1 Introduction

In 2017, Oleson et al. introduced a method for detecting time-specific differences in the trajectory of outcomes between experimental groups **?**. Particularly in the case of a densely sampled time series, the construction of evaluating differences at each point in time results in a series of highly correlated test statistics expanding the family-wise error rate, accommodated with an adjustment to the nominal alpha based on this autocorrelation. This was followed up with in 2018 with the introduction of the `bdots` package to CRAN **?**. Here, we introduce the second version of `bdots`, an update to the package that broadly expands the capabilities of the original.

This manuscript is not intended to serve as a complete guide for using the `bdots` package. Instead, the purpose is to showcase major changes and improvements, with those seeking a more comprehensive treatment directed to the package vignettes. Rather than taking a "compare and contrast" approach, we will first enumerate the major changes, followed by a general demonstration of the package use:

1. Major changes to underlying methodology with implications for prior users of the package
2. Simplified user interface
3. Introduction of user defined curves
4. Permit fitting for arbitrary number of groups
5. Automatic detection of paired tests based on subject identifier
6. Allows for non-homogeneous sampling of data across subjects and groups
7. Introduce formula syntax for bootstrapping difference function
8. Interactive refitting process

We start by clearly delineating the type of problem that `bdots` has been created to solve.

Bootstrapped differences in time series A typical problem in the analysis of differences of time series, and the kind that `bdots` is intended to solve, involves that of two or more experimental groups containing subjects whose responses are measured over time. This may include the growth of tumors in mice or the change in the proportion of fixations over time in the context of the VWP. In either case, we assume that each of the subjects $i = 1, \ldots, n$ in the groups being considered has observed data of the following form:

$$y_{it} = f(t|\theta_i) + \epsilon_{it} \tag{1}$$

where f represents a functional mean structure while the error structure of ϵ_{it} is open to being either IID or possess an AR(1) structure. At present, `bdots` requires that each of the subjects being compared have the same parametric function f, though is not strictly necessary at the theoretical level and future directions

of the package include accommodating non-parametric functions. While each of the subjects are required to be of the same parametric form $f(\cdot|\theta)$, each differs in their instance of their own subject-specific parameters, θ_i.

An explicit assumption of the current iteration of `bdots` is that each subject is' parameters within a group $g = 1, \ldots, G$ is drawn from a group level distribution

$$\theta_i \sim N(\mu_g, V_g). \tag{2}$$

Bootstrapping parameters to estimate this distribution and evaluating the function f at these values gives us an estimate of the distribution of functions. As these function are *in time*, this in turn gives a representation of the temporal changes in group characteristics. It is precisely the identification of if and when these temporal changes differ between groups that `bdots` seeks to perform.

Homogeneous Means Assumption The assumption presented in Equation 2 differs from the original iteration of `bdots` in a critical way. In **?**, there was no assumption of variability between subject parameters, congruent the assumption that $\theta_i = \theta_j$ for all subjects i, j within an experimental group. The most relevant consequence of the homogeneous means assumption is that only within-subject variability is estimated, inflating the type I error when this assumption does not hold (see Chapter 3 for a discussion). For now, we will give a methodological overview of the process used by `bdots` along with a presentation of an updated bootstrapping process and the introduction of a permutation test.

2 Methodology and Overview

A standard analysis using `bdots` consists of two steps: fitting the observed data to a specified parametric function, $f(\cdot|\theta)$, and then using the observed variability to construct estimates of the distributions of groups whose differences we wish to investigate. Here, we briefly detail how this is implemented in practice and introduce the new methodologies in `bdots`. A more comprehensive treatment of these new methods, along with their justifications, is offered in Chapter 3.

2.1 Establishing subject-level curves not sure if these should be subsections or paragraphs

We begin with the assumption that for subject i in group g, we have collected observed data of the form given in Equation 1, with the subject specific parameter θ_i following the distribution in Equation 2. Each

subject is then fit in `bdots` via the nonlinear curve fitting function `nlme::gnls`, returning for each set of observed data an estimated set of parameters $\hat{\theta}_i$ and their associated standard errors. Assuming large sample normality, we are able to construct a sampling distribution for each subject:

$$\hat{\theta}_i \sim N(\theta_i, s_i^2). \tag{3}$$

This provides us with an estimate of within-subject variability of the observed parameters.

2.2 Estimating Group Distributions

Once sampling distributions are created for each subject, we are prepared to begin estimating group distributions given in Equation 2. We then propose the following algorithm for creating bootstrapped estimates of the group distributions:

1. For a group of size n, select n subjects from the group *with replacement*. This allows us to construct an estimate of V_g.

2. For each selected subject i in bootstrap b, draw a set of parameters from the distribution

$$\theta_{ib}^* \sim N(\hat{\theta}_i, s_i^2). \tag{4}$$

3. The the mean of each of the bootstrapped θ_{ib}^* in group g to construct the bth group bootstrap, θ_{gb}^* where

$$\theta_{gb}^* \sim N\left(\mu_g, \frac{1}{n}V_g + \frac{1}{n^2}\sum s_i^2\right) \tag{5}$$

4. Perform steps (1)-(3) B times, using each θ_b^* to construct a distribution of population curves, $f(\cdot|\mu_g)$. (not sure I like this notation)

Note that the distribution in Equation 5 differs from that under the homogeneous means assumption by the factor of V_g in the variance term.

The final population curves from (4) can be used to create estimates of the mean response and an associated standard deviation at each time point for each of the groups bootstrapped. These estimates are used both for plotting and in the construction of confidence intervals. They also can be, but do not necessarily have to be, used to construct a test statistic, which is the topic of our next section.

2.3 Hypothesis testing for statistically significant differences

We now turn our attention to the primary goal of an analysis in `bdots`, the identification of time windows in which the distribution of curves of two groups differ significantly. A problem unique to the ones addressed by `bdots` is that of multiple testing; and especially in densely sampled time series, we must account for multiple testing while controlling the family-wise error rate (FWER). There are primarily two ways by which we are able to do this which we detail below.

2.3.1 α Adjustment

Just as in the original iteration of `bdots`, we are able to construct test statistics from the bootstrapped estimates described in the previous section. These bootstrapped test statistics $T_t^{(b)}$ can be written as

$$T_t^{(b)} = \frac{(\overline{p}_{1t} - \overline{p}_{2t})}{\sqrt{s_{1t}^2 + s_{2t}^2}}, \tag{6}$$

where $\overline{p}_{gt}$ and s_{gt}^2 are mean and standard deviation estimates at each time point t and for groups 1 and 2, respectively. As was demonstrated in **?**, these test statistics can be highly correlated in the presence of densely sampled test statics, leading to an inflated type I error. The FWER in this case can be controlled with the Oleson adjustment proposed in original bootstrap paper. In addition to this, adjustments to the nominal alpha can also be made using all of the adjustments present in `p.adjust` from the R `stats` package.

