# Tobii Eyetracker Data Analysis

Brock Ferguson
Psychology, Northwestern University

Updated July 17, 2012

# Table of Contents

# Introduction

The relatively recent availability of commercial eyetracking hardware has given cognitive psychologists the ability to observe gaze behaviour with a higher temporal and spatial resolution than ever before (Aslin, 2011). However, making sense of the massive datasets generated by this new equipment has proven to be both controversial and time consuming.

This guide will serve as a walkthrough for analysing a typical two-alternative forced choice (2AFC) looking paradigm, introducing several possible statistical analyses suitable for eyetracking data in the context of a custom R library. It assumes that you are equipped with the following:

(1) Tobii X60 Eyetracker

(2) R v2.11 or greater

(3) `process_tobii_dataIbf.R` library

(4) `analyze_tobii_data_bf.R` library

(5) phase timing and AOI files (see examples included with libraries)

R is available for Windows or Mac environments free at r-project.org. The two R libraries are available from Brock.

We will begin this walkthrough after data has been exported from the data eyetracker. Each participant's data should be in a uniquely named data file (e.g., ANCAT-Informative-ANCAT60.tsv) exported using the standard settings on the Tobii eyetracker. Place each of these files along with the two referenced R libraries in a folder for the analysis. From here, we can begin our data preparation.

# Data Preparation

Tobii exports individual files containing time course data for each of our participants in a study. These are raw data and not very useful. However, we will make them useful by cleaning and aggregating their data so that we can perform statistical analyses and plot these data in a meaningful way.

Our first step will be to create a "master file" which will house all of our subject data merged together in one big file. Moreover, we will be able to merge/create new columns of information in this master file which will be useful in most future analyses. These new columns may represent binomial data regarding looking to/away from areas of interest (AOIs), subject data such as age or vocabulary information, and individual trial factors.

After our master file is prepared, we will examine the data lost to "trackloss" (when the location of the infant's eye gaze is unable to be determined) and create a "clean file" where individual data points or entire trials are removed due to trackloss. This will essentially serve as our new master file for analyses, but of course we keep the old master file for its raw data.

Finally, from this clean file, we will create a "test file". This smaller dataset will include only the datapoints that we want to analyze and look for differences between our conditions. Most commonly, the test file subsets the clean file based on two criteria: (1) trial names (including only relevant trials and not training trials, etc.) and, (2) a relevant time window. This test file will be fed to most of our analysis functions and plotted for the sake of presentations and publications.

## The Master File

As stated above, our master file serves to merge all of our participant's raw Tobii data files together, as well as add new columns to this data representing looking within AOIs, subject data, or trial data. Most of this data will be merged through the use of two methods - `auto_process_tobii()` and `prepare_master()` - however some data may take custom R scripting. Both of these methods will be detailed here.

Let us first use the `auto_process_tobii()` method to add three new columns to each individual participant's data file: SceneName, Position, and SceneType. These three columns can represent different things in a study but they essentially represent three types of information related to an AOI. The information related to AOIs and the timing of their appearance on the screen

during a trial should be located in `aoi-file.txt` and `phase-timing.txt` (in the working directory). This walkthrough is noticeably scant on details regarding these files, but simply use existing files as a template and they should be fine. Please note that even if you have several conditions in the experiment with unique trial timings or AOIs, these can all be included in one AOI file and phase timing file provided you give the respective trials unique names that relate to the unique movie names used in the Tobii eyetracker. The order of the information in these files is not important.

So, returning to our initial processing of participant data files, we need only to enter a few commands into R.

```
> setwd("~/Documents/Studies/MyStudy/Analyses/July 17, 2012/")
> source('process_tobii_data_bf.R')
> auto_process_tobii('aoi-file.txt','phase-timing.txt')
```

The first line sets the working directory of R to the folder that we created for this analysis. This allows the `auto_process_tobii()` function to automatically find our participant data files.

The second line loads our Tobii processing R library into the workspace so that we can access its methods.

Finally, the third line calls our function and passes to it the filenames of our AOI file and phase timing file, also located in this working directory. If you wanted to run this function several times (perhaps once for each condition, using unique AOI files), you can pass optional 3rd and 4th arguments to it specifying the condition and order to process. Only participants within this condition and/or order will be processed by the function. For example:

```
> auto_process_tobii('aoi-file-verb.txt','phase-timing-
verb.txt','VERB')
> auto_process_tobii('aoi-file-noun.txt','phase-timing-
noun.txt','NOUN')
```

In the above example, we will process the participants in the two conditions separately.

When we are done processing, we will have new files named `processed-*.csv` in our working directory, where the wildcard `*` represents the participant number. This number will automatically be retrieved from the ParticipantName in the raw Tobii data file.

Now, it's time to merge all of our individual, processed subject files into the first version of our master file. We will do this using one simple function, `concat_csv()`, like so:

```
> concat_csv('master.csv')
```

This function will automatically find and import all of our `processed-*.csv` files and concatenate them into one large CSV master file named `master.csv`. In doing so, it will retain only the column headers from the first file.

Next, we will process this master file to merge in general subject and trial data, as well as create columns that represent looks within our AOIs using `prepare_master()`. These AOI columns are automatically generated from the SceneType values specified in our AOI file. For each datapoint in which this SceneType is activated (with a look in the AOI), this new column will be populated with a "1". For all datapoints where another SceneType is active, this column will be populated with a "0". Finally, for datapoints lost to trackloss, these new columns will have a "NA" value.

In order to merge subject data into our master file, we will create an optional CSV file (e.g., `subjects.csv`) and pass this filename to the function. This file has only one mandatory column, ParticipantName. The values in this column are matched to the corresponding values in the column of our master file of the same name so that the proper data is merged row-by-row. For example, our subjects file may look something like:

```
ParticipantName,Sex,Age,MCDI_Noun,MCDI_Verb,MCDI_Total
ANCAT1,M,19.53,20,15,35
ANCAT2,F,19.21,9,4,13
ANCAT3,M,20.32,24,5,29
```

These values will merge with our existing master file in new columns. The values must be strings and the column names must not contain spaces or other atypical characters (underscores are allowed).

