6.txt	Aug 7, 2015 2:14 AM	29 KB	Plain Text
ExperimentalTest	Today, 12:10 PM	--	Folder
testsubj4.mat	Aug 6, 2015 8:20 PM	242 KB	MATLAB Data
testsubj4.txt	Aug 7, 2015 2:14 AM	29 KB	Plain Text
testsubj5.mat	Aug 6, 2015 8:20 PM	242 KB	MATLAB Data
testsubj5.txt	Aug 7, 2015 2:14 AM	29 KB	Plain Text
testsubj6.mat	Aug 6, 2015 8:20 PM	242 KB	MATLAB Data
testsubj6.txt	Aug 7, 2015 2:14 AM	29 KB	Plain Text

You only actually need the `.txt` files, but having the `.mat` files there doesn't hurt anything.

Next, get your data into R:

```
> my_data <- .getDataBatch("~/Desktop/Data/")
```

The `my_data` in this call is a variable that holds all of your logs. You can choose whatever name you want (for example, you could call it `pgf2a_dat` or `aggressionAssay`) but you need to use the same name in all the following steps. If you have more than one dataset, give them different names.

Follow the prompts to either use the start of video as the assay start for all logs, or select a mark left in the logs in scorevideo to be used as the assay start. Saving a mark as a default will result in all logs with that mark using it as an assay start. You can have more than one default at a time.

The prompts will then direct you to check and see if you more than one name for the same behavior. If you have a lot of these to fix, you may want to say "no" and use a cleaning function later. (see "Write a cleaning function" below for more information).

Finally, enter the length of your assay in seconds. Behaviors that occur after the end of the assay will be left out of the logs. For example, for a 30-minute assay you would enter 1800.

Double check that this all went well by typing

```
> names(my_data)
```

and verifying all your logs are there, then try

```
> my_data[[1]]
```

to see one of your logs and make sure everything seems reasonable.

Step 3: Get your data ready for analysis

There are several little things you may need to do before you analyze your data, but you probably don't need to do all of them. Just as a note, CAPITALIZATION MATTERS for all of these things!

Most of these functions work by doing something to your data to produce a modified copy (the part after the <–) then saving it back to the same variable (the name before the <–). If you're not sure whether you're doing something right, you might want to save it to a temporary variable instead (for example, replace `my_data <-` with `my_data_tmp <-` in any of these function calls). You can then save it back to your "real" data by typing

```
> my_data <- my_data_tmp
```

You can also make multiple versions of your data this way, as long as they all have different names (ie `my_data_startOnly`, `my_data_renamed`, `my_data_noApproach`, etc.)

Stitch Logs Together

Do this if: you have videos that were long and got split into multiple files, so you have multiple scorevideo logs for the same subject at the same time. Most people will not need this.

You need to have marked assay starts in your logs for this to work; otherwise, the program will not know which order the different log files should be in.

Unify the different log files by calling

```
> my_data <- .stitchLogsTogether(my_data)
```

and entering the names of each of your subjects one by one (with the foldername and slash before them!). Check each time that the program has pulled up *all* the log files belonging to that subject at that timepoint, and *only* the log files belonging to that subject at that timepoint.

This is annoying and takes a long time. If you can code, you might want to write your own function to do this; look at `.stitchLogsAustin` for an example.

Sort Folders into Groups and Timepoints

Do this if: Everyone has to do this.

Call

```
> my_data <- .pairGroups(my_data)
```

Follow the prompts to tell the program how many experimental groups you measured at how many timepoints. If you only had one experimental group at one timepoint, you won't need to input anything, but you must still run this function.

If you had more than one timepoint and know how to code, you may wish to provide a function to match up logs across timepoints:

```
> pairMyLogs = function(subjectLog, followupGroup) {
+     subjectID = gsub("^.*/.*(subj[0-9]*)\\.txt", "\\1",
subjectLog);
+     pairLog = grep(subjectID, followupGroup, value = T);
+     if (length(pairLog) > 1) {
+          warning("MORE THAN ONE MATCH", immediate. = T);
+          pairLog = pairLog[1];
+     } else if (length(pairLog) < 1) {
+          warning("NO MATCH", immediate. = T);
+          pairLog = "";
+     }
+     return(pairLog);
+ }
> my_data <- .pairGroups(my_data, pairMyLogs)
```

Your pairing function should take as its first parameter a log name from the first timepoint (preceded by a folder and a slash) and as its second parameter a character vector of log names from any single followup timepoint (again preceded by their folder and a slash). It should return the log name from the character vector that is the followup log for the log given as the first parameter. No error checking is performed to make sure this name is valid; that's up to you.

Edit: Change behavior names

Do this if: you have single behaviors with multiple different names (thus treated as multiple behaviors), you want to change specific behaviors into categories like "aggression" and "courtship", you are dissatisfied with capitalization, you found a spelling mistake, …

If you want to change "Female follows" to "Female Follows", you might call

```
> my_data <- .replaceBehAll(my_data, "Female follows", "Female
Follows")
```

If you want to change multiple behaviors to "Aggression", you could call

```
> my_data <- .replaceBehAll(my_data, c("frontal display",
"lateral display", "border dispute", "chase"), "Aggression")
```

AT ANY POINT, you can call

```
> .printFindDupBehaviors(my_data)
```

to see what behavior names you currently have in your data and how many logs they are in.