2.3.2 Permutation testing

In addition to modified correction based on the bootstrapped test statistics, `bdots` provides a permutation test for controlling the FWER without any additional assumptions of autocorrelation.

In doing so, we begin by creating an observed test statistic in the following way: first, taking each subject's estimated parameter $\hat{\theta}_i$, we find the subject's corresponding parametric curve $f(t|\hat{\theta}_i)$. Within each group, we use *these* curves to create estimates of the mean population curves and associated standard errors at each each point[1]. Letting p_{gt} and s_{gt}^2 represent the mean population curve and standard error for group g at time t, we define our observed permutation test statistic,

$$T_t^{(p)} = \frac{|\overline{p}_{1t} - \overline{p}_{2t}|}{\sqrt{s_{1t}^2 + s_{2t}^2}}. \tag{7}$$

.

[1]This differs from the bootstrapped test statistic in which the mean of the subjects' parameters was used to fit a population curve, i.e., $\frac{1}{n}\sum f(t|\theta_i)$ compared with $f\left(t|\frac{1}{n}\sum\theta_i\right)$. The implications of this have not been investigated any further

We then going about using permutations to construct a null distribution against which to compare the observed statistics from Equation 7. We do so with the following algorithm:

1. Assign to each subject a label indicating group membership
2. Randomly shuffle the labels assigned in (1.), creating two new groups
3. Recalculate the test statistic $T_t^{(p)}$, recording the maximum value from each permutation
4. Repeat (2.)-(3.) P times. The collection of P statistics will serve as our null distribution, denoted $\widetilde{T}$. Let $\widetilde{T}_\alpha$ be the $1-\alpha$ quantile of $\widetilde{T}$. Areas where the observed $T_t^{(p)} > \widetilde{T}_\alpha$ are designated significant.

Paired statistics can also be constructed in both the bootstrap and permutation methods. This is implemented by ensuring that at each bootstrap the same subjects are selected for each group or by ensuring that each permuted group contains one observation from each subject.

A demonstration of power and FWER control for both the heterogeneous bootstrap and permutation test are given in Chapter 3.

3 Example Analysis

In this next section we are going to review a worked example of a typical use of the `bdots` package. We will use as our illustration a study (source?) comparing tumor growth for the 451LuBr cell line in mice data with repeated measures in five treatment groups.

```
> head(mouse, n = 10)
      Volume Day Treatment ID
 1:   47.432   0         A  1
 2:   98.315   5         A  1
 3:  593.028  15         A  1
 4:  565.000  19         A  1
 5: 1041.880  26         A  1
 6: 1555.200  30         A  1
 7:   36.000   0         B  2
 8:   34.222   4         B  2
 9:   45.600  10         B  2
10:   87.500  16         B  2
```

Figure 1: Illustration of Mouse data in long format

A new feature of `bdots` is the ability to fit and analyze subjects with non-homogeneous time samples, which we see in the `Day` variable values in the mouse data in Figure 1. Details on how to adjust how non-homogenous times are handled in a later section (they're not – I'm not sure what to do abou this yet when

bootstrapping functions that are FAR outside of their observed range. I almost think the default should be a mini-max approach, creating an arbitrary range of values in the intersection of the range of all of the observed data, but with ability to specify time range directly in bootstrap function). For the present analysis, we are interested in determining if and when the trajectory of tumor growth, measured in (Volume?) changes between any two treatment groups.

There are two primary functions in the `bdots` package: one for fitting the observed data to a parametric function and another for estimating group distributions and identifying time windows where they differ significantly. The first of these, `bfit`, is addressed in the next section.

3.1 Curve Fitting

The curve fitting process is performed with the `bfit` function (previously `bdotsFit`), taking the following arguments:

```
bfit(data, subject, time, y, group, curveType, ar, ...)
```

Figure 2: Main arguments to `bfit`, though see `help(bfit)` for additional options

The `data` argument takes the name of the dataset being used. `subject` is the subject identifier column in the data and should be passed as a character. The `time` and `y` arguments are column names of the time variable and outcome, respectively. Similarly, `group` takes as an argument a character vector of each of the group columns that are meant to be fit, accommodating the fact that `bdots` is now able to fit an arbitrary number of groups at once, provided that the outcomes in each group adopt the same parametric form. This brings us to the `curveType` argument, which is addressed in the next section. [need to describe `ar`]

[say this somewhere] It is important to note here that the identification of paired data is done automatically; in determining if two experimental groups are paired, `bdots` checks that the intersection of subjects in each of the groups are identical with the subjects in each of the groups individually.

Curve functions Whereas the previous iteration of `bdots` had a separate fitting function for each parametric form (i.e., `logistic.fit` for fitting data to a four-parameter logistic), we are now able to specify the curves we wish to fit independent of the fitting function `bfit`. This is done with the `curveType` argument. Unlike the previous arguments which took either a `data.frame` or character vector, `curveType` takes as an argument a function call, for example, `logistic()`. The motivation for this is detailed elsewhere (the appendix, maybe?), but in short, it allows the user to pass additional arguments to further specify the curve. For example, among the parametric functions included in `bdots` is now the `polynomial` function, taking as an additional argument the number of degrees we wish to use. To fit the observed data with a

five parameter polynomial in `bfit`, one would then pass the argument `curveType = polynomial(degree = 5)`. Curve functions currently included in `bdots` include `logistic()`, `doubleGauss()`, `expCurve()`, and `polynomial()`. `bfit` can also accept user-created curves; detailed vignettes for writing your own can be found with `vignette("bdots")`. [and maybe the appendix]

We fit the mouse data to an exponential curve with `expCurve()` and using the column names found in Figure 1:

```
mouse_fit <- bfit(data = mouse, subject = "ID", time = "Day",
                  y = "Volume", group = "Treatment", curveType = expCurve())
```

Return object and generics The function `bfit` returns an object of class `bdotsObj`, inheriting from class `data.table`. As such, each row uniquely identifies one permutation of subject and group values. Included in this row are the subject identifier, group classification, summary statistics regarding the curves, and a nested `gnls` object. Inheriting from `data.table` also permits us to use `data.table` syntax to subset the object as in Figure 5, for example, where we elect to only plot the first four subjects.

```
> class(mouse_fit)
[1] "bdotsObj"   "data.table" "data.frame"

> head(mouse_fit)
   ID Treatment        fit      R2   AR1 fitCode
1:  1         A <gnls[18]> 0.97349 FALSE       3
2:  2         B <gnls[18]> 0.83620 FALSE       4
3:  3         E <gnls[18]> 0.96249 FALSE       3
4:  4         C <gnls[18]> 0.96720 FALSE       3
5:  5         D <gnls[18]> 0.76156 FALSE       5
6:  7         B <gnls[18]> 0.96361 FALSE       3
```

Figure 3: A `bfit` object inheriting from `data.frame`

The number of columns will depend on the total number of groups specified, with the subject and group identifiers always being the first columns. Following this is the `fit` column, which contains the fitted object returned from `gnls`, as well as `R2` indicating the R^2 statistic. The `AR1` column indicates whether or not the observed data was able to be fit with an AR(1) error assumption. Finally, there is the `fitCode` column, which we will describe in more detail shortly.