Likewise, we can create additional columns merged by the Trial column. For example, if I have a study looking at word learning, I may have some words that are nouns and some that are verbs. I want to compare these within-subjects trial conditions in a later analysis, so I merge in trial information in a trial file called `trials.csv`. This file may look like:

```
Trial,GrammaticalCategory
Bottle,Noun
```

```
Car,Noun
Wash,Verb
Run,Verb
```

Each trial name is specified in the phase timing file and used here to merge information beyond that specified in the phase timing and AOI files.

When we bring all of this additional data together, we get a call such as this:

```
> prepare_master(subjectfile = 'subjects.csv', trialfile =
'trials.csv', inputfile = 'master.csv', outputfile = 'master-
prepared.csv')
```

However, given that we do not need to merge subject or trial information, and the `inputfile` and `outputfile` parameters default to the values above, this call *could* - for some studies - be as simple as:

```
> prepare_master()
```

For most studies, we now have a finished master file named `master-prepared.csv`! However, for other studies, we may have additional information that we need to add to this master file. For example, imagine a study in which we are teaching novel names for animals to a child. We assume that most 20-month-olds won't know the word "rhinocerous", however some parents report that their child knows this word. We want to add another column/factor to the data called KnownAnimal to track these instances. We can do so with a simple script like this:

```
> data <- read.csv('master-prepared.csv')
> data$KnownAnimal <- 0
> data[which(data$ParticipantName == 'ANCAT34' & data$Trial ==
'Rhinocerous'), 'KnownAnimal'] <- 1
> data[which(data$ParticipantName == 'ANCAT17' & data$Trial ==
'Tiger'), 'KnownAnimal'] <- 1
> write.csv(data, 'master-prepared.csv')
```

This script is obviously an oversimplification, but it illustrates the point nicely. In order to add additional columns of data to our file, we need only to import the dataset into R (line 1), add a new column (line 2), and insert values into this column based on the values of other columns (lines 3 and 4). Finally, we can write the updated dataset to the original file (line 5).

*Now*, all of our data is merged and we have prepared our raw master data file! Our next step is to take a look at the trackloss in this file and begin cleaning it up.

## Trackloss & The Clean File

Compiling our raw master file (`master-prepared.csv`) was rather straightforward. We followed a series of steps in cookbook fashion and compiled *all* of our data into one file, regardless of quality. However, compiling our clean master file will be the first point in which we begin to make critical theoretical decisions in how we analyze the data.

Before we begin making these decisions, we will simply duplicate and rename our raw master file to `master-clean.csv` so that we keep our raw data file safe from any changes we make going forward (should we choose to reverse them):

```
> file.copy('master-prepared.csv','master-clean.csv')
```

We will also load our analysis library for access to our custom R functions:

```
> source('analyze_tobii_data_bf.R')
```

Our first order of business will be cleaning our file by movie name. In a typical study, we will of course have our experimental trials (aptly named in our phase timing file). However, we will also have several non-experimental trials in the dataset relating to calibration trials, training trials, or datapoints that lack a trial name (due to Tobii data spillover before/after a movie plays). We want to remove these from our dataset. We remove these by *movie* name and not *trial* name because, if left out of our phase timing files, our trials will continue across interstitial movies such as attention getters.

Let's retrieve a list of all the movie names in our dataset:

```
> movies <- get_movie_names('master-clean.csv')
> movies
```

From this list in `movies`, we can remove the movies we don't want using a series of calls in the format of:

```
> movies <- movies[which(movies != 'baby_xvid.avi')]
```

Alternatively, instead of removing the movies we *don't* want, we can add only the ones we *do* want, like so:

```
# print the unique names of movies in the file
> movies

# create a list of all the movies we do want, using the list above as a
reference
> movies <- c('movie1','movie2','movie4','movie99')
```

Either way, we end up with a list of the movies we want to analyze in our `movies` object.

After running a command like the above for each movie you want to remove from the list, we can issue a command to clean the master dataset, leaving only the movies we *do* want in the data. This command references the `movies` object we have been pruning.

```
> clean_by_movies('master-clean.csv',movies)
```

Our new clean dataset is automatically written to our clean-file-in-preparation at `master-clean.csv`.

It is now time to examine and deal with trackloss. Trackloss occurs when the eyetracker does not know where the eyes are looking on the screen. They may in fact not be looking at the screen. We can also choose to treat looks outside of our AOIs as equivalent to trackloss, and thus only include looks to AOIs in our analysis. This makes sense in a 2AFC design where we are pitting looks to one AOI versus looks to another. In order to treat non-AOI looks as trackloss, run the following command:

```
> treat_outside_looks_as_trackloss('master-clean.csv')
```

In order to deal with trackloss, we'll start by gathering data with `get_trackloss_data()` so we can make some informed decisions:

```
> trackloss <- get_trackloss_data(file = 'master-clean.csv',
window_start = 1000, window_end = 5000)
```

This command's arguments are rather straightforward. First, we pass it our clean file as `file` so that it can be read and analyzed. Second, we will pass it an optional `window_start` (in ms) parameter which will trim out all data prior to this window (e.g., the beginning of the test phase of our experiment). And third, we

will pass an optional `window_end` parameter which will trim all data following this time (again, in ms). Either one or both of these window parameters can be omitted. For example, any of the following calls are valid:

```
> trackloss <- get_trackloss_data(file = 'master-clean.csv',
window_start = 1000)
> trackloss <- get_trackloss_data(file = 'master-clean.csv', window_end
= 1000)
> trackloss <- get_trackloss_data(file = 'master-clean.csv')
```

However, it is my opinion that these window parameters should be used if we are going to perform our analyses on a given test looking window.

