Reintroduction of the `bdots` R package: Methodological and syntactical changes to the bootstrapped differences of timeseries in the context of the Visual World Paradigm

Collin Nolte[a,*], Patrick Breheny[b]

[a] *Dept. of Statistics, Grinnell College, United States*
[b] *Dept. of Biostatistics, University of Iowa, United States*

Abstract

In 2018, Seedorff et al. [1] introducts the `bdots` package to CRAN, an implementation of the bootstrapped differences in timeseries methodology first introduced by Oleson et al. [2]. The present manuscript expands on this work by introducing two critical changes to the `bdots` package: first, it corrects a methodological issue in the original method that drastically underestimates the Type I error rate; alongside this correction we introduce a new permutation method for identifying differences in timeseries. Second, the entire package has been rewritten to generalize existing functions and accomodate a new collection of tools including automated paired testing, an interactive refitting process, and the ability to define arbitrary curve fitting functions to the observed data. Examples of this new syntax along with illustrations of the added functionality are included. We conclude with a discussion of the present state of the package along with potential avenues for future work.

1. Introduction

In 2017, Oleson et al. [2] introduced the "boostrapped differences in timeseries", a method for detecting time-specific differences in the trajectory of outcomes between experimental groups. Accompanying this was a novel method for controlling the family-wise error rate, particularly in the case of densely sampled time series where constructed test statistics exhibit high degrees of autocorrelation. In 2018, this methodology was instantiated with the introduction of the `bdots` package to CRAN [1]. Here we reintroduce the `bdots` package, both with a significant correction to the underlying methodology as well as a number of upgrades that greatly expand upon 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 an outline of the updated methodology and a general demonstration of package use. The most significant changes include:

1. Major corrections to underlying methodology with implications for prior users of the package
2. Introduction of permutation methods
3. Simplified user interface
4. Introduction of user defined curves
5. Allow fitting an arbitrary number of groups at once
6. Automatic detection of paired tests based on subject identifier

*Corresponding author
Email address: `noltecollin@grinnell.edu` (Collin Nolte)

7. Allows for non-homogeneous sampling of time points across subjects and groups
8. Introduction of formula syntax for bootstrapping difference function
9. Introduction of interactive refitting process

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

Bootstrapped differences in time series. A typical problem in the analysis of differences in time series involves that of two or more experimental groups containing subjects whose responses are measured over time. In particular, we concern ourselves with the identification of which, if any, of the arbitrarily many points observed witin two groups may differ. Concomitant with this problem is one of multiple comparisons: often these timeseries take the form of dense, highly correlated observations. Examples of these include the growth of tumors in mice or in the changes in proportions of eye fixations over time such as in the Visual World Paradigm which served as the original catalyst for this methodology. In either case, we begin with the assumption that each of the subjects $i = 1, \ldots, n$ within an experimental group has observed data of the form

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

where f represents a functional mean structure and the error term ϵ_{it} is assumed to be either IID or to possess an AR(1) structure. At present, `bdots` requires that each of the subjects being compared have the same parametric function f, though this is not strictly necessary at the theoretical level and future goals of the package include accommodating non-parametric mean structures. While the mean structures for each of the subjects is required to be of the same parametric form $f(\cdot|\theta)$, each differs in their instance of 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

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

The distribution of parameters θ is then used to construct a distribution of functions, $f(\cdot|\theta)$, giving 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 accomplish.

2. Methodology and Overview

A standard analysis using `bdots` consists primarily of two steps: using the observed data to estimate group-level distributions of time series and performing hypothesis testing to identify statistically significant differences between them. We will first introduce an updated algorithm for estimating these distributions, followed by a discussion of various methods used to identify temporal differences.

2.1. Estimating Group Distributions

We begin with the assumption that for subject i in group g, we have collected observed data of the form given in Eq. 1, with the subject specific parameter θ_i following the distribution in Eq. 2. Each subject is then fit in `bdots` via the nonlinear curve fitting function `nlme::gnls` [3], returning for each set of observed data an estimated set of parameters $\hat{\theta}_i$ and their associated standard errors. From these estimates we are able to construct a sampling distribution for each subject:

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

Just as in Oleson et al. [2], this distribution provides an estimate of the within-subject uncertainty in the estimate of subject-specific function parameters.