Several methods exist for this object, including `plot`, `summary`, and `coef`, returning a matrix of fitted coefficients obtained from `gnls`.

Fit Codes The `bdots` package was originally introduced to address a very narrow scope of problems, and the `fitCode` designation is an artifact of this original intent. Specifically, it assumed that all of the

observed data was of the form given in Equation 1 where the observed time series was dense and the errors were autocorrelated. Autocorrelated errors can be specified in the `gnls` package (used internally by `bdots`) when generating subject fits, though there were times when the fitter would be incapable of converging on a solution. In that instance, the autocorrelation assumption was dropped and constructing a fit was reattempted.

R^2 proved a reliable metric for this kind of data, and preference was given to fits with an autocorrelated error structure over those without. From this, the hierarchy given in Table 1 was born. `fitCode` is a numeric summary statistic ranked from 0 to 6 detailing information about the quality of the fitted curve, constructed with the following pseudo-code:

```
AR1 <- # boolean, determines AR1 status of fit
fitCode <- 3*(!AR1) + 1*(R2 < 0.95)*(R2 > 0.8) + 2*(R2 < 0.8)
```

A fit code of 6 indicates that `gnls` was unable to successfully fit the subject's data.

`bdots` today stands to accommodate a far broader range of data for which the original `fitCode` standard may no longer be relevant. The presence of autocorrelation cannot always be assumed, and users may opt for a metric other than R^2 for assessing the quality of the fits. Even the assessments of fits on a discretized scale may be something of only passing interest. Even then, however, this is how the current implementation of `bdots` categorizes the quality of its fits, with the creation of greater flexibility in this regard being a large priority for future directions. Outside of general summary information, the largest impact of this system is in the refitting process, which organizes the fits by `fitCode`. There is still flexibility in how this is handled, though we will reserve discussion for the relevant section.

`fitCode`	AR(1)	R^2
0	TRUE	$R^2 > 0.95$
1	TRUE	$0.8 < R^2 < 0.95$
2	TRUE	$R^2 < 0.8$
3	FALSE	$R^2 > 0.95$
4	FALSE	$0.8 < R^2 < 0.95$
5	FALSE	$R^2 < 0.8$
6	NA	NA

Table 1: Description of the `fitCode` statistic

Summaries and Plots Users are able to quickly summarize the quality of the fits with the `summary` method now provided.

```
> summary(mouse_fit)

bdotsFit Summary

Curve Type: expCurve
Formula: Volume ~ x0 * exp(Day * k)
Time Range: (0, 106) [31 points]

Treatment: A
Num Obs:  10
Parameter Values:
        x0          k
172.232953   0.056843
########################################
############### FITS ###################
########################################
AR1,       0.95 <= R2        -- 2
AR1,       0.80 < R2 <= 0.95 -- 1
AR1,       R2 < 0.8          -- 0
Non-AR1,   0.95 <= R2        -- 0
Non-AR1,   0.8 < R2 <= 0.95  -- 3
Non-AR1,   R2 < 0.8          -- 4
No Fit                       -- 0

[...]

All Fits
Num Obs:  42
Parameter Values:
        x0          k
102.487118   0.053662
########################################
############### FITS ###################
########################################
AR1,       0.95 <= R2        -- 4
AR1,       0.80 < R2 <= 0.95 -- 2
AR1,       R2 < 0.8          -- 0
Non-AR1,   0.95 <= R2        -- 9
Non-AR1,   0.8 < R2 <= 0.95  -- 16
Non-AR1,   R2 < 0.8          -- 11
No Fit                       -- 0
```

Figure 4: Abridged output from the summary function (missing summary information for groups B-E. Note that this includes data on the formula used, the quality of fits and mean parameter estimates by group, and a summary of all fits combined

It is also recommended that users visually inspect the quality of fits for their subjects, which includes a plot of both the observed and fit data. There are a number of options available in `?plot.bdotsObj`, including the option to fit the plots in base R rather than `ggplot2`. This is especially helpful when looking to quickly assess the quality of its (rather than reporting) because `ggplot2` can be notoriously slow with large data sets. Figure 5 includes a plot of the first four fitted subjects.

```
plot(mouse_fit[1:4, ])
```