This command will return two objects: `$trackloss` and `$trials_committed`. The first will contain trackloss information by participants and trials so that we can see general trackloss trends. We can look at these datasets with:

```
> trackloss$trackloss
> trackloss$trials_committed
```

We can also quantify these trends by looking at the mean number of frames (datapoints) committed per trial, the mean proportion of good data points to all data points, and the relevant standard deviations:

```
> mean(trackloss$trackloss$FramesLooking) # count good non-trackloss
frames
> sd(trackloss$trackloss$FramesLooking)
> mean(trackloss$trackloss$TracklossProportion) # mean proportion of
good frames
> sd(trackloss$trackloss$TracklossProportion)
```

These data will help us decide the maximum amount of trackloss we want to include in our dataset for a single trial. For example, if we have a mean of 250 frames committed per trial and a standard deviation of 75 frames, we can conservatively exclude all trials that committed less than 100 frames of good data (i.e., 2 standard deviations less than our mean).

Importantly, these data do not reflect trials in which zero frames were tracked. For example, in an experiment with 6 trials, we may only have FramesLooking data for 5 trials for a given baby if they fussed out for the 6$^{th}$ trial or looked away from the screen through the entire looking window. To see how

many trials we lost to fuss outs or total non-looking, we can load the second object returned by `get_trackloss_data()`:

```
> trackloss$trials_committed
> mean(trackloss$trials_committed$Trials)
> sd(trackloss$trials_committed$Trials)
```

These will be data that we may want to report in a paper or presentation. They also verify that our clean file (which we trimmed by movie names) hasn't been accidentally *over*-cleaned.

Returning to our trackloss numbers, let's decide what we will do to prepare our dataset for analysis. If we do want to trim trials by their proportion of trackloss, we simply need to decide our minimum criterion for the number of good frames in a given trial. A conservative measure may be 2 standard deviations less than the mean number of good frames. This can be calculated with:

```
> minimum_frames <- mean(trackloss$trackloss$FramesLooking) -
(2*sd(trackloss$trackloss$FramesLooking))
```

Above, we set `minimum_frames` to a potential cutoff value (2 standard deviations less than the mean).

We can then issue the following call to trim all trials in which this minimum frames threshold was not reached:

```
> clean_by_trackloss(file = 'master-clean.csv', window_start = 1000,
window_end = 5000, minimum_frames = minimum_frames)
```

Once again, if we have a particular looking window, we should pass our `window_start` and `window_end` arguments. We also pass our `minimum_frames` criterion and, of course, our master clean file. Our trials are automatically dropped from the master file.

After cleaning by trackloss, we can generate some final numbers regarding how many trials were committed by each participant:

```
> trial_counts <- final_trial_counts(file = 'master-clean.csv',
window_start = 1000, window_end = 5000)
```

And we can look at these data and quantify them for reporting:

```
> trial_counts
```

```
> mean(trial_counts$Trials)
> sd(trial_counts$Trials)
```

This concludes our (optional) process of trimming by total trackloss within a given trial.

However, we still have another trackloss-related decision to make. Do we want to keep *individual* data points lost to trackloss? Or do we want to remove them? To keep these trackloss datapoints is essentially to perform all of our future analyses on the proportion of looking to one target out of all total looking (on and off the screen). To remove these trackloss datapoints is to perform all of our future analyses on the proportion of looking to one target out of looking towards all targets (i.e., removing looks away from the screen from the proportion calculation). I favour the latter in 2AFC designs, however in designs in which habituation or looks away may be a factor, you may want to keep these datapoints.

Executing your decision regarding individual trackloss is as easy running *one* of the following two commands:

```
> remove_trackloss('master-clean.csv')
# or
> keep_trackloss('master-clean.csv')
```

`remove_trackloss()` will remove all datapoints where trackloss has a value of 1, and `keep_trackloss()` will set all of our column values as 0 (and thus include them as non-looks in a proportion calculation).

We have now dealt with trackloss and have a clean master file comprising looks across entire trials. Next, we will create our final dataset file - the "test file" - that will subset this master clean file by a looking window.

## The Test File

We now have a clean master file that contains data from time 0 of every trial through to the end of the trial. We may, however, have a particular looking window in which we want to focus our analyses. If we do have a window, its start and end times were probably already used in calculating trackloss percentages and cleaning by trackloss. We will now use this window one last time in subsetting our master clean file so that we can run our analyses.

If you do not have a looking window in mind and want to look for differences across the entire trial, you can skip this entire section after running this command to duplicate our clean file with the appropriate name:

```
> file.copy('master-clean.csv','master-test.csv')
```

Now non-windowers can skip to the Descriptive Statistics section.

If we do want to focus on a particular window, we need only to execute one command:

```
> create_test_file(input_file = 'master-test.csv', output_file =
'master-test.csv', window_start = 1000, window_end = 5000)
```

The `window_start` and `window_end` parameters are once again specified in time (ms).

Congratulations! Our data is now completely prepared for analysis. We have 3 files: `master-prepared.csv`, `master-clean.csv`, and `master-test.csv`, each subsetting and cleaning its predecessor. From this point forward, we will use `master-test.csv` as our primary dataset as we forge on through gathering descriptive statistics and performing our analyses.

## Descriptive Statistics

With our clean test dataset, we are ready to take the first look at our results by using simple descriptive statistics. Our custom Tobii analysis library provides us with tools to do this.

But first, let's load our data into R as a dataframe. **Having your dataset loaded into R will be a requirement from this point forward in the manual**. Simply run this command to do so:

```
> data <- read.csv('master-test.csv')
```

We now have a `data` object that will be passed to *most* future R library methods as we peruse and analyse our dataset.