Edit: Pair behaviors to make them durational

Do this if: you have two behaviors like "Enter pot" and "Exit pot" that you'd like to pair up so you can get a duration of time spent in the pot.

For enter and exit pot, call

```
> my_data <- .makeDurationalBehaviorAll(my_data, "Enter pot",
"Exit pot")
```

If there are not the same number of starts and stops, this function will stop and tell you where the problem is in the log so you can correct it. If you want the function to just warn you instead of stopping and get rid of unpairable starts and stops, call

```
> my_data <- .makeDurationalBehaviorAll(my_data, "Enter pot",
"Exit pot", removeExtra = T)
```

Edit: Make durational behaviors non-durational

Do this if: you scored "Quiver" and "Lead" as durational and you want to consider it now as a point behavior, or you want to not consider durations for anything.

For JUST "Quiver" and "Lead" (or any subset of behaviors) call

```
> my_data <- .filterDataList(my_data, startOnly = c("Lead",
"Quiver"))
```

To make all behaviors non-durational, call

```
> my_data <- .filterDataList(my_data, startOnly = TRUE);
```

AT ANY POINT, you can call

```
> .startStopBehs(my_data)
```

to see which behaviors are currently durational.

Edit: Ignore behaviors

Do this if: you suddenly realized that fish approach each other all the time, so you don't want approaches included in your analysis. Or if you scored a behavior like "Female Flee" in half your logs, but then decided you didn't want to score it anymore, so you want to remove it from the logs where it was scored to be consistent.

To remove "Female Flee", call

```
> my_data <- .filterDataList(my_data, toExclude = "Female Flee")
```

To remove "Approach male" and "Approach female", call

```
> my_data <- .filterDataList(my_data, toExclude = c("Approach
male", "Approach female"))
```

Write a cleaning function

Do this if: you had to do a lot of these edits and you're sick of typing the same thing over and over and over again.

You can write a simple function that does multiple renames, pairings, etc. Here's an example:

```
> cleaningFunction = function(my_data) {
+     my_data <- .filterDataList(my_data, toExclude=c("approach",
"APPROACH", "BITE", "flee", "QUIVER"))
+     my_data <- .filterDataList(my_data, startOnly=c("chase",
"female FOLLOW", "FLEE", "FOLLOW", "lead", "male CHASE", "male
LEAD", "male QUIVER", "quiver"))
+     clean <- .replaceBehAll(my_data, "female FOLLOW", "female
follow");
+     clean <- .replaceBehAll(clean, "female IN POT", "female in
pot");
+     clean <- .replaceBehAll(clean, "FLEE", "female flee");
+     clean <- .replaceBehAll(clean, "FOLLOW", "female follow");
+     clean <- .replaceBehAll(clean, "inside pot", "male in pot");
+     clean <- .replaceBehAll(clean, "inside POT", "female in
pot");
+     clean <- .replaceBehAll(clean, "male BITES", "bite");
+     clean <- .replaceBehAll(clean, "male CHASE", "chase");
+     clean <- .replaceBehAll(clean, "male IN POT", "male in
pot");
+     clean <- .replaceBehAll(clean, "male LEAD", "lead");
```

```
+     clean <- .replaceBehAll(clean, "male QUIVER", "quiver");
+     return(clean);
+ }
> my_data = cleaningFunction(my_data)
```

Step 4: Save your data as an R object

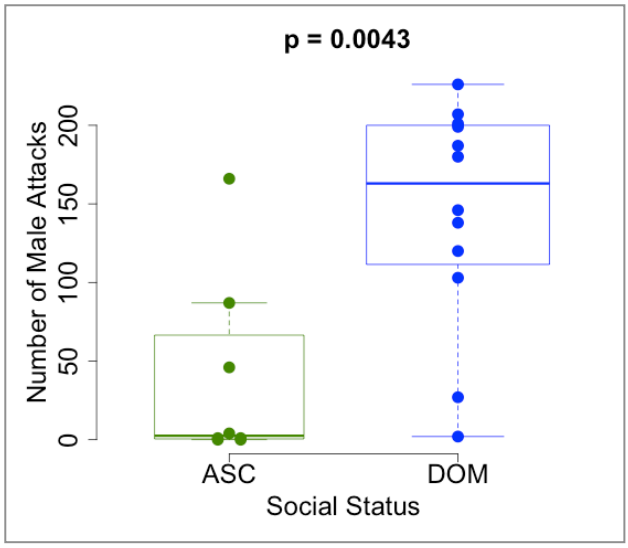
Just in case something happens, R crashes, you accidentally write over your data, etc.

```
> dump("my_data", file = '~/Desktop/Data/my_data_in_R.R')
```

If something does happen, you can get your data back by sourcing it in:

```
> source('~/Desktop/Data/my_data_in_R.R')
```

Step 5: Run statistical comparisons.