The estimation of the group-level distribution given in Eq. 2 is then estimated with a modified bootstrapping procedure where, in addition to random sampling subjects within a group, subject-specific parameters are drawn from the estimated distribution in Eq 3. This permits two distinct sources of variability to be

present in our estimate of group distributions: the variability inherent to the distribution of the group and the uncertainty present in the estimation of individual subject parameters. With this in mind, we propose the following updated algorithm for creating bootstrapped estimates of group level distributions:

1. For each subject, fit a nonlinear curve to the functional mean structure given in Eq. 1 to derive estimates for the subject-specific distribution given in Eq. 3
2. For a group of size n, select n subjects from the group *with replacement*
3. For each selected subject i in bootstrap b, draw a set of parameters from the subject-specific distribution derived from the curve fitting process in step (1),

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

4. Find the mean across all of the bootstrapped θ^*_{ib} in group g to construct the bth group bootstrap, θ^*_{gb} where, by the law of total variance, we have

$$\theta^*_{gb} = \frac{1}{n}\sum \theta^*_{ib}, \qquad \theta^*_{gb} \sim N\left(\mu_g, \frac{1}{n}V_g + \frac{1}{n^2}\sum s^2_i\right). \tag{5}$$

   That is, the bootstrapped estimate follows a *group level* distribution with an additional variance term to account for uncertainty in model estimates
5. Perform steps (2)-(4) B times, using each θ^*_{gb} to construct a sample of population curves, $f(\cdot|\theta^*_{gb})$.

The algorithm presented here differs from the original iteration of `bdots` in a critical way. In Oleson et al. [2], there is no assumption of the between-subject variability presented in Eq. 2, congruent with the assumption that $\theta_i = \theta_j$ for all subjects i, j within a group. This occurs because of the omission of step (2) in the algorithm above, where instead each subject is included in each bootstrapped sample making the only variance accounted for is that contributed by the within-subject variability. The most relevant consequence of this assumption is that bootstrapped parameter estimates from Eq. 5 fail to account for the $\frac{1}{n}V_g$ term in the variance which empirically tends to be the larger of the two. In practical terms, this results in a drastic inflation of the Type I error rate. A second, smaller difference presented here is that our algorithm no longer imposes an assumption of an AR(1) error structure by default, though the option to do so is still present in the `bdots` package.

For now, the final population curves constructed in Step (5) of our algorithm can be used to create estimates of the mean response and associated standard errors at each time point for each of the groups bootstrapped. These estimates are used both for plotting and in the construction of confidence intervals.

2.2. Hypothesis Testing for Statistically Significant Differences in Time Series

We now turn our attention to the primary goal of an analysis in `bdots`, the identification of specific time points in which the distribution of curves of two groups differ in statistically significant ways. An intuitive method for doing so involves testing differences of means at each observed time point; though, in the case of a densely sampled time series, we must take into account the issue of multiple testing to control the family-wise error rate (FWER). This problem is addressed by `bdots` in two ways: making adjustments to the nominal α value and through permutation testing, the details of each are presented below.

2.2.1. α Adjustment

The first method whereby we control the FWER involves making adjustments to the nominal alpha value, similar to whah is done in a standard Bonferonni correction. 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 test statistics from the bootstrapped estimates, $T^{(b)}_t$, can be written as

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

where $\overline{p}_{gt}$ and s_{gt} are mean and standard deviation estimates of the estimated functions at each time point t and for groups 1 and 2, respectively. These statistics can be highly correlated in the presence of densely sampled time series, leading to an inflated Type I error. The FWER in this case can be controlled with the adjustment proposed in Oleson et al. [2], which indeed does simplify to a Bonferonni correction in the event that the observations are found to be independent. Additionally, `bdots` allows for any of the adjustments present in the `p.adjust` function from the R `stats` packages, i.e., Benjamini & Hochberg, FDR, etc.,.

2.2.2. Permutation Testing

In addition to modified correction based on the bootstrapped test statistics, `bdots` provides a permutation test for controlling the FWER. 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 $f(t|\hat{\theta}_i)$ (optionally, the user can select to redraw $\hat{\theta}_i$ from Eq. 3. to account for within-subject variability within each permutation). Within each group, we use these curves to create estimates of the mean population curves and associated standard errors at each point[1]. Letting p_{gt} and s_{gt} represent the mean population curve and standard error for group g at time t, we define our observed permutation test statistic,

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

.