Before continuing, however, I should mention that these "magic" functions in our Tobii library can of course be ignored and you can use your own R code to look at that data, like so:

```
> mean(data[which(data$Condition == 'Informative'), 'Animate'])
> sd(data[which(data$Condition == 'Informative'), 'Animate')
```

These commands directly access our `data` object and retrieve statistics. They also subset the data object using simple `which()` function calls that match column values to specified values.

Now, let's walk through some functions that may speed up this process.

### Single DV Descriptives by Condition

In most studies, we care most about the differences in looking between two experimental conditions. So, our R library offers a quick method that will calculate the mean values of a single dependent measure (i.e., column) by condition. This method, describe_dv_by_condition(), takes two objects. The first is the name of our data object (likely `data`) and the second is the name of our column that we want to describe. For example:

```
> describe_dv_by_condition(data, 'Animate')
```

In this example, we want to see the proportion looking towards our animate object in the test dataset by each level of the Condition column in our dataset.

Because we specified "Animate" SceneTypes in our AOI file, Animate is now a column in our dataset.

Our output for this call will be somewhat messy but will include all the statistics we need (just ignore the junk at the beginning):

```
DV: Animate
========================================================================
 X1: Informative
 X2: F
 [Mean]: 0.681  [SD]:   0.466   [Var]:  0.217   [Min]:  0   [Max]:  1
------------------------------------------------------------------------
 X1: Uninformative
 X2: F
 [Mean]: 0.586   [SD]:   0.493   [Var]:  0.243   [Min]:  0   [Max]:  1
------------------------------------------------------------------------
```

## Single DV Descriptives by Multiple Factors

Besides doing a simple analysis by each level of Condition, we may want to look at the interaction between multiple levels of more than one factor. For example, we may want to know what happens within each Sex nested under Condition. Again, each of these factors represents a column in our `data` object. We'll use a simple call passing multiple factors:

```
> describe_dv_by_factors(data, 'Animate', c('Condition','Sex'))
```

This time, our dependent measure (the second argument above) will be examined as a function of each level of each of these factors.

## Multiple DV Descriptives by Multiple Factors

Finally, we may want to look at multiple dependent variables by multiple factors. For this, we simply pass multiple column names in the second parameter of the function above, like so:

```
> describe_dv_by_factors(data, c('Animate','PupilLeft','PupilRight'),
c('Condition','Sex')
```

These are simple ways to get a quick look at the data and see if it (1) corresponds with your intuitions and, (2) leads you to believe that there may be some theoretically important trends in the dataset. The latter will inform further

analyses. Even more informative than these descriptives, however, will be *looking* at our data with the plotting functions described next.

## Plotting the Data

Inferential statistics exist to tell us which differences we observe are *real* (i.e., which may generalize to a population). And they do a good job at that. However, we should begin every analysis by plotting our data and using another valuable experimental tool - the human eye - to verify that the data distributes as it should, and to look for quantifiable trends in the data. Luckily, this analysis toolkit has some simple functions that will help us rapidly plot eyetracking data. The plotting function we use (`plot_data`) is a wrapper for a slightly modified version of the spaghetti plot code released by Jaeger (2012).

Prior to plotting, the data is aggregated by subject across all trials so that the error bars are calculated from subject means and not trial means.

Let's dive in with the simplest form of our graphing function:

```
> data <- read.csv('master-test.csv')
> plot_data(data, dv = 'Animate', factor = 'Condition')
```

The call above passes three simple arguments to `plot_data()` and a new graph (called `graph.png`) is placed in our R working directory. First, the data we want to plot is loaded from a CSV file (line 1). This will most likely be the data for our test window as we can see here. However, we may be interested in plotting data aggregated across the entire trial, in which case we may load `master-clean.csv` instead. In line 2, we pass this data object to `plot_data()`, along with the name of our dependent variable as `dv`, and the name of the `factor` whose levels will comprise the lines on the graph.

We can now look at a more complex version of this call in order to break down the advanced arguments we can pass to this function:

```
> data <- read.csv('master-test.csv')
> plot_data(data, output_file = 'graph.png', dv = 'Animate', factor =
'Condition', title = 'Looking to the Animate Stimulus', y_title =
'Proportion Looking to the Animate', x_title = 'Time (ms)', type =
'empirical', vertical_lines = c(867), bin_size = 100, x_gap = 500,
width = 1000, height = 600)
```

The call above includes all possible parameters. The `data`, `dv`, and `factor` arguments are the same as our first call. However, the rest are new. `output_file` specifies the filename for the new graph to be named in our R working directory. `title`, `y_title`, and `x_title` specify the main title, Y axis title, and X axis title

respectively. `type` can be either "empirical" or "smoothed" depending on the type of graph we would like. Smoothed graphs are lines of best fit and include 95% confidence margins. Empirical graphs are rigid points plotted from data with standard error bars. The `vertical_lines` argument can include one (e.g., `c(867)`) or several (e.g., `c(867, 1500)`) points in time (ms) that should be marked with a black vertical line on the plot. These are useful for marking events of interest. `bin_size` specifies the size (in time, ms) by which to bin the looking data. `x_gap` specifies the interval between x-axis time markings at the bottom of the graph, and `width` and `height` specify the absolute dimensions of out the graph image. All of these additional arguments function independently so you can add one or all of them to the simplest version of the `plot_data()` call in order to customize your graph.

## Time-based Analyses

One of the major benefits of modern eyetracking technology over manual coding of videos is the temporal resolution. We have the ability to look precisely at cognitive processes unfolding over time instead of collapsing across time and comparing total proportion values. This guide will introduce a function in the R library that will prepare datasets that are ready for time-based analyses. Moreover, it will introduce the theoretical motivations for these analyses, as well as some advice for modelling your data.

While initially this section included a magic all-in-one function that prepared the data, ran the statistical analyses, and reported its results, I have since dropped this in favour of a simpler function, but one that requires more work and thought from the researcher in how they want to analyze their data. Let's begin, though, with a look at what this function does.