You probably want to create a folder to hold all the excel spreadsheets and plots before you do this. Let's say you made a folder called "Output" in your "Data" folder from earlier. Then you'd call

```
> .compareBasicStats(my_data, "~/Desktop/Data/Output/")
```

By default, this step runs a bootstrap power test, t-test, Mann-Whitney test, and bootstrap test on every combination you can think of (compare groups at each timepoints, compare timepoints within each group, compare groups' changes between timepoints). It also outputs box-and-whisker plots of all of this data. *Note: currently power tests are not run for paired comparisons (ie, between timepoints); this will be fixed soon.*

This outputs:

1. For each folder of logs (group/timepoint), an excel file called data_<foldername>.csv that has the counts and durations for each behavior in each log.
2. For each experimental group and pair of timepoints, an excel file called diffs_<groupname>_<timepoint1><timepoint2>.csv giving the difference in counts and durations for each behavior in each log between timepoints (ie count at timepoint2 minus count at timepoint 1).

3. A folder called _groups with
    a. Box and whisker plots for each timepoint of each behavior's count/duration in each group called <timepoint>_plots_<behavior name>.tiff
    b. A spreadsheet for each pair of groups at each timepoint giving the mean, median, and standard deviation for each group at that timepoint as well as the statistical power and the output of stats tests comparing those two groups at that timepoint, called <timepoint>_<group1><group2>_stats.csv
4. A folder called _timepoints with
    a. Box and whisker plots for each group of each behavior's count/duration at each timepoint called <group>_plots_<behavior name>.tiff
    b. A spreadsheet for each group at each pair of timepoints giving the mean, median, and standard deviation for each timepoint in that group as well as the output of stats tests comparing those two timepoints for that group, called <group>_<timepoint1><timepoint2>_stats.csv
5. A folder called _diffs with
    a. A spreadsheet for each pair of groups at each pair of timepoints giving the mean, median, and standard deviation for the differences between the timepoints in each group as well as the statistical power and the output of stats tests comparing those two groups' changes across the two timepoints, called <timepoint1><timepoint2>_<group1><group2>_stats.csv

If you only have one timepoint or one group, not all of these tests will be run – it obviously doesn't make sense to compare groups if there is only one group. If you want to only run some of these tests, you can call

```
> .compareBasicStats(my_data, "~/Desktop/Data/Output/",
comparisons = 'groups')
```

to get parts 1 and 3,

```
> .compareBasicStats(my_data, "~/Desktop/Data/Output/",
comparisons = 'timepoints')
```

to get parts 1, 2, and 4, or

```
> .compareBasicStats(my_data, "~/Desktop/Data/Output/",
comparisons = 'diffs')
```

to get parts 1, 2, and 5.

If you want to use stats tests other than the default t-test, Mann-Whitney, and bootstrap, you can pass in your own tests using the `tests` parameter (for unpaired tests on groups and diffs) and/or the `paired_tests` parameter (for paired tests on timepoints). For example, to use just the t-test and the Mann-Whitney test, you could call

```
> .compareBasicStats(my_data, "~/Desktop/Data/Output/", tests =
list(ttest = t.test, mannwhitney = wilcox.test))
```

The functions you provide must take the data as parameters named `x` and `y`, and must return a list with the p-value as an element called `p.value`. Most unpaired stats tests built into R work this way already. If you need to pass an argument to a function, you can write a wrapper for it; for example, the paired t-test is given as

```
> .pairedTTest = function(x,y) {
+     return(t.test(x,y,paired = TRUE))
}
```

For more information about passing arguments to these passed-in functions, including arguments that are not the same for every comparison to be run, see the documentation for `.runStatsTwoGroups()` in `behavior.R`.

Since latencies are time-censored data, they cannot be compared with the same stats tests. There is currently no function to do paired comparisons of latencies; unpaired comparisons are done by default with a Mantel-Cox test. This can only be run if you gave the length of your assays back when you loaded the data in; if you didn't, the latencies will still be output but the tests will be skipped. To run a different test on latencies, use the parameter `latTests` just like `tests` or `paired_tests`.

You can also compare transitional probabilities with

```
> .compareTransitionalProbabilities(my_data,
"~/Desktop/Data/Output/")
```

and entropies with

```
> .compareEntropy(my_data, "~/Desktop/Data/Output/")
```

These functions have all the same options as `.compareBasicStats()`, and their outputs also take the same form.