We then going about using permutations to construct a null distribution against which to compare the observed statistics from Eq. 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) to create two new groups
3. Recalculate the test statistics $T_t^{(p)}$, 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 within each bootstrap the same subjects are selected for each group or by ensuring that each permuted group contains one observation from each subject.

In summary, two methodological changes have been implemented in this current version of `bdots`: first and most critically, we have corrected the original underlying algorithm to include estimates of the *between-subject* variability in the distribution of group curves, again with important implications for prior users of the package who may find that their Type I error rates have been severely understated. Additionally, we have introduced a permutation method for identifying temporal-specific differences between groups that requires fewer assumptions about the data generating process. A thorough exploration of the power and FWER of each of these methods, in both the paired and unpaired cases, is presented in need to arxiv submit ch4 of dissertation.

3. Package Use

The remainder of this paper will detail the implementation of each of these methods using the `bdots` package. The two major components of a typical analysis using the `bdots` package are curve fitting and the identification of statistically significant differences in time series. Within each of these components, we will begin with a high-level explanation of the major changes that have been made, along with a detailed description of the new syntax. We then contextualize this discussion with a real-world example along with illustrations of function calls and return objects.

[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 $f\left(t|\frac{1}{n}\sum\theta_i\right)$

3.1. Curve Fitting

The first step in performing an analysis with `bdots` involves specifying the parametric function which defines the mean structure from Eq. 1 and fitting curves to the observed data for each subject. Throughout this discussion and into the next section, we will use as our real world example a comparison of tumor growth for the 451LuBr cell line in mice with repeated measures across five treatment groups [4]. A depiction of this data is given in Figure 1:

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

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. For example, consider the `Day` column for our mouse data shown in Figure 1, where the first four observations for ID 1 are all different than those for ID 2. For the present analysis, we will be 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 the topic of this section.

The `bfit` Function. The curve fitting process is performed with the function `bfit`, taking the following arguments:

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

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

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. It is important to note here that the identification of paired data is now done automatically based on the subject identifiers in compared groups; as such, it will be important for the user to be sure that if the data is paired that the subject identifiers are the same between subject groups.

The `time` and `y` arguments are column names of the time variable and observed outcome, respectively. The `group` argument takes 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. `ar` is a boolean argument indicating whether or not the observed data should be fit with an AR(1) assumption. Lastly, the `curveFun` argument is used to specify details of the parametric function to which the data will be fit.

`curveFun` 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 independently of the fitting function by passing an argument to `bfit`. Unlike the previous arguments which took either a data.frame or character vector, `curveFun` takes as an argument

a *function call*, for example, `logistic()`. In short, this allows the user to pass additional arguments to further specify the curve at the time the `bfit` function is called. To briefly illustrate, among the parametric functions now included in `bdots` is 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 pass the argument `curveFun = polynomial(degree = 5)`. This permits us to further qualify aspects of the `polynomial()` function at the time the function is declared. Curve functions currently included in `bdots` are `logistic()`, `doubleGauss()`, `expCurve()`, and `polynomial()`. In addition to the functions provided by default in the `bdots` package, `bfit` can also accept user-created curves; a detailed explanation of how this is done is provided in the appendix as well as with `vignette("bdots")`.

Using our mouse data with the columns shown in Figure 1, we are ready to fit curves to each of the subjects. For this analysis we will fit data to an exponential curve of the form

$$f(t|\theta) = x_0 e^{tk}, \tag{8}$$

where $\theta = [x_0, k]$. This form is specified in the `expCurve()` provided by the `bdots` package. The syntax for calling `bfit` with the mouse data is as follows:

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

Having successfully fit curves to our data, we now consider the return object and provided summary functions.