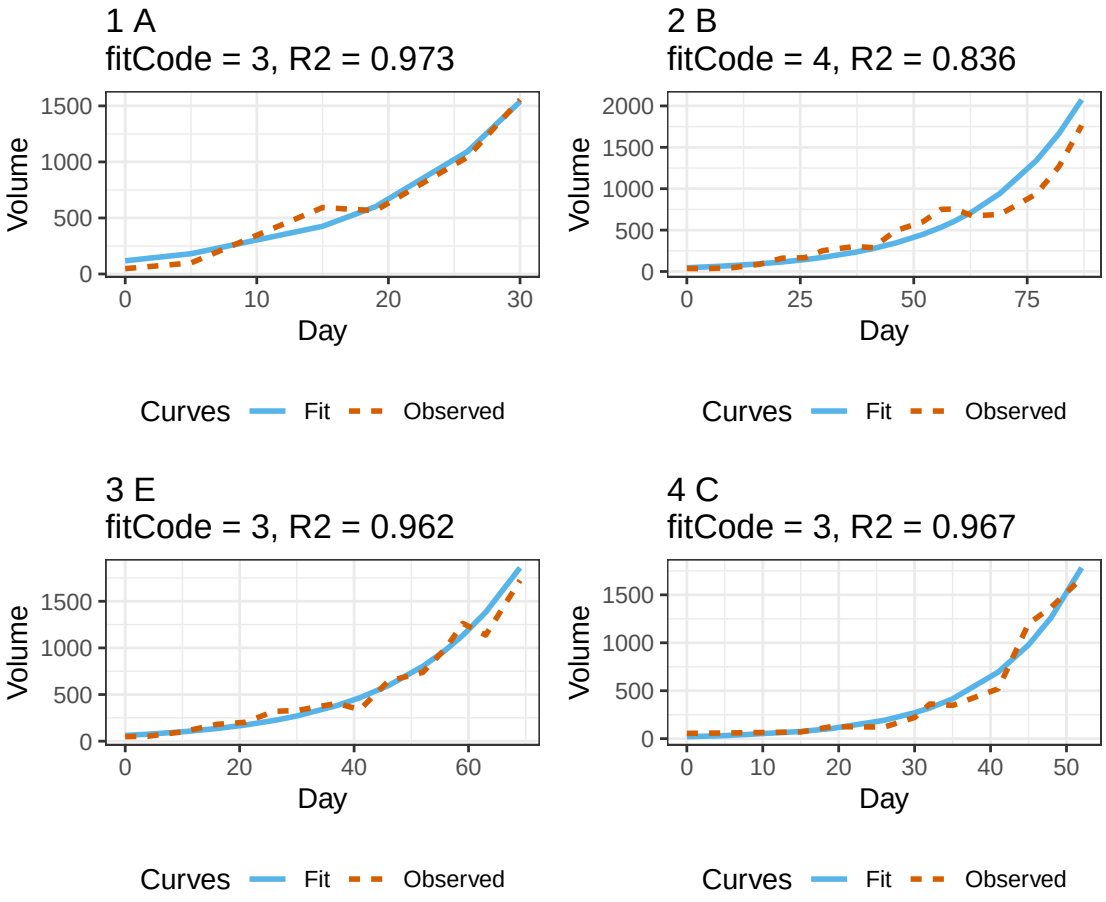

Figure 5: Plot of `mouse_fit`, using `ggplot2` (default) and `data.table` syntax to subset to only the first four observations

3.2 Bootstrapping

Once fits have been made, we are ready to begin estimating the group distributions and investigating temporal differences. This is done with the bootstrapping function, `bboot`. The number of options included in the `bboot` function have expanded to include a new formula syntax for specifying the analysis of interest as well as to include options for permutation testing. A call to `bboot` takes the following form:

```
bboot(formula, bdObj, B, alpha, permutation = TRUE, padj = "oleson", ...)
```

The `formula` argument is new to `bdots` and will be discussed in the next section. As for the remaining arguments, `bdObj` is simply the object returned from `bfit` that we wish to investigate, and `B` serves the dual

role of indicating the number of bootstraps/permutations we wish to perform; `alpha` is the rate at which we wish to control the FWER. `permutation` and `padj` work in contrast to one another: when `permutation = TRUE`, the argument to `padj` is ignored. Otherwise, `padj` indicates the method to be used in adjusting the nominal `alpha` to control the FWER. By default, `padj = "oleson"`. Finally, as previously mentioned, there is no longer a need to specify if the groups are paired, and `bboot` determines this automatically based on the subject identifiers in each of the groups.

Formula As the `bfit` function is now able to create fits for an arbitrary number of groups at once, we rely on a formula syntax in `bboot` to specify precisely which groups differences we wish to compare. Let `y` designate the outcome variable indicated in the `bfit` function and let `group` be one of the group column names to which our functions were fit. Further, let `val1` and `val2` be two values within the `group` column. The general syntax for the `bboot` function takes the following form:

```
y ~ group(val1, val2)
```

Note that this is an *expression* in R and is written without quotation marks. To give a more concrete example, suppose we wished to compare the difference in tumor growth curves for A and B from the `Treatment` column in our mouse data (Figure 1). We would do so with the following syntax:

```
Volume ~ Treatment(A, B)
```

There are two special cases to consider when writing this syntax. The first is the situation that arises in the case of multiple or nested groups, the second when a difference of difference analysis is conducted. Details on both of these cases are handled in the appendix.

Summary and Analysis Let's begin first by running `bboot` using bootstrapping to compare the difference in tumor growth between treatment groups A and E in our mouse data using permutations to test for regions of significant difference.

```
mouse_boot <- bboot(Volume ~ Treatment(A, E), bdObj = mouse_fit, permutation = TRUE)
```

This returns an object of class `bdotsBootObj`. A summary method is included to display relevant information:

```
> summary(mouse_boot)

bdotsBoot Summary

Curve Type: expCurve
Formula: Volume ~ x0 * exp(Day * k)
Time Range: (0, 59) [21 points]

Difference of difference: FALSE
Paired t-test: FALSE
Difference: Treatment

FWER adjust method: Permutation
Alpha: 0.05
Significant Intervals:
     [,1] [,2]
[1,]   15   32
```

There are a few components of the summary that are worth identifying when reporting the results. In particular, note the time range provided, an indicator of if the test was paired, and which groups were being considered (noticing now it only has `Treatment`, not `A` or `E`). The last section of the summary indicates the testing method used, an adjusted `alphastar` if `permutation = FALSE`, and a matrix of regions identified as being significantly different. This matrix is `NULL` is no differences were identified at the specified alpha; otherwise there is one row included for each disjointed region of significant difference.

In addition to the provided summary output, a `plot` method is available, with a list of additional options included in `help(plot.bdotsBootObj)`.

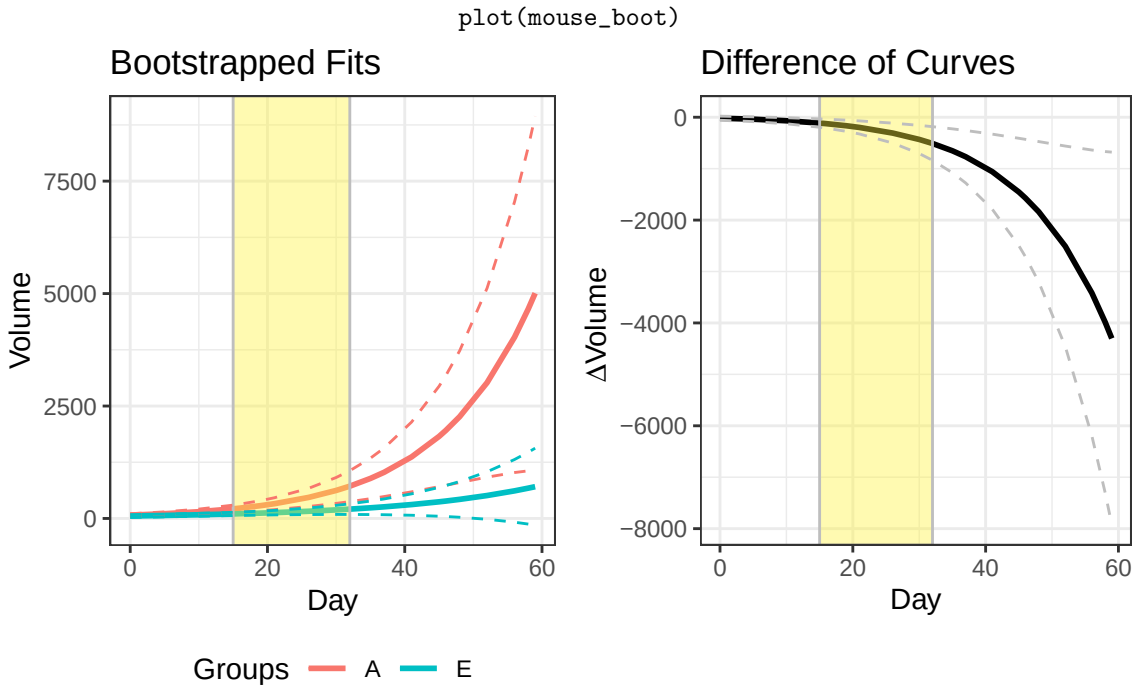

Figure 6: Bootstrapped distributions with regions of significant difference determined via permutation testing There are some obvious issues with time for non-homogenous samples, namely, what do we use for bootstrapping? It will be quick fix, whatever we decide, but I don't think "union of all observed times" is going to work. Here, I artificially cut it back to only 0-60