Step 6: Input supplemental data (GSI, hormone levels, …)

To compare behavioral metrics (counts, latencies, etc) with other data, you'll need to get that data into R. To do this, put it in a spreadsheet that has one column with the subject ID (a string of characters that appears in the log names for ONLY ONE subject), one column with the experimental condition (if more than one), and additional columns with each other metric (eg GSI or hormone levels) you want to examine. If your subject IDs are different at different timepoints, use the ID from the first timepoint. The first row of the spreadsheet should have labels for each column. You should end up with something like this:

	A	B	C	D
1	ID	Condition	GSI	Days in tank
2	subj1	Control	0.2	4
3	subj2	Control	0.31	5
4	subj3	Control	0.43	3
5	subj4	Experimental	0.74	6
6	subj5	Experimental	0.52	4
7	subj6	Experimental	0.6	5

Save this file as a `.csv` – let's say you call it `gsis.csv` and save it to your Data folder. Then get this data into R by calling

```
> supplementalData <- read.csv('~/Desktop/Data/gsis.csv')
```

Check to make sure this worked:

```
> supplementalData
```

This will print out what got read in to R. It should look like this:

```
     ID    Condition  GSI Days.in.tank
1 subj1      Control 0.20            4
2 subj2      Control 0.31            5
3 subj3      Control 0.43            3
4 subj4 Experimental 0.74            6
5 subj5 Experimental 0.52            4
6 subj6 Experimental 0.60            5
```

Now, we need to match up these data with the actual behavior logs. To do this, call

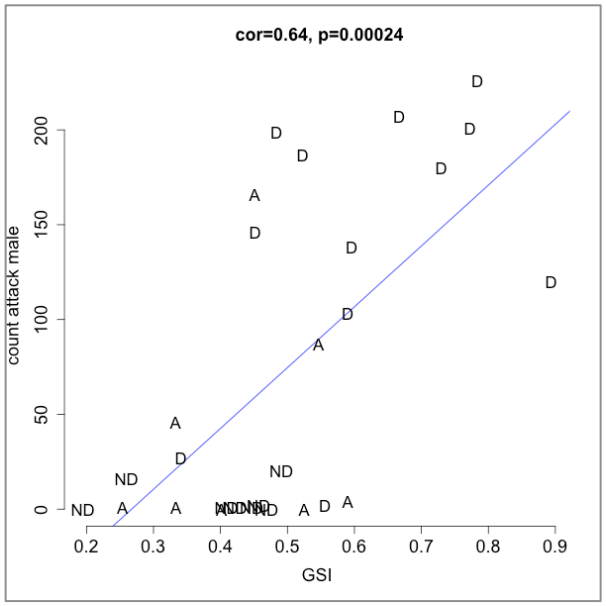
```
> datForCorrelations <- .orderSupplementalData(supplementalData,
my_data, 'ID')
```

To make a boxplot like those output by the compare functions of anything here, you can call the `.prettyBoxplotNoList` function. As an example, to plot GSI by experimental group and save the plot:

```
> .prettyBoxplotNoList(datForCorrelations$GSI,
datForCorrelations$Condition, outfile =
"~/Desktop/Data/Output/gsiplot.tiff")
```

To plot other things, just switch out what column name you put after the `$`. To plot to the active R plotting window without saving, just omit the `outfile` argument.

Step 7: Scatterplots and correlations



This is under development, so there's not a good UI yet. Sorry!

For this step and the next one, there isn't currently code to handle different timepoints. This will get fixed soon; sorry! In the meantime, you can split up your data by timepoint. If you have 2 timepoints, baseline and test:

```
> my_data_baseline <- .dataFromTimepoint(my_data, 'Baseline')
> my_data_test <- .dataFromTimepoint(my_data, 'Test')
```

And then below wherever it says `my_data` just run it once with `my_data_baseline` and once with `my_data_test`, saving to `counts_baseline` and `counts_test`, etc.

First, you'll need to get the behavioral data you want to correlate:

```
> counts <- as.data.frame(t(.extractBasicStats(my_data,
.behnames(my_data), NULL)$counts))
> durations <- as.data.frame(t(.extractBasicStats(my_data,
.behnames(my_data), .startStopBehs(my_data)$durations))
```

Then you can make scatterplots like this! (in this example, correlating GSI with number of quivers)

```
> verboseScatterplot(datForCorrelations$GSI, counts$Quiver)
```

If your behavior name has spaces in it, you need to put weird quotes around it to get it to work. These weird quotes (`) are on the same key on the keyboard as the ~. Example:

```
> verboseScatterplot(datForCorrelations$GSI, counts$`Attack
Female`)
```

You can also correlate behaviors with each other:

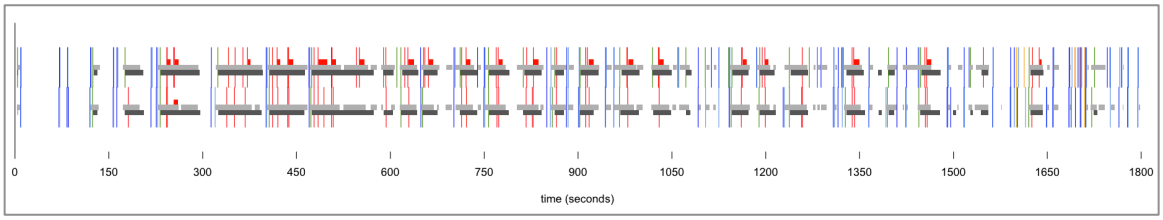
```
> verboseScatterplot(counts$`Attack Male`, counts$`Attack
Female`)
```

If you want to get fancy and have the first letter of each group be the point on the scatterplot instead of just a plain black circle, you can do this:

```
> textScatterplot(datForCorrelations$GSI, counts$`Attack Female`,
letters = substr(datForCorrelations$Condition, 1, 1))
```

For the first two letters, change that second 1 to a 2; for the first three to a 3, etc.