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 combination of subject and group values. Included in this row are the subject identifier, group classification, a nested `gnls` object, and summary statistics and fitting diagnostics regarding the quality of fit for the curves. Inheriting from `data.table` also permits us to use `data.table` syntax to subset the object as is illustrated in Figure 5 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.table`

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

Fitting Diagnostics. 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 Eq. 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 are times when the fitter is incapable of converging on a solution. In that instance, the autocorrelation assumption is dropped and constructing a fit is 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 derived. `fitCode` is a numeric summary statistic ranked from 0 to 6 detailing information about the quality of the fitted curve, constructed with the following logic:

```
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. Additionally, the methodological changes made in this updated version and the introduction of permutation testing makes `bdots` far more robust to the misidentification of autocorrelationin the identification of statistically significant differences. Although the current iteration of `bdots` continues to utilize this discrete scale in the assessment of fit quality, the creation of greater flexibility is a priority for future directions of the package.

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

Table 1: Description of the `fitCode` statistic

While the fit code offers a simple diagnostic for assessing quality of individual subjects, it will often be useful to consider broader summaries for reporting on the quality of fits for groups as a whole. This is done most simply using the `summary` and `plot` functions.

Summaries and Plots. Users are able to quickly summarize the quality of the fits with the `summary` method now provided. For example, we may consider a summary of the fitted mouse data:

```
> summary(mouse_fit)

bdotsFit Summary

Curve Function: 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. 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 fits as `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, ])
```

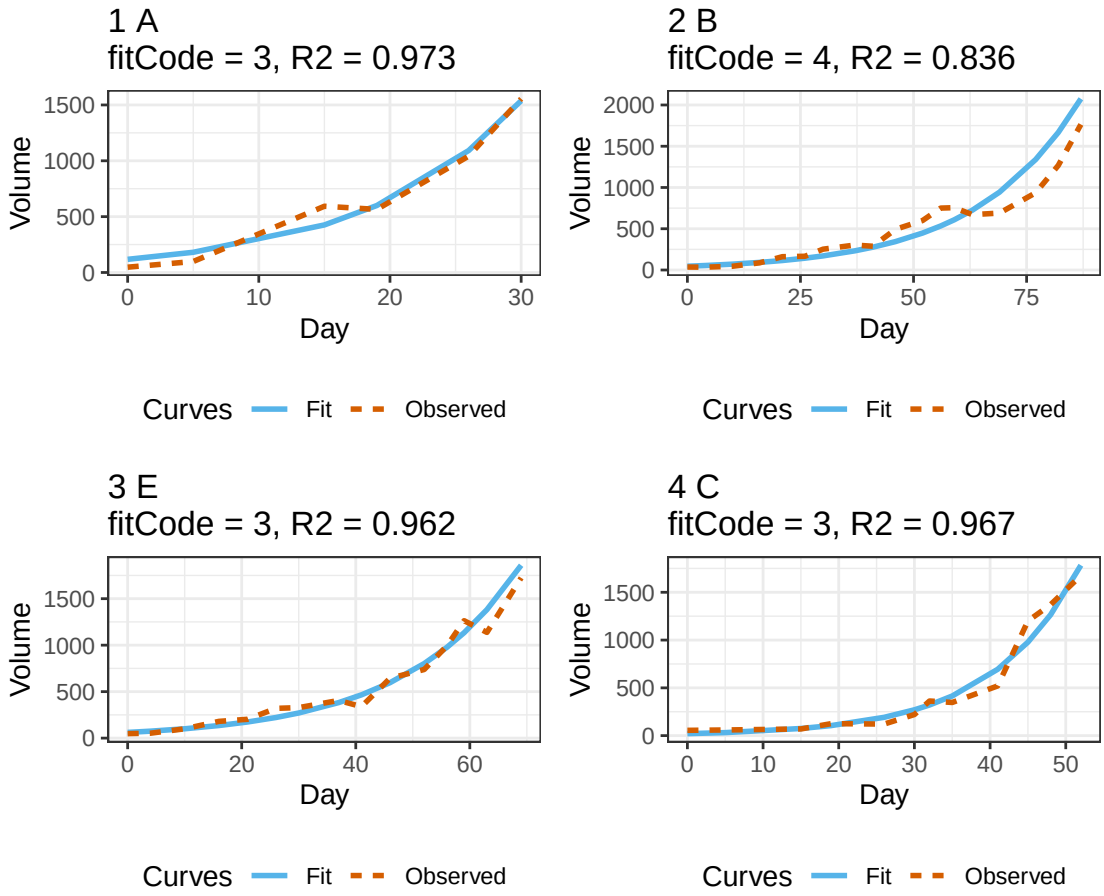

Figure 5: Plot of `mouse_fit` using `data.table` syntax to subset to only the first four observations

`bdots` provides now an interactive refitting function, `brefit`, which provides a number of options to help users recalibrate low quality fits. Details on this function and how it is used are provided in Section 3.3.

3.2. Identification of Group Differences

Having satisfactorily fit subject-specific parametric curves to the observed data, we are ready to begin estimating the group distributions and investigating temporal differences. This section introduces the function `bboot`, along with the introduction of a formula syntax that is new and unique to `bdots`. As before, we will follow each of our descriptions with an illustration of use with the mouse tumor data. Following this, we summarize new generics that are available to the object returned by `bboot`, including those for the `summary` and `plot` functions.

The `bboot` Function. The number of options included in the `bboot` function have expanded to include a new formula syntax for specifying the groups we wish to compare, as well as an option to use permutation testing for identifying differences. 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 as `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:

$$\texttt{y} \sim \texttt{group(val1, val2)}$$

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 given as `Volume` for A and B from the `Treatment` column in our mouse data (Figure 1). We would do so with the following syntax:

$$\texttt{Volume} \sim \texttt{Treatment(A, B)}$$

The formula syntax is robust to a number of situations not present in our example tumor data, including situations that arise in the case of multiple and/or nested groups and when considering a difference in differences analysis. Each of these is expounded upon in the supplementary materials.

Summary and Analysis. Let's begin first by running `bboot` 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)
```