### Preparing the Data

The analysis library includes a method, `time_analysis()`, which takes any specified test dataset (e.g., that `data` object we loaded at the beginning of our analysis) and return three datasets: one aggregated across trials by subjects, one aggregated across subjects by trials, and one dataset that crosses subjects and trials (only binning by time).

Here's a look at how we call the function:

```
> time_data <- time_analysis(data, dv = 'Animate', factors =
c('Condition','KnownAnimal'))
```

We pass as factors the main categorical predictors that we will use for the analysis. Unfortunately, we cannot use covariates like age because we bin across both *subjects* and *items* for time analyses and there is no way to relate something like age to items. If you want to control for these sort of variables, or partial out their variance, you can do that with a window analysis (see that section).

We now have our 3 datasets stored within the `time_data` list object. So, we will extract them so that we have 3 new dataframes in our R workspace:

```
> time_bySubj <- time_data$bySubj
> time_byItem <- time_data$byItem
> time_crossed <- time_data$crossed
```

With the dataframes extracted, we are ready for analysis. But first, let's consider some of the choices we'll have to make when we fit our models to these datasets.

## Dependencies in Eye Gaze Measurements

Most popular statistical analyses are adaptations of the linear model (y = mx + b). The Analysis of Variance (ANOVA) approach is perhaps the most popular form of linear regression in experimental psychology. The ANOVA is brilliant in its simplicity and its interpretability. It also makes few assumptions about the data entered into the analysis. However, of the few assumptions it makes, proportional data (the data we get in a 2AFC paradigm) breaks these assumptions and thus can lead to increased Type I and Type II errors in a typical ANOVA analysis.

Modern eye trackers can record someone's eye gaze at a rate of 1000 or even 2000 measurements per second. Even our Tobii samples at 60 times per second. This results in a sampling every 16.66667ms. This is great for temporal resolution, but is actually faster than a typical saccade, which may take 200ms to plan for an unexpected stimulus and last 20-200ms each. This increases the likelihood that any two or three sequential samples will be the same, and thus decreases the independence between our observations. The independence assumption is a hallmark of any regression-based analysis, and we would break it if we didn't work a little more with our dataset.

We aggregate our data into 50ms bins (3 frames, for us) to reduce the dependence between any two samples. We also aggregate across trial observations by Items (i.e., trials) and by Subjects (creating two unique datasets) to further reduce the dependence between samples. The Items dataset will have one datapoint per 50ms bin per trial. The Subjects dataset will have one datapoint per 50ms bin per subject. These aggregation techniques are alternatives to other methods of reducing dependence such as using robust standard errors (Barr, 2008). They do lose a little power (because we are losing datapoints) but are a nice compromise for our typically smaller datasets.

This aggregation happens automatically in this function but it's something that the researcher should understand and be aware of.

## The Logistics of Logistics

Beyond the assumption of independence, eyetracking data also violates the assumption of linearity that underlies ANOVA as a form of linear regression.

In linear data, an equal measured difference between any two points on the curve represents the same difference in the underlying latent ability. Consider weight, for example. Someone who is 150lbs is 10lbs heavier than someone who is 140lbs, and two times heavier than someone who is 75lbs. These are basic features of a ratio scale. In contrast, consider scores on a test out of 25. Is the difference between a score of 17 and a score of 12 equal to the difference between a score of 24 and 19? Perhaps intuitively, these are equivalent differences. These intuitions are wrong. The fact that the scale is bounded by 0 and 25 means that smaller measured differences towards either end of the scale are equivalent to larger measured differences in the middle of the scale. This strange fact of math is better known, and perhaps better understood, by the phenomena named *ceiling effects* or *floor effects*.

Proportional data is more like a school test than weight. Proportions are bounded by 0 and 1 and, despite it being hard to imagine, there's no reason why someone couldn't be 400, 500, 600, or 700lbs. Weight is, for our sakes, unbounded. Crucially, the underlying latent ability in any of these scenarios is unbounded. Skill in math is unbounded, and so are the language and cognitive abilities we measure in our experiments. So, while the measurements asymptote at 0 and 1, the latent abilities continue on with each greater/lesser score representing more of a difference in the latent ability.

This gives us a dataset that is not represented by the linear curve, but rather a logistic curve. The ANOVA's assumption of a linear curve no longer fits our data, because the ANOVA will assume that the distributions underlying our experimental conditions will both normally distribute. This may result in ANOVA assuming that our proportion of 0.87 has a confidence interval from .60 to 1.14. But knowing that our scores are bounded by 1 tells us that the ANOVA is underestimating the likelihood that our proportion could be lower, closer to the mean of .50. However, it also overestimates the variance around a .87 mean proportion. The variance estimates of proportions actually decrease towards either 1 or 0 in a logistic curve. The ANOVA doesn't capture this, either.

Thankfully, the Generalized Linear Model (GLM) takes the same principles of linear regression and generalized them to other data models, one being the logistic model. Logistic regression with the GLM still has underlying linear regression with its predictors but *links* it to the dependent variable using a logit function. Now, we know our probability estimates and predictions are in line with the logistic form of our dataset.

Researchers have adopted logistic regression. We could too, and we could feed each of our eye gaze samplings with either a "1" or a "0" in our dependent variable into the model and see if our factors accurately predicted performance in our dependent variable. However, because of our initial aggregation to reduce dependence between our samples, we must use one of two approximations of the logistic curve. Depending on what we choose, we will be entering different syntax into the model we specify to analyse the data.

Our first option is to follow Barr and use the empirical logit transformation (Barr, 2008). This takes the number of frames looking in a bin and the number of total frames in a bin and calculates a transformed value which can serve as our dependent variable. Notably, it stretches the proportions above and below 0 and 1 and therefore rids us of the boundedness that violated linear regression's assumption. Furthermore, the empirical logit transformation is usually combined with weighted regression (Barr, 2008; McCullagh & Nelder, 1989) so that we can prioritize bins that have more datapoints in them when fitting a model to the data. Empirical logit values and their weights are automatically computed in the datasets returned by `time_analysis()`.