4 Ancillary Functions

Include here are a number of different function in `bdots` facilitating analysis but are otherwise not strictly necessary

4.1 Refitting

There are sometimes situations in which the fitted function returned by `bfit` is a poor fit. The nonlinear curve fitting algorithm used by `nlme::gnls` in `bfit` can be sensitive to starting parameters. Sensible starting parameters are computed from the observed data as part of the curve fitting functions (i.e., within the `logistic()` function), though these can often be improved upon.

The quality of the fit can be evidenced by the `fitCode` or via a visual inspection of the fitted functions against the observations for each subject. When this occurs, there are several options available to the user, all of which are provided through the function `brefit` (previously `bdotsRefit`). `brefit` takes the following

arguments:

```
brefit(bdObj, fitCode = 1L, subset = NULL, quickRefit = FALSE, paramDT = NULL)
```

The first of these arguments outside of the `bdObj` is `fitCode`, indicating the minimum fit code to be included in the refitting process. As discussed in Section 3.1, this can be sub-optimal. To add flexibility to which subjects are fit there is now the `subset` argument taking either a logical expression or collection of indices that would be used to subset an object of class `data.table` or a numeric vector with indices that the user wishes to refit. For example, we could elect to refit only the first 10 subjects or refit subjects with $R^2 < 0.9$:

```
refit <- brefit(fit, subset = 1:10) # refit the first 10 subjects
refit <- brefit(fit, subset = R2 < 0.9) # refit subjects with R2 < 0.9
```

When an argument is passed to subset, the `fitCode` is completely ignored.

To assist with the refitting process is the argument `quickRefit`. When set to `TRUE`, `brefit` will take the average coefficients of accepted fits within a group and use those as new starting parameters for poor fits. The new fits will be retained if they have a larger R^2 value by default. When set to `quickRefit = FALSE`, the user will be guided through a set of prompts to refit each of the curves manually.

Finally, the `paramDT` argument allows for a `data.table` with columns for subject, group identifiers, and parameters to be passed in as a new set of starting parameters. This `data.table` requires the same format as that returned by `bdots::coefWriteout`. The use of this functionality is covered in more detail in the `bdots` vignettes and is a useful way for reproducing a `bdotsObj` from a plain text file.

When `quickRefit = FALSE`, the user is put through a series of prompts along with a series of diagnostics for each of the subjects to be refit. Here, for example, is the option to refit subject 11 from the mouse data:

```
Subject: 11
R2: 0.837
AR1: FALSE
rho: 0.9
fitCode: 4

 Model Parameters:
       x0         k
53.186497  0.051749

Actions:
1) Keep original fit
2) Jitter parameters
3) Adjust starting parameters manually
4) Remove AR1 assumption
5) See original fit metrics
6) Delete subject
99) Save and exit refitter
Choose (1-6):
```

There are a number of options provided in this list. The first, of course, keeps the original fit of the presented subject and moves on to the next subject in the list. The second option takes the values of the fitted parameter and "jitters" them, changing each of the values by a prespecified magnitude. Given the sensitivity of `nlme::gnls` to starting parameters, this is sometimes enough for the fitter to converge on a better fit for the observed data. Alternatively, the third option gives the user the ability to select the starting parameters manually. The third option gives the user the ability to attempting refitting the observed data without an AR(1) error assumption, though this is only relevant if such an assumption exists. Option (5) reprints summary information and the final option allows the user to delete the subject all together.

When any attempt to refit the observed under new conditions is presented (options (2)-(4)), a plot is rendered comparing the original fit side-by-side with the new alternative, Figure 7.

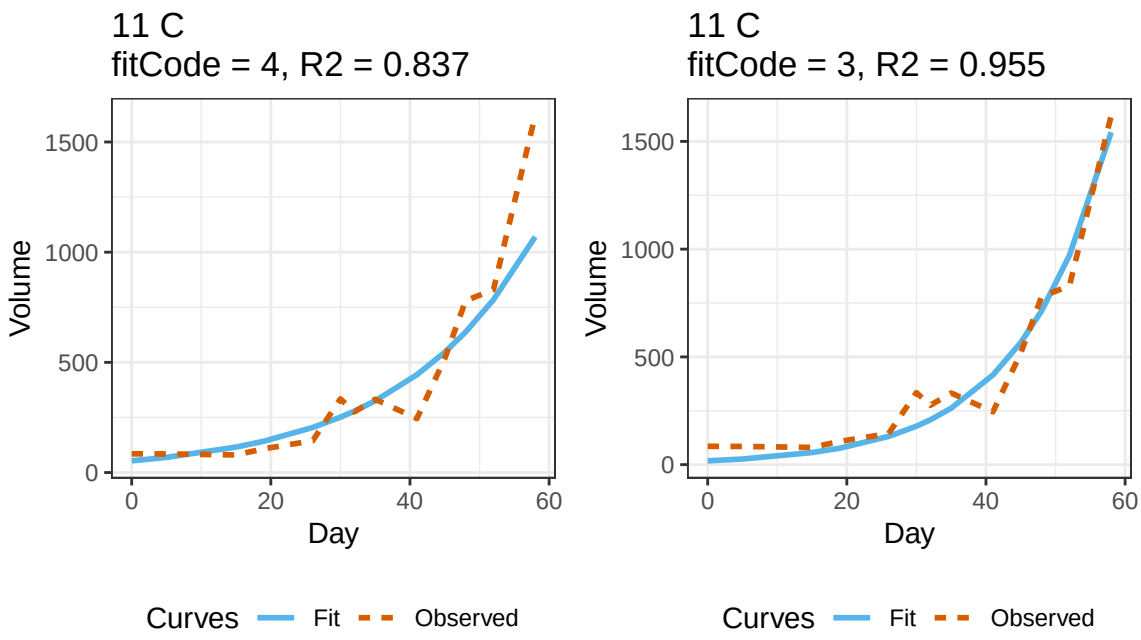

Figure 7: before and after refit

As the menu item suggests, users have the ability to end the manually refitting process early and save where they had left off. To retain previously refit items and start again at a later time, pass the first refitted object back into the refitter as such:

```
refit <- brefit(fit, ...)
refit <- brefit(refit, ...) # pass in the refitted object
```

A final note should be said regarding the option to delete a subject. As `bdots` now automatically determines if subjects are paired based on subject identifiers (necessary for calculations in significance testing), it is critical that if a subject has a poor fit in one group and must be removed that he or she is also removed from all additional groups in order to retain paired status. This can be overwritten with a final prompt in the `brefit` function before they are removed. The removal of subjects can also be done with the ancillary function, `bdRemove`, useful for removing subjects without undergoing the entire refitting process. See `help(bdRemove)` for details.