Step 8: Make raster plots



First, you need a color key. To get this, call

```
> my_color_key <- .buildColorKey(.behnames(my_data))
```

and follow the prompts.

Then, call

```
> .makeMulticolorRasterPlots(my_data, "~/Desktop/Data/Output/",
my_color_key)
```

and raster plots should be automatically generated! This makes a separate image of the raster plots for each folder (group/timepoint); to get a single raster plot use

```
> .makeMulticolorRasterPlot(my_data, my_color_key)
```

to output to the R plotting window, or

```
> .makeMulticolorRasterPlot(my_data, my_color_key,
"~/Desktop/Data/Output/rasterplot.jpg")
```

to save it as a file.

There are a ton of options for this function.

1. Graphics options.
    a. `wiggle` controls how much height the durational behavior bars are allowed to take up. It should always be between 0 and 0.5 (default 0.2). Bigger value = thicker bars.
    b. `durationalBehs` is a vector of behaviors that should be plotted as bars instead of tick marks. The default is that all durational behaviors are plotted this way.
    c. `staggerSubjects` if TRUE causes male behaviors to be plotted above the line for a subject and female behaviors to be plotted below the line. (default FALSE)
    d. `widthInInches` and `rowHeightInInches` control the dimensions of the output plot.
    e. `horizontalLines` if TRUE, a horizontal black line is drawn behind the raster plot for each subject. (default FALSE)
    f. `linesBetweenLogs` if TRUE, a horizontal black line is drawn behind the raster plot for each subject. (default FALSE)
    g. `sep` The amount of separation there should be between logs; between 0 and 1. (default 0)
    h. `labelSpace` The interval in seconds between tick marks on the x-axis. (default 150 = 2.5 minutes)
2. Align at the nth occurance of a behavior. For example, the first spawn:

```
> .makeMulticolorRasterPlots(my_data,
"~/Desktop/Data/Output/", my_color_key, zeroBeh =
"spawning")
```

   The fifth quiver:

```
> .makeMulticolorRasterPlots(my_data,
"~/Desktop/Data/Output/", my_color_key, zeroBeh = "quiver",
zeroBeh.n = 5)
```

   The first female-directed behavior:

```
> .makeMulticolorRasterPlots(my_data,
"~/Desktop/Data/Output/", my_color_key, zeroBeh =
c("quiver", "lead", "chase female"))
```

   This option doesn't currently work with `.makeMulticolorRasterPlot()`.
   If you provided an assay length, a grey box will be drawn for each row showing the start and end of the assay.
3. Sort by something. To do this, you need a `sortAttribute`. For example, to sort by number of quivers, you could do

```
> sortAttr <- .getCountAttribute('quiver', my_data)
```

   You can also use the functions `.getLatencyAttribute()`, `.getTotalDurAttribute()`, and `.getAverageDurAttribute()` to sort by a behavior's latency, total duration, or average duration, respectively.

   Alternatively, you can sort by outside data input in step 6. Then, your sort attribute becomes

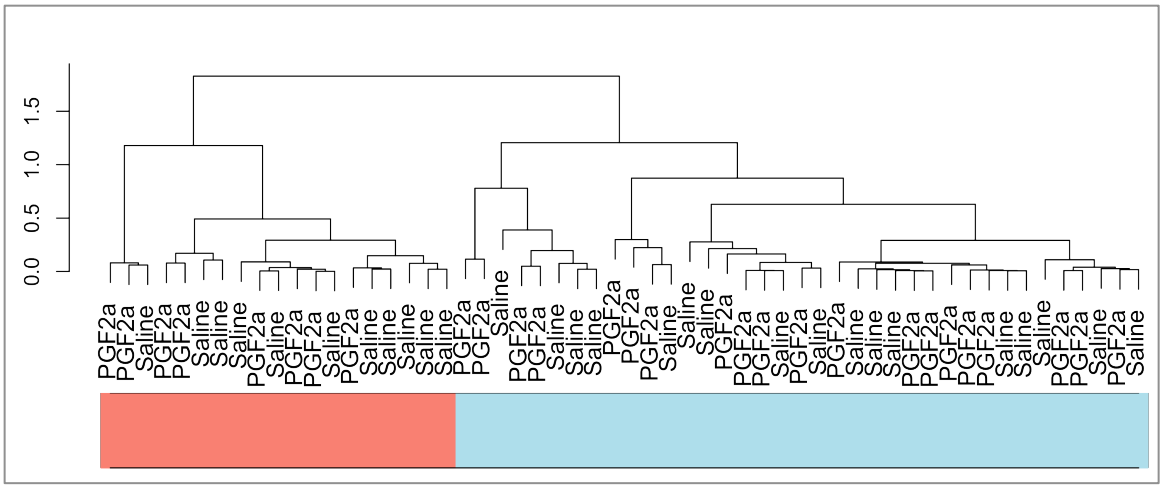
```
> sortAttr <- datForCorrelations$GSI
```