We can also use the arcsin-root transformation to linear*ize* our proportional data. This transformation takes our bin means and applies the function `asin(sqrt())` to them. These values are also calculated in our dataframe automatically.

I personally recommend a weighted empirical logit regression because it follows Barr (2008) and the weighting can help us clean infant looking data which will be fully of frames lost to trackloss.

See Jaeger (2008) for a much more thorough review of why ANOVA's and linear regression are bad for proportional and categorical data.

## Time as a Linear or Polynomial Function

In a basic time analysis, we look to see linear change over time. In essence, as each 50ms bin ticks away, is there a positive or negative slope change between our factors? They may move together over time, or away over time. For this, we can use our `t1` factor in the datasets to represent a main effect of time and an interaction between time and any of our other factors.

Often, however, the differences between factors may not be captured in linear fashion. For example, both groups may exponentially increase their

looking towards a target over time. These data would fit a quadratic curve. However, importantly, one may fit the quadratic curve differently than the other (representing a steeper/shallower curve based on the parameter estimates). For a quadratic curve, we would use our `t2` factor in the dataset (a natural polynomial).

It may also be the case that our groups fit *different* types of curves. For example, one may fit a quadratic curve and the other a cubic curve. Or, maybe we don't have any idea about what kind of curves they fit and we want to compare their fit to several options. For this, we can use our orthogonal polynomials (Mirman, 2008) and see if either one correlates significantly more with one level of our factor than another (suggesting differential fits). For thus, we can use our orthogonal polynomial factors in each dataset: `ot1`, `ot2`, `ot3`, and `ot4`. Orthogonal polynomials are better than natural polynomials when we want to use them simultaneously because they do not correlate with each other. This independence allows us to include them all in a simultaneous test without one correlation affecting the strength of another. This contrasts natural covariates such as age and intelligence, where both factors take a piece of the same variance.

We can try fitting each of these factors in our model along with our experimental factors and see what happens. Or, we can let the data choose which factors might be worth investigating and fit the grand mean of our data by all 4 orthogonal polynomials and seeing which have a significant fit.

## Correlated & Uncorrelated Random Effects

Most random effect terms are specified such that they may be correlated with each other, like `(1 + Condition + Word | ParticipantName)`. This terms says that the intercept, slope of condition, and slope of word may all vary by participant, and that the way the vary may correlate with each other (e.g., someone who is reactive to condition may also be highly reactive to word type).

However, for some factors, this may not be the case and we may want to fit these random slopes are uncorrelated. In the example above, we would specify like `(1 | ParticipantName) + (0 + Condition | ParticipantName) + (0 + Word | ParticipantName)`.

The way we should specify these for a particular dataset can be checked by fitting a model with correlated random effects, then fitting a model with uncorrelated random effects, and entering both of their model outputs into an `anova()` function to compare which is a significantly better fit. For orthogonal polynomials (which have no theoretical reason to correlate), it may be the case

that uncorrelated slopes are better. In most cases though, correlated random effects will be a better fit.

## Random Slopes

There's always a question about what we should enter for random effects. The intercept is a no-brainer, and can be entered like `(1 | ParticipantName)`. Random slopes are more difficult. Barr, Levy, Scheepers, & Tily (in press) suggest that the specification of random slopes is a huge factor in model fits though, and can make or break the fitting and significance of a model. They recommend "keeping it maximal" and entering all possible random slopes. But what's a possible random slope?

If subjects are in multiple levels of a factor, their differential response to levels of the factor could be unique. For example, some people may react very strongly to angry words and very strongly to calm words. This would be a generally "high-reactive" person. Others may react modestly to both types of words, but still a little differently between levels. This would be a "low-reactive" person. So, these peoples' slopes differ between levels of a factor. We can account for these random effects of slope by adding this to our error term `(1 + Condition | ParticipantName)`. This will isolate idiosynratic slope differences but retain any main effects of condition.

Likewise, when we deal with time, some participants may have massive changes over time. Others may have smaller differences over time. We would likely then want to enter our natural or orthogonal timecodes as random slopes, too, like `(1 + Condition + ot1 + ot2 + ot3 | ParticipantName)`.

## p-values in lmer() Model Fits

By default, the lmer() function we use for model fitting will not return p-values representing the "significance" of our differences. Instead, it will return *t* statistics, or the parameter estimate divided by the standard error of the estimate. For large samples, these *t*'s can be assumed to be normally distributed with a standard deviation of 1 like *z*-scores. However, for smaller samples, they may overestimate significance.

If you don't assume the *t*'s are equivalent to *z*'s, we have a couple of other options for checking p-values. If we didn't use correlated random effects, we can use Markov Chain Monte Carlo (MCMC) sampling using the `languageR` packages's `pvals.fnc()` function. If we did use correlated random effects, we

can compare models with and without particular terms of our model using the `anova()` function which will do a log-likelihood ratio test and return a Chi square value, degrees of freedom, and p-value for the difference. Previously, `pMCMC` values returned by `pvals.fnc()` were the gold standard, but research from Barr (in revision) has suggested that Chi squares, then *t*'s, then `pMCMC` values are preferred, in that order.

## Model Fitting Examples

With the theoretical introductions settled, I will present several example model fitting calls here so that you can specify the model(s) that fit your data best. Each model will be called with the `data_time_bySubj` dataset we created at the beginning of this section. For a proper analysis though, we should also run each of these against our `data_time_byItem` dataset with the appropriately-changed random effects (focused on `Trials`, not `ParticipantNames`) to see if the effects generalize across both subjects and items.