4.2 Correlations

There are sometimes cases in which we are interested in determining the correlation of a fixed attribute with group outcome responses across time . This can be done with the `bcorr` function (previously `bdotsCorr`),

which takes as an argument an object of class `bdotsObj` as well as a character vector representing a column from the original dataset used in `bfit`

```
bcorr(fit, "value", ciBands, method = "pearson")
```

Need to elaborate here with example

4.3 α Adjustment

There may also be situations in which users wish to make an adjustment to autorcorrelated test statistics using the modified Bonferonni adjustment provided in **?**, though in a different context than what is done in `bdots`. To facilitate this, we introduce an extension to the `p.adjust` function, `p_adjust`, identical to `p.adjust` except that it accepts method `"oleson"` and takes additional arguments `rho`, and `df`. `rho` determines the autocorrelation estimate for the oleson adjustment while `df` returns the degrees of freedom used to compute the original vector of t-statistics. If an estimate of `rho` isn't available, one can be computed on a vector of t-statistics using the `ar1Solver` function in `bdots`:

```
t       <- diffinv(rnorm(5))
rho     <- ar1Solver(t)
unadj_p <- pt(t, df = 10)
adj_p   <- p_adjust(unadj_p, method = "oleson",
                    df = 10, rho = rho, alpha = 0.05)
```

The `p_adjust` function returns both adjusted p-values, which can be compared against the specified alpha (in this case, 0.05) along with an estimate of alphastar, a nominal alpha at which one can compare the original p-values:

```
> unadj_p
[1] 0.5000000 0.0849965 0.0381715 0.1601033 0.0247453 0.0013016
> adj_p
[1] 0.9201915 0.1564261 0.0702501 0.2946514 0.0455408 0.0023954
attr(,"alphastar")
[1] 0.027168
```

Here, for example, we see that the last two positions of `unadj_p` have values less than `alphastar`, identifying them as significant; alternatively, we see these same two indices in `adj_p` significant when compared to `alpha = 0.05`

5 Discussion

The original implementation of `bdots` set out to address a narrow set of problems. Previous solutions beget new opportunities, however, and it is in this space that the second iteration of `bdots` has sought to expand. Since then, the interface between user and application has been significantly revamped, creating a intuitive, reproducible workflow that is able to quickly and simply address a broader range of problems. The underlying methodology has also been improved and expanded upon, offering better control of the family-wise error rate.

While significant improvements have been made, there is room for further expansion. The most obvious of these is the need to include support for non-parametric functions, the utility of which cannot be overstated. Not only would this alleviate the need for the researcher to specify in advance a functional form for the data, it would implicitly accommodate more heterogeneity of functional forms within a group. Along with this, the current implementation is also limited in the quality-of-fit statistics used in the fitting steps to assess performance. R^2 and the presence of autocorrelation are relevant to only a subset of the types of data that can be fit, and allowing users more flexibility in specifying this metric is an active goal for future work. In all, future directions of this package will be primarily focused on user interface, non-parametric functions, and greater flexibility in defining metrics for fitted objects.

6 Appendix

This section currently commented out. Involves instructions for fitting difference of difference and nested groups. Also may possibly include custom curve fitting.

6.1 Formula for nested groups and difference of differences

The formula syntax introduced in Section 3.2 is straightforward enough in the case in which we are interested in comparing two groups within a single category, as was the case when we compared two treatment groups, both within the `Treatment` column. As `bdots` now allows multiple groups to be fit at once, there may be situations in which we need more precision in specifying what exactly we wish to compare. Consider for example an artificial dataset that contains some outcome `y` for a collection of vehicles, consisting of eight distinct groups, nested in order of vehicle origin (foreign or domestic), vehicle class (car or truck), and vehicle color (red or blue)

We will illustrate use of the updated `bdots` package with a worked example, using an artificial dataset to help detail some of the newer aspects of the package. The dataset will consist of outcomes for a collection

of vehicles, consisting of eight distinct groups. These groups will be nested in order of vehicle origin (foreign or domesetic), vehicle class (car or truck), and vehicle color (red or blue). Further, vehicles of different color but within the same origin and class groups will be considered paired observations. A table detailing the relationship of the groups is given in Table 2.

Origin	Class	Color
foreign	car	red
		blue
	truck	red
		blue
domestic	car	red
		blue
	truck	red
		blue

Table 2: Example of nested vehicle classes

Beginning with a simple case, suppose we want to investigate the difference in outcome between all foreign and domestic vehicles. Notionally, we would write

```
y ~ Origin(foreign, domestic)
```

just as we did in the mouse data example: here, the name of the group variable `Origin`, followed by the values we are interested in comparing, `domestic` and `foreign`. Alternatively, if we wanted to limit our investigation to only foreign and domestic *trucks*, we would do this by including an extra term specifying the group and the desired value. In this case,

```
y ~ Origin(foreign, domestic) + Class(truck).
```

Similarly, to compare only foreign and domestic *red* trucks, we would add an additional term for color:

```
y ~ Origin(foreign, domestic) + Class(truck) + Color(red)
```

There are also instances in which we might be considered in the interaction between two groups. Although there is no native way to handle interactions in `bdots`, this can be done indirectly through the difference of differences `?`. To illustrate, suppose we are interested in understanding how the color of the vehicle differentially impacts outcome based on the vehicle class. In such a case, we might look at the difference in outcome between red cars and red trucks and then compare this against the difference between blue cars and blue trucks. Any difference between these two differences would give information regarding the differential impact of color between each of the two classes. This is done in `bdots` using the `diffs` syntax in the formula:

```
diffs(y, Class(car, truck)) ~ Color(red, blue)
```

Here, the *outcome* that we are considering is the difference between vehicle classes, with the groups we are interested in comparing being color. This is helpful in remembering which term goes on the left hand side of the formula.

Similar as to the case before, if we wanted to limit this difference of differences investigation to only include domestic vehicles, we can do so by including an additional term:

```
diffs(y, Class(car, truck)) ~ Color(red, blue) + Origin(domestic).
```

As before, this can be further subset with an arbitrary number of nested groups.