Once you have your `sortAttribute`, call

```
> .makeMulticolorRasterPlots(my_data,
"~/Desktop/Data/Output", my_color_key, sortAttribute =
sortAttr, sort.name = "GSI");
```

`sort.name` should be a description of the attribute you sorted by; another reasonable one would be "Number of Quivers". You can control the order of the sort by adding `sort.decreasing = TRUE` to sort in decreasing order and/or `sort.na.last = FALSE` to put NAs first `or sort.na.last = NA` to exclude NA values of your attribute. (NA means not available; this could happen if you didn't measure GSI for some of your subjects, for example).
This option also doesn't currently work with `.makeMulticolorRasterPlot()`, and it doesn't work if you have more than one timepoint.

Step 9: Clusters



This is just one easy function call!

```
> .clustersEtc(my_data)
```

This clusters logs by counts of behavior. There are several options for this function:

1. `clusterOn` is by default set to "counts", but you can change it to `clusterOn = "transprobs"` to cluster by transitional probabilities instead.
2. `logLabels` is the labels that show up on the plotted dendrogram. To make these be the folder name, try:

```
> .clustersEtc(my_data, logLabels = gsub('^([^/]*)/.*$',
'\\1', names(my_data)))
```

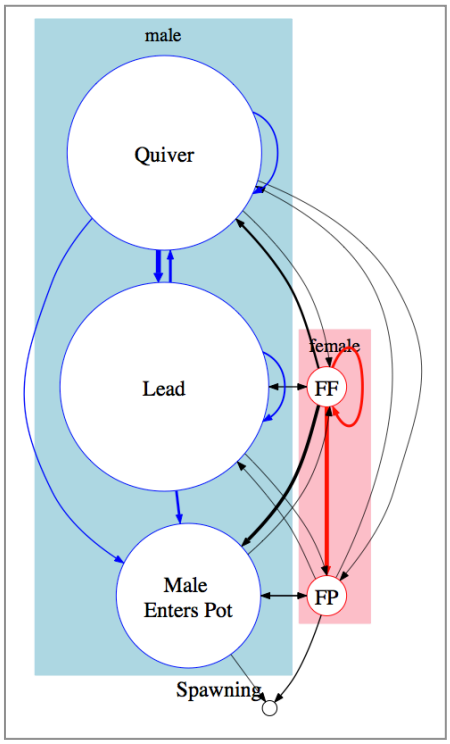
3. If you're having troubles or crashing, try adding `cor.use = 'p'`.
4. You can add `hclust.method = 'average'` to make the clustering algorithm prioritize linkages differently. You'll end up with a similar but slightly-different tree.
5. `cutree.minClusterSize` controls the minimum number of fish that need to be in a cluster for it to count as a cluster. The default value is pretty high; if you end up with all your fish in one cluster, or see something that looks like a cluster but was assigned not-clustered-grey, try setting this parameter to a lower value like 3 or 4. On the other hand, if you have like 8 different clusters try making it higher.

6. You can control the colors of the modules by passing in a list of colors. For example,

```
> .clustersEtc(my_data, colors = c('red', 'darkorange',
'gold'))
```

The first color in the list will be used for the biggest module, the next for the second-biggest, etc. If you don't provide enough colors, the program will just pick its own colors anyway and ignore you, but it will throw a warning to let you know that that's what it's doing.

Step 10: Markov chains



NOTE: Markov chains and branching diagrams are saved as .dot files. You will need to install the free software GraphViz from http://www.graphviz.org/ to view them.

To make a Markov chain for each group at each timepoint, call

```
> .makeGroupDotPlots(my_data, '~/Desktop/Data/Output/markov')
```

In all likelihood, this will be a tangled mess. There are several ways to fix this. You can not plot nodes that you aren't interested in; here's an example from a spawning assay where you're only interested in behaviors that lead up to spawning:

```
> .makeGroupDotPlots(my_data, '~/Desktop/Data/Output/markov',
nodesToExclude = c('bite', 'chase', 'flee', 'male in pot stop',
'female in pot stop', 'spawning stop'))
```

You can also not plot lines that represent transitions less-frequent than a given probability. To only plot transitions with at least 2% likelihood:

```
> .makeGroupDotPlots(my_data, '~/Desktop/Data/Output/markov',
minValForLine = 0.02)
```

You can also draw a Markov chain where the lines are weighted not by transitional probability (if you just did behavior A, how often do you next do behavior B?) but rather by transitional frequency (out of the entire log, how often does the sequence AB show up?). To do this, add the parameter `byTotal = TRUE`. If you do this, you'll also probably want a lower threshold for `minValForLine`.