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

```
> summary(mouse_boot)

bdotsBoot Summary

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

Difference of difference: FALSE
Paired t-test: FALSE
Difference: Treatment (A,E)

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. 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 if 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).

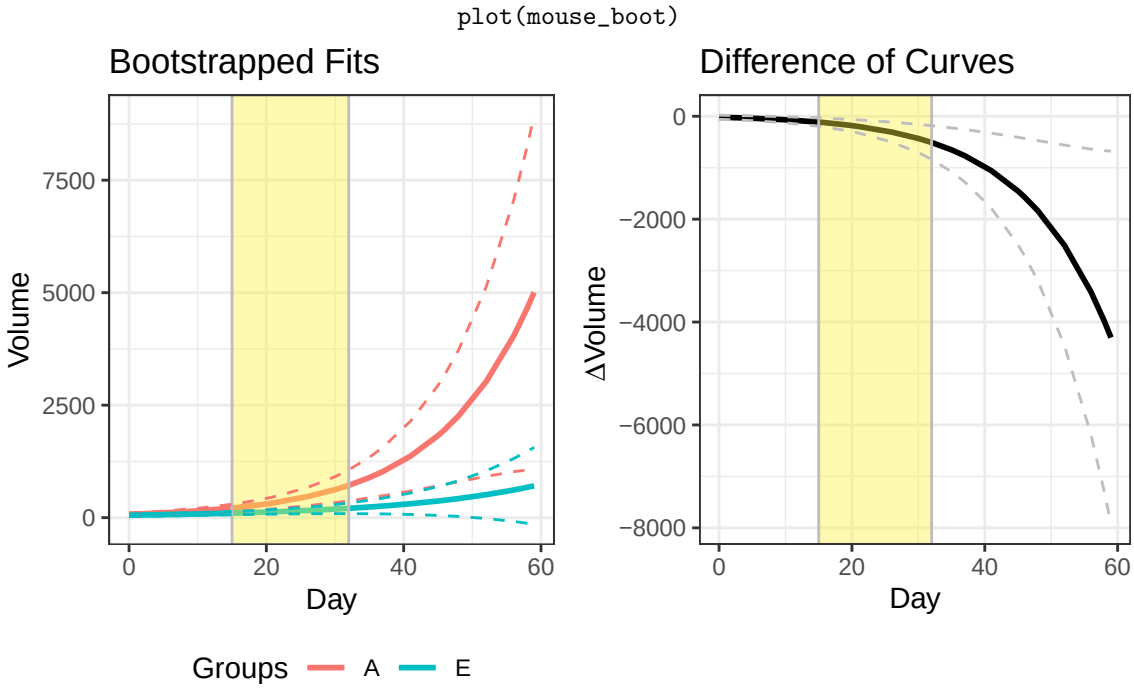

Figure 6: Bootstrapped distributions with regions of significant difference determined via permutation testing

3.3. Ancillary Functions

Outside of a standard analysis using the bdots package, there are a suite of additional functions that users may find helpful. Brief descriptions of these functions are given here.

Refitting. The nonlinear curve fitting algorithm used by nlme::gnls in bfit can be sensitive to the 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 fits can often be evidenced by the fitCode or via visual inspections of the fitted functions against the observed data. Occasionally, the quality of these fits will be poor. When this occurs, there are several options available to the user, all of which are provided through the function brefit. 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 a sub-optimal way to specify data to subset. To add flexibility to which subjects are fit there is now the subset argument taking either a logical expression, a 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` argument is completely ignored.

Assisting 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 for the refitted curve with new parameters. This offeres a great deal of utility by automatically attempting to refit curves without requiring additional user input. 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 ID 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 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 fourth option gives the user the ability to attempt 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 option (6) 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 as in Figure 7.