**Fit a model predicting empirical logit transformed changes over linear time that differ between levels of our factor with weighted regression:**

```
> linear_bySubj <- lmer(elog ~ t1 + Condition + t1:Condition + (1 + t1
| ParticipantName), data = time_bySubj, weights = 1/wts)
```

A main effect of `t1` suggests a general change over time across all data. A main effect of `Condition` suggests a difference at the intercept (i.e., the start of our window, where `t1 = 0`) between conditions. A `t1:Condition` interaction suggests differential slopes over time.

**Fit a model predicting arcsin transformed changes over linear time that differ between levels of our factor:**

```
> linear_bySubj <- lmer(ArcSin ~ t1 + Condition + t1:Condition + (1 +
t1 | ParticipantName, data = time_bySubj)
```

**Fit a model on within-subjects data where participants were exposed to multiple levels of our factor:**

```
> linear_bySubj <- lmer(elog ~ t1 + Conditon + t1:Condition + (1 + t1 +
Condition | ParticipantName), data = time_bySubj, weights = 1/wts)
```

By adding `Condition` as a random slope, we are now controlling for differential slopes of condition between participants.

**Fit a model with two factors**:

```
> linear_bySubj <- lmer(elog ~ t1 + Condition + Word + t1:Condition +
t1:Word + t1:Condition:Word + (1 + t1 | ParticipantName), data =
time_bySubj, weights = 1/wts)
```

**Fit a model to the grand mean of our data to see which orthogonal polynomial codes may predict looking behaviour:**

```
> gca_test <- lmer(elog ~ ot1 + ot2 + ot3 + ot4 + (1 + ot1 + ot2 + ot3
+ ot4 | ParticipantName), data = time_bySubj, weights = 1/wts)
```

If we see significant main effects of any of these codes, we would follow up with a model like the one below where we look for differential fits to curve types between levels of our factor.

**Fit a model looking for curve type fit differences between levels of a factor:**

```
> gca_fit <- lmer(elog ~ ot1 + ot2 + ot3 + Condition + ot1:Condition +
ot2:Condition + ot3:Condition + (1 + ot1 + ot2 + ot3 |
ParticipantName), data = time_bySubj, weights = 1/wts)
```

A main effect of `Condition` here would suggest different *overall* looking times. This differences to natural polynomials (`t1` and `t2`) because those main effects are anticipatory differences, or differences at baseline.

**Fit a model looking for curve type fit differences between levels of a factor with uncorrelated random slopes for our orthogonal polynomial codes:**

```
> gca_fit <- lmer(elog ~ ot1 + ot2 + ot3 + Condition + ot1:Condition +
ot2:Condition + ot3:Condition + (1 | ParticipantName) + (0 + ot1 |
ParticipantName) + (0 + ot2 | ParticipantName) + (0 + ot3 |
ParticipantName), data = time_bySubj, weights = 1/wts)
```

Now we say the participant's likelihoods of fitting cubic vs. quadratic vs. linear functions are independent. This makes sense, as these are *orthogonal* codes. It also lets us use MCMC random sampling to see our p-values.

**Retrieve p-values with MCMC random sampling:**

```
> p_values <- pvals.fnc(gca_fit)
```

10,000 simulations are run to generate the `pMCMC` values which represent our p-values now.

**Retrieve p-values assuming that the *t*-scores distribute like *z*-scores:**

```
> p_values <- p_values(model_fit_output)
```

**Fit two models and compare their fits (in this case, to see if a main effect is significant)**:

```
> model1 <- lmer(elog ~ t1 + Condition + t1:Condition + (1 |
ParticipantName), data = data_time_bySubj, weights = 1/wts)
> model2 <- lmer(elog ~ t1 + t1:Condition + (1 | ParticipantName), data
= data_time_bySubj, weights = 1/wts)
> anova(model1, model2)
```

**Fit a model with the Items dataset to see if our differences hold up:**

```
> items_fit <- lmer(elog ~ t1 + Condition + t1:Condition + (1 +
Condition | TrialName), data = data_time_byItem, weights = 1/wts)
```

Because our items are seen by both conditions, we include condition as a random slope.

## Window Analysis

Up until this point, we have considered time as a factor in all of our analyses. Now, we will collapse across time and look at mean/total differences within our entire looking window with the `window_analysis()` function.

This function is slightly different than the others. Instead of passing full model specifications, this model simply accepts `data`, `dv`, and `factors` arguments and returns to you a dataset that is *ready* for analysis.

First, let's see a call:

```
> window_data <- window_analysis(data, dv = 'Animate',
factors=c('Condition','KnownAnimal'))
```

Our `data` object, the `dv` we want to analyze, and the `factors` with which we want to predict are all passed and we retrieve a `window_data` object. Here, unlike in the time analysis, we can include numeric covariates (e.g., age, vocabulary scores) in the list of factors. However, after we return the `window_data` object form the `window_analysis()` function, we must ensure that each numeric factor is converted back to a *numeric* mode (and is not treated as categorical).

```
> window_data$Age <- as.numeric(as.character(window_data$Age))
```

We can view this data object, but more importantly we can fit models to the data to examine the significance of our factors. Several new columns are automatically created for us, and these give us options in how we perform our analys.

We could perform an empirical logit transformed regression with weights using our new `elog` and `wts` columns, like so:

```
> model_elog <- lmer(elog ~ Condition + KnownAnimal + (1 |
ParticipantName) + (1 | Trial), data = window_data, weights = 1/wts)
```

We could also predict the value of an arcsin-root transformed version of our mean proportion for each trial (no weights):

```
> model_arcsin <- lmer(ArcSin ~ Condition + KnownAnimal + (1 |
ParticipantName) + (1 | Trial), data = window_data)
```

Either of the above are acceptable, as they both transform our non-linear proportional data into an approximation of the logistic curve. But, you could go rogue and predict the mean proportions themselves:

```
> model_proportions <- lmer(Proportion ~ Condition + KnownAnimal + (1 |
ParticipantName) + (1 | Trial), data = window_data)
```

The first two analyses are interchangeable, but the last is heresy.