7 Writing Custom Curve Functions

One of the most significant changes in the newest version of `bdots` is the ability to specify the parametric curve independently of the fitting function. Not only does this simply a typical analysis, reducing all fitting operations to the single function `bfit`, it also provides users with a way to modify this function to meet their own needs. In this section we will detail how the curve function is used in `bdots` and how users can write their own. Finally, we will conclude with an example of how this was used in Chapter 2 to create adequate fits with the look onset method when the typical method for constructing estimated starting parameters based on the proportion of fixation method failed.

To begin, it is important to understand a little of how `bdots` works internally. In the curve fitting steps using `bfit`, the data is split by subject and group, creating a list whereby each element is the set of all observations for a single subject and a single group (i.e., in a paired setting, an individual subject would have two separate elements in this list). Ultimately, the data in each element will be used to construct a set of estimated parameters and standard errors for each subject provided by the function `lmer::gnls`. Doing so requires both (1) a formula to which we fit the data and (2) starting parameter estimates. Proving the both of these is the role of the curve function. Figure 8 provides an illustration of this process.

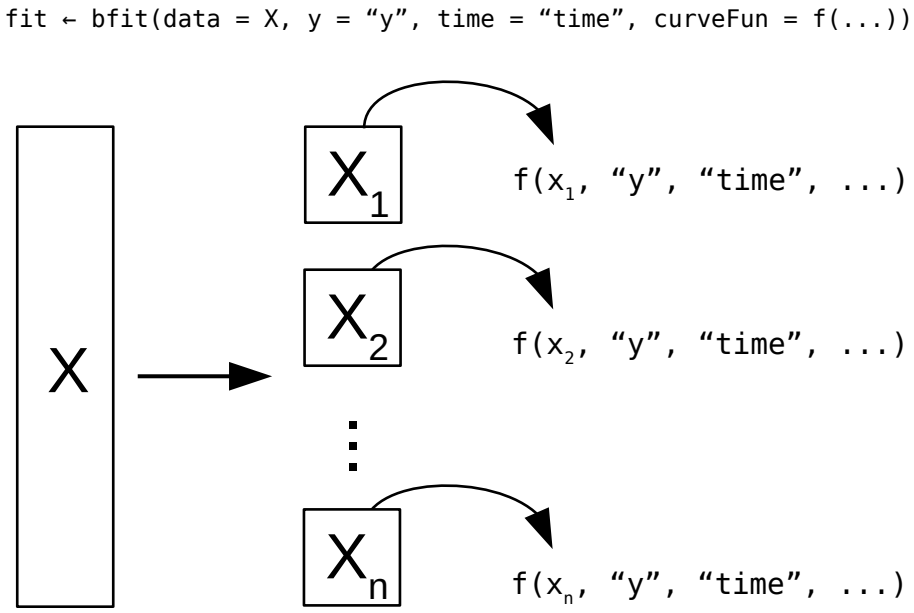

Figure 8: A call to the function `bfit` with data `X` and outcome and time variables `"y"` and `"time"`. `bfit` splits the dataset `X` by subject/group and passes each individual `data.frame` into the curve function `f()`, along with time and outcome character vectors as well as any other arguments passed into `....` In particular, the `...` argument allows the user to specify characteristics of the curve function that apply to all instances, as would be the case, for example, if `curveFun = polynomial(degree = 5)`. Finally, each instance of `f(...)` returns both a formula for `lmer::gnls` as well as subject-specific starting parameters.

We turn our attention now to the curve function itself, using as an example a curve function for sitting a straight line to the observed data, given in Figure 9. In describing the purpose for each, we also include enough detail so that this may be used as a template in constructing a new one all together. We do this in an enumerated list, each item corresponding to the numbered portion in Figure 9.

```
① linear <- function (dat, y, time, params = NULL, ...) {

      linearPars <- function(dat, y, time) {
          time <- dat[[time]]
②         y <- dat[[y]]
          if (var(y) == 0) {
              return(NULL)
          }
          mm <- (max(y) - min(y))/max(time)
          bb <- mean(y) - mm * mean(time)
          return(c(intercept = bb, slope = mm))
      }

③     if (is.null(params)) {
          params <- linearPars(dat, y, time)
      }
④     if (is.null(params)) {
          return(NULL)
      }
      y <- str2lang(y)
      time <- str2lang(time)
⑤     ff <- bquote(.(y) ~ slope * .(time) + intercept)
      attr(ff, "parnames") <- names(params)
      return(list(formula = ff, params = params))
  }
```

Figure 9: An example curve function with its constituent parts

1. The first part of the curve function is the collection of arguments to be passed, also known as formals. Each curve function should have an argument `dat`, which takes a `data.frame` as described in Figure 8, as well as arguments `y` and `time` which will take character strings indicating which columns of `dat` represent the outcome and time variables, respectively. Following this is the prespecified argument `params = NULL`, which is used by `bdots` during the refitting process, where the estimated starting parameters for the function are retrieved from outside the curve fitting function. During the initial fitting process, however, these arguments are generally constructed from the observed data. The only exception to this would be if the user decided to specify the initial starting parameters for *all* subjects when calling `bfit`, as in the call

```
fit <- bfit(dat, "y", "time", curveFun = linear(intercept = 0, slope = 1),
```

however this should not be common. Following the `params` argument, any other arguments specific to the curve function could be included. Although there are none for `linear`, an example of when they might be used would be for `polynomial`, in which the degree of the polynomial to be fit would be included. Finally, there is the `...` argument, which is needed to accommodate the passing of any additional arguments from `bfit` that are not a part of the curve function. Generally, this is not needed by the users but should be included nonetheless.

2. Also included in a curve function is a second function to estimate starting parameters from the observed data. While not strictly necessary that it be included *within* the curve function, it is useful for keeping the curve function self contained; parameter estimating functions defined outside of the curve function will otherwise still be used if they exist in the users calling environment. For estimating starting parameters for a linear function we see here the function `linearPars`, taking as its arguments `dat`, `y`, and `time`. In this example, we check in case `var(y) == 0`, which causes issues for `lmer::gnls`, though in general it is a good idea to check for any other potential issues when estimating starting parameters (negative values for a logistic, for example). Importantly, this function returns a named vector, with the names of the parameters needing to match the parameter names in the formula given in (5).

3. As detailed in (1), with the argument `params = NULL`, the curve function should begin by estimating starting parameters. When different parameters are passed into `params`, this is skipped

4. This is a quick check on the result from (3). Had `linearPars` returned a `NULL` object, the curve function itself should return a `NULL` object so that it is not passed to the fitter

5. Finally, we have the most intricate part of the curve function, which is the construction of the formula object to be used by `lmer::gnls`. The first two lines of this use the base R function `str2lang` which turns a character string into an R language object (specifically, an unevaluated expression), making the names of the outcome and time variable suitable for a formula. The next line using the base R function `bquote`. The function `quote` returns its argument exactly as it was passed as an unevaluated expression; `bquote` does the same but first substituting any of its elements wrapped in `.()`. As it is written here, this will return a formula object using `slope` and `intercept` as is, but while replacing `.(y)` and `.(time)` with the appropriate names based on the columns of `dat`. Finally, the names of the parameters are included as attributes to the formula object and the curve function concludes by returning a named list including both the formula object, as well as the named vector of parameters.