Another option is to cluster the nodes into different types of behavior. The easiest way to do this is by subject; you'll also need to provide colors for each cluster (arrows) and lightcolors (background). Here's an example where we sort by the subject who did the behavior (male is blue and female is red).

```
> .makeGroupDotPlotsClust(my_data,
'~/Desktop/Data/Output/markov', subjects = c('male', 'female'),
colors = c('blue', 'red'), lightcolors = c('lightblue', 'pink'))
```

`minValForLine` and `nodesToExclude` still work with this function, so you can combine the three any way you choose.

Currently, the only way to sort behaviors into clusters is using the subject column, but you can edit this to force the computer to cluster nodes in a different way. Example:

```
> my_data_categories <- .setSubject(my_data, c('frontal display',
'lateral display', 'border dispute', 'chase male'), 'aggression')
> my_data_categories <- .setSubject(my_data_categories,
c('quiver', 'lead'), 'courtship')
> my_data_categories <- .setSubject(my_data_categories, c('in
pot', 'dig'), 'territory')
> .makeGroupDotPlotsClust(my_data_categories,
'~/Desktop/Data/Output/markov', subjects = c('aggression',
'courtship', 'territory'), colors = c('red', 'blue', 'gold'),
lightcolors = c('pink', 'lightblue', 'lightyellow'),
indivMarkovChains = FALSE)
```

You can also make some fancier Markov chains:

Bouts: To say that any behavior that occurs after 10 seconds of inactivity constitutes a new "bout", call

```
> .makeGroupDotPlots(my_data, '~/Desktop/Data/Output/markov',
boutInterval = 10)
```

Change over assay: To get a Markov chain from 10 minutes into the assay to 20 minutes into the assay, call

```
> .makeGroupDotPlots(my_data, '~/Desktop/Data/Output/markov',
startTime = 10 * 60, endTime = 20 * 60)
```

Ignore ends of behaviors: To eliminate all those pesky transitions from "behavior start" to "behavior end", call

```
> .makeGroupDotPlots(my_data, '~/Desktop/Data/Output/markov',
startOnly = TRUE)
```

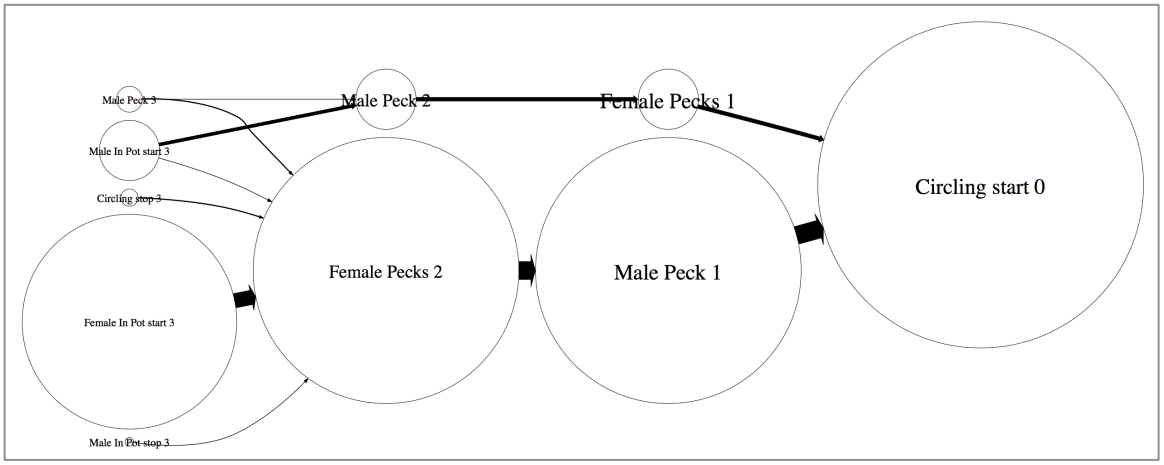
You can also give a list of behaviors to ignore ends of as in the filtering step, eg `startOnly = c("Lead", "Quiver")`.

Single subject: To get a Markov chain of just what the male does, call

```
> .makeGroupDotPlots(my_data, '~/Desktop/Data/Output/markov',
subjects = "male")
```

These options unfortunately do not work with `.makeGroupDotPlotsClust` at this time.

Step 11: Branching diagrams



NOTE: Markov chains and branching diagrams are saved as .dot files. You will need to install the free software GraphViz from http://www.graphviz.org/ to view them.