The window analysis - as you can see - does not generate p-values or R-squared values like our previous functions. Here's how you generate these:

To generate p-values, use the languageR package's `pvals.fnc()` method by passing the model fitting output directly to it and using the `pMCMC` to see the significance of each parameter estimate as established with Markov Chain Monte Carlo sampling:

```
> pvals.fnc(model_elog)
```

Finally, if you would like to see the R-squared value for the model's fit to the data, you can use the following, substituting `model_fit` for the object holding the result of your `lmer()` call:

```
> summary(lm(attr(model_fit, "y") ~ fitted (model_fit)))$r.squared
```

## First Looks Analyses

Not finding any differences in cumulative looking time? You may want to try doing a "first looks" analysis, or a distractor-target-switch analysis. This analysis looks at each trial and compares the delay in switching away from the distractor to the delay in switching away from the target. It is predicted that babies who happen to be looking at the target at the onset of the trial will take longer to switch to the distractor than the reverse.

**Note: Because the `find_looks()` method includes the ability to smooth over trackloss gaps, you will likely want to create a new master clean (and subsequent master test) file to process with `find_looks()` so that you don't inflate the number of looks .**

This process begins with generate a looks dataframe for the test datafile:

```
source('process-tobii-data-bf.R')
looks <- find_looks(inputfile = 'master-test-with-trackloss.csv')
```

This dataframe holds data for each look, such as its length, beginning time, end time, fixation length rank, fixation order rank, etc. We will use this frame to generate another dataframe suitable for a first looks analysis, passing the old `looks` dataframe to the new function:

```
> first_looks <- first_looks(looks)
```

We can also optionally pass CSV subjects and trial files (identical to that used in creating our `master-prepared.csv` file) to this function in order to look at factors beyond condition, such as age, vocabulary, etc.

```
> first_looks <- first_looks(looks, 'subjects.csv', 'trials.csv')
```

We now have a table that we can analyze.

We may begin just by looking at the mean switch times for one beginning scene type (e.g., target) versus another (e.g., distractor):

```
> mean(first_looks[which(first_looks$FirstSceneType == 'Target'),
'SwitchTime'])
> mean(first_looks[which(first_looks$FirstSceneType == 'Distractor'),
'SwitchTime'])
```

```
> t.test(first_looks[which(first_looks$FirstSceneType == 'Target'),
'SwitchTime'], first_looks[which(first_looks$FirstSceneType ==
'Distractor'), 'SwitchTime'], var.equal = T)
```

Note that the above example does not aggregate over subjects and thus inflates the statistical power of the analysis by reducing independence between observations.

In order to aggregate over subjects, we can use mixed models with this dataset, or any of the techniques below:

```
> switches <- lmer(SwitchTime ~ Condition*Age*FirstSceneType + (1 |
Trial) + (1 | ParticipantName), data = first_looks)
```

**Analysing First Looks**

Now, we have the `first_looks` table that contains the switch times by subjects crossed with trials for each of our scene types. In order to analyse this, we can do one (or all) of these three analyses:

1) Mixed-effects model (crossing both subjects' and items' random effects)

2) Repeated-measures subjects ANOVA

3) Repeated-measures items ANOVA

For the mixed-effects model, we can do the following:

```
library(lme4)
output <- lmer(SwitchTime ~ Condition:FirstSceneType + (1 | Trial) + (1
| ParticipantName), data = first_looks)
```

For a repeated-measures subjects ANOVA, we can do this:

```
subjects_first <- aggregate(first_looks$SwitchTime, by =
list(first_looks$ParticipantName, first_looks$Condition,
first_looks$FirstSceneType), FUN = mean)
colnames(subjects_first) <- c('ParticipantName',
'Condition','FirstSceneType','SwitchTime')
output <- aov(SwitchTime ~ Condition* FirstSceneType +
Error(ParticipantName), data = subjects_first)
```

For a repeated-measures items ANOVA, we can do this:

```
items_first <- aggregate(first_looks$SwitchTime, by =
list(first_looks$Trial, first_looks$Condition,
first_looks$FirstSceneType), FUN = mean)
colnames(items_first) <- c('Trial',
'Condition','FirstSceneType','SwitchTime')
output <- aov(SwitchTime ~ Condition* FirstSceneType +
Error(Trial/Condition), data = items_first)
```

## References

Aslin, R. N. (2011). Infant Eyes: A Window on Cognitive Development. *Infancy*, Infant Eyes and Cognitive Development, *17*(1), 126–140.

Baayen, R. H., Davidson, D. J., & Bates, D. M. (2008). Mixed-effects modeling with crossed random effects for subjects and items. *Journal of Memory and Language*, *59*(4), 390–412.

Barr, D. J. (2008). Analyzing "visual world" eyetracking data using multilevel logistic regression. *Journal of Memory and Language*, *59*, 457–474.

Barr, D. J., Levy, R., Scheepers, C., & Tily, J. J. (in revision). Random effects structure in mixed-effects models: Keep it maximal.

Gelman, A., & Hill, J. (2007). *Data Analysis Using Regression And Multilevel/Hierarchical Models* (p. 625). Cambridge University Press.

Hoffman, L., & Rovine, M. J. (2007). Multilevel models for the experimental psychologist: Foundations and illustrative examples. *Behavior research methods*, *39*(1), 101–117.

Jaeger, T. F. (2008). Categorical data analysis: Away from ANOVAs (transformation or not) and towards logit mixed models. *Journal of Memory and Language*. *59*. 434-446.

Jaeger, T. F. (2012). Creating spaghetti plots of eye-tracking data in R. http://hlplab.wordpress.com/2012/02/27/creating-spaghetti-plots-of-eye-tracking-data-in-r/

Mirman, D., Dixon, J., & Magnuson, J. S. (2008). Statistical and computational models of the visual world paradigm: Growth curves and individual differences. *Journal of Memory and Language*. *59*. 475-494.