The object returned by the curve function is not limited to just providing starting parameters for observed data; the formula itself is converted by `bdots` into a function proper, capable of evaluating and bootstrapping

values from that function in `bboot`. And so long as a user is able to recreate the steps provided, they should be able to construct any sort of nonlinear function to be fit to their data, even if it is not included in `bdots`.

While there is obvious utility in being able to specify *new* curves for `bdots` to fit, we describe a case here in which the flexibility of the curve function was used to recreate the `doubleGauss()` function for use with our simulated data. In short, Chapter 2 details a proposed method for fitting data in the Visual World Paradigm relying *not* on a densely sampled function in time, but rather as a collection of unordered binary observations. When the `doubleGauss` function was originally introduced to `bdots`, the empirically observed data was a relatively close match for its parametric form:

$$f(t|\theta) = \begin{cases} \exp\left(\frac{(t-\mu)^2}{-2\sigma_1^2}\right)(p-b_1)+b_1 & \text{if } t \leq \mu \\ \exp\left(\frac{(t-\mu)^2}{-2\sigma_2^2}\right)(p-b_2)+b_2 & \text{if } t > \mu \end{cases} \tag{8}$$

An example of how the observed data matched the proposed functional form is taken from an example given in Figure 10.

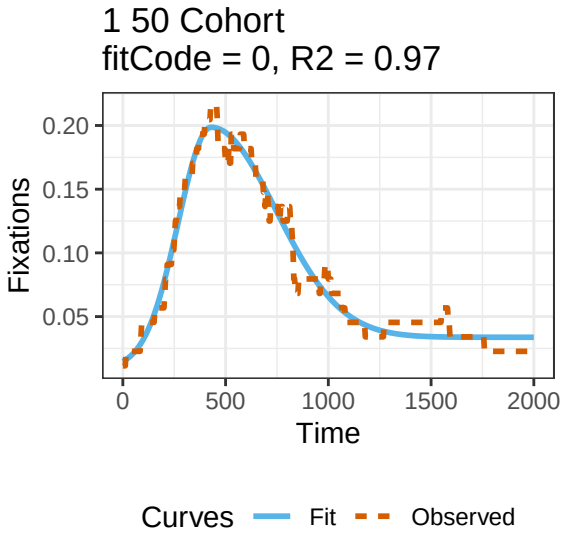

Figure 10: Example data for a four parameter logistic

Accordingly, an appropriate function for estimating the starting parameters took the form given in Figure 11.

```
dGaussPars <- function (dat, y, time) {
    time <- dat[[time]]
    y <- dat[[y]]
    if (var(y) == 0) {
        return(NULL)
    }
    mu <- time[which.max(y)]
    ht <- max(y)
    base1 <- min(y[time < mu])
    base2 <- min(y[time > mu])
    y1 <- y - base1
    y1 <- rev(y1[time <= mu])
    time1 <- rev(time[time <= mu])
    totalY1 <- sum(y1)
    sigma1 <- mu - time1[which.min(abs((pnorm(1) - pnorm(-1)) *
        totalY1 - cumsum(y1)))]
    y2 <- y - base2
    y2 <- y2[time >= mu]
    time2 <- time[time >= mu]
    totalY2 <- sum(y2)
    sigma2 <- time2[which.min(abs((pnorm(1) - pnorm(-1)) * totalY2 -
        cumsum(y2)))] - mu
    return(c(mu = mu, ht = ht, sig1 = sigma1, sig2 = sigma2,
        base1 = base1, base2 = base2))
}
```

Figure 11: Estimate starting parameters for empirical asymmetric Gaussian

This is appropriate, for example, when considering that the `mu` parameter is estimated by finding *when* the observed data is at its peak, while the `ht` parameter is found to be the peak itself. In the case of the look onset method, however, data consists of non-ordered binary observations $\{0, 1\}$ in time, making estimates for even these two parameters completely unreliable. An immediate consequence of this was that in a simulation of 1,000 subjects fit with asymmetric Gaussian data was conducted, less than half were able to return adequate fits from `bdots`, proving problematic for any kind of systematic analysis in assessing the proposed method.

The solution, of course, was to provide a new curve function utilizing a different interval function for estimating starting parameters. The particular interest in presenting this here is how broad of a solution this may be to any number of problems: rather than attempting to construct parameters directly from the data (which may be misbehaved), we provide a reasonable distribution of starting parameters, drawing any number of samples from it, and retaining those which best fit the observed data:

```
dgaussPars_dist <- function(dat, y, time, startSamp = 8) {
        time <- dat[[time]]
        y <- dat[[y]]

        if (var(y) == 0) {
            return(NULL)
        }

        spars <- data.table(param = c("mu", "ht", "sig1", "sig2", "base1", "base2"),
                            mean = c(630, 0.18, 130, 250, 0.05, 0.05),
                            sd = c(77, 0.05, 30, 120, 0.015, 0.015),
                            min = c(300, 0.05, 50, 50, 0, 0),
                            max = c(1300, 0.35, 250, 400, 0.15, 0.15))

        fn <- function(p, t) {
            lhs <- (t < p[1]) * ((p[2] - p[5]) *
                       exp((t - p[1])^2/(-2 * p[3]^2)) + p[5])
            rhs <- (t >= p[1]) * ((p[2] - p[6]) *
                       exp((t - p[1])^2/(-2 * p[4]^2)) + p[6])
            lhs + rhs
        }
        npars <- vector("list", length = startSamp)
        for (i in seq_len(startSamp)) {
            maxFix <- Inf
            while (maxFix > 1) {
                npars[[i]] <- Inf
                while (any(spars[, npars[[i]] <= min | npars[[i]] >= max])) {
                  npars[[i]] <- spars[, rnorm(length(npars[[i]])) * sd + mean]
                }
                maxFix <- max(fn(npars[[i]], time))
            }
        }
        r2 <- vector("numeric", length = startSamp)
        for (i in seq_len(startSamp)) {
            yhat <- fn(npars[[i]], time)
            r2[i] <- mean((y - yhat)^2)
        }
        finalPars <- npars[[which.min(r2)]]
        names(finalPars) <- c("mu", "ht", "sig1", "sig2", "base1", "base2")
        return(finalPars)
    }
```

Figure 12: Using random distribution to estimate starting parameters

By utilizing an existing fitting function while substituting the method by which the starting parameters are estimated, we were able to go from recovering less than half of the simulated starting parameters successfully to well over 80%.