To make a picture of the three behaviors that come before spawning, call

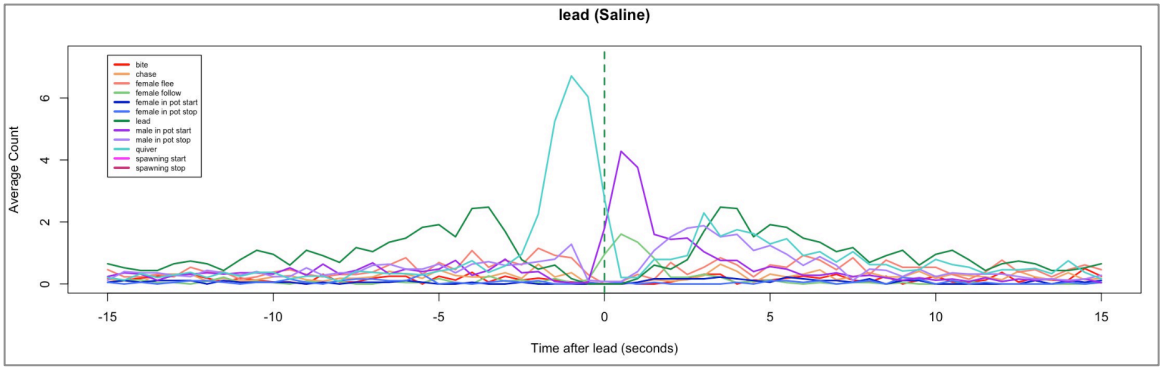
```
> .branchingDiagram(my_data, "spawning", kb = 3, ka = 0, file =
'~/Desktop/Data/Output/spawning.dot')
```

For the three behaviors that come after quivers, call

```
> .branchingDiagram(my_data, "quiver", kb = 0, ka = 3, file =
'~/Desktop/Data/Output/spawning.dot')
```

You can also add the parameters `fontsize` to control the font size, `minTimesToDrawNode` to only draw nodes that happened at least that many times (default 1 = draw all the nodes), and `minLine` to only draw transitions that happened at least that many times (default 1 = draw all the arrows)

Step 12: Behavioral density plots



You might want a different color key from the raster plots for this; in the raster plots you'll often want all courtship behaviors to be similar in hue, whereas for this you want behaviors that occur close together to have maximum contrast. But there's nothing to prevent you from using the same one. If you choose to create a new color key, just follow the directions from the raster plot section.

To make behavioral density graphs:

```
> .behavioralDensityGraphs(my_data, my_color_key, filePref =
'~/Desktop/Data/Output/')
```

If you only want graphs centered at a few behaviors (say "lead" and "female enters pot"), use the parameter `targetBehs`:

```
> .behavioralDensityGraphs(my_data, my_color_key, filePref =
'~/Desktop/Data/Output/', targetBehs = c('lead', 'female enters
pot'))
```

The parameter `weightingStyle` controls which actual y-axis values are plotted. It must be either "density", "singlebeh", "allbeh", "centerbeh", or "rawcounts"). If it is "density" (default), the output of built-in function `density()` is plotted; you can additionally specify `densityBW` (default 1) which is the bandwidth in seconds of the density calculations, and `densityN` (default 512), which is basically the resolution.

All the other values of this parameter yield histograms, with behaviors binned together in timebins of width `secondsPerBin` (default 0.5). The exact value of `weightingStyle` controls what the counts in each bin are divided by to normalize them ("rawcounts" yields no normalization). "allbeh" divides by the total number of behaviors for that subject, "singlebeh" divides by the count in that subject's log of the behavior the line represents, and "centerbeh" divides by the count in that subject's log of the behavior the graph is centered around.

Additional parameters:

1. `lim` (default 15) the number of seconds before and after the center behavior that should be plotted.
2. `lineWidth` (default 2) the line width in pixels
3. `lineType` (default "solid") the style of line that should be drawn
4. `ymax` (default automatically calculated for each plot) the height of the y-axis

To not plot a behavior, just remove it from your colorkey. You can do this with

```
> editedColorKey <- .editColorKey(my_color_key,
.behnames(my_data))
```

Step 13: Send feedback

What did you like? What did you hate? What do you wish you could do, or could do more easily? What took you a long time to figure out how to do? What crashed? What had a typo? What was annoying and fiddly? What did you think was ugly?

Please, please, please send feedback to kkent17@stanford.edu!

How to Do a Helpful Bug Report

Run the thing that crashes.

Call

```
> traceback()
```

Call

```
> warnings()
```

Copy paste from the function call you made that threw the error all the way down to the end of what printed out from `warnings()`.

For bonus points, make a list of all the variables that were in your function call. This is anything that is not a number, is not right before an equals sign, and is not in quotes. For example, if the call was

```
> .behavioralDensityGraphs(my_data, my_color_key, filePref =
'~/Desktop/Data/Output/', targetBehs = c('lead', 'female enters
pot'))
```

the variables are `my_data` and `my_color_key`. `filePref` and `targetBehs` are before equals signs, and `~/Desktop/Data/Output/`, `lead`, and `female enters pot` are in quotes, so they don't count. Save these to a file:

```
> dump(c("my_data", "my_color_key"), file =
'~/Desktop/bugReportData.R')
```

Send me all of this in an email, and I'll deal with it as soon as I can! The more information you provide, the more likely I am to be able to figure out exactly what went wrong.