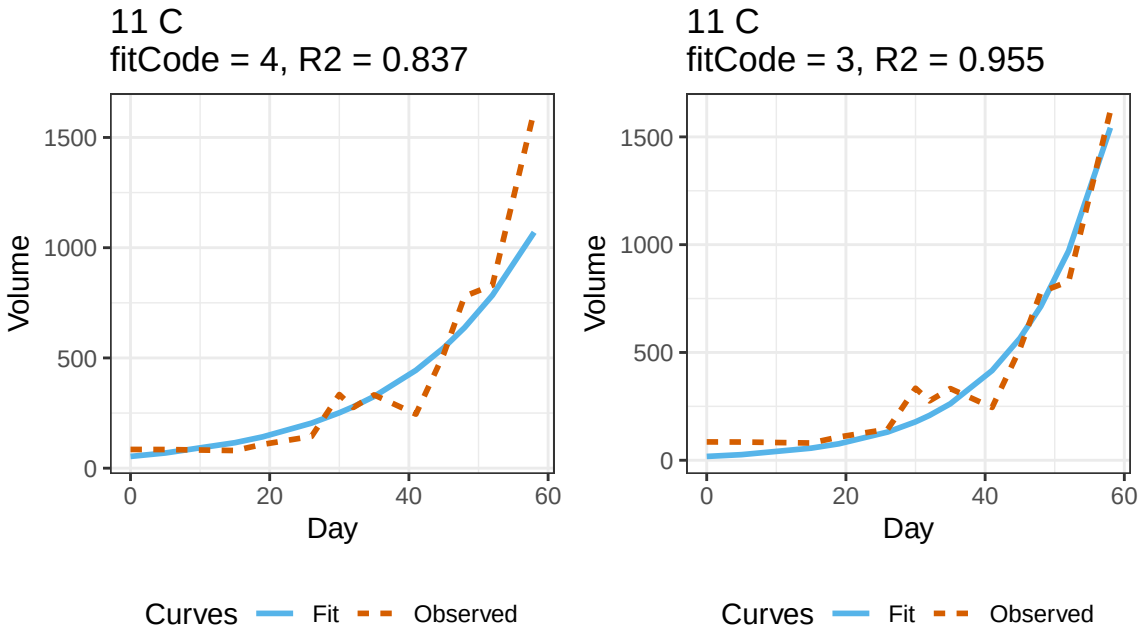

Figure 7: Presentation of curves before and after refitting, with the original given on the left

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 refitted 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 they are also removed from all additional groups in order to retain paired status. When paired subjects do exist, `brefit` provides a final prompt to the user before removing them. 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.

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 [2], 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`

4. 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 an 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 tighter 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 assessing 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.

Appendix

5. Extended Formula Syntax

There are two special cases to consider when using the new `bboot` formula syntax. The first is the situation that arises in the case of multiple or nested groups, the second when considering the "difference of differences" between two gruops. We will describe these in the next section, though unfortunately the mouse data being used to provide illustration to the package use does not naturally accommodate either of these extensions. As such, we will begin by briefly introducing a toy data structure and then using it to illustrate the extensions of the syntax.

Formula Syntax for Nested Groups and the 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 is the case when we compare 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). 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)
```

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 method [5]. 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. 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.

6. 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 simplify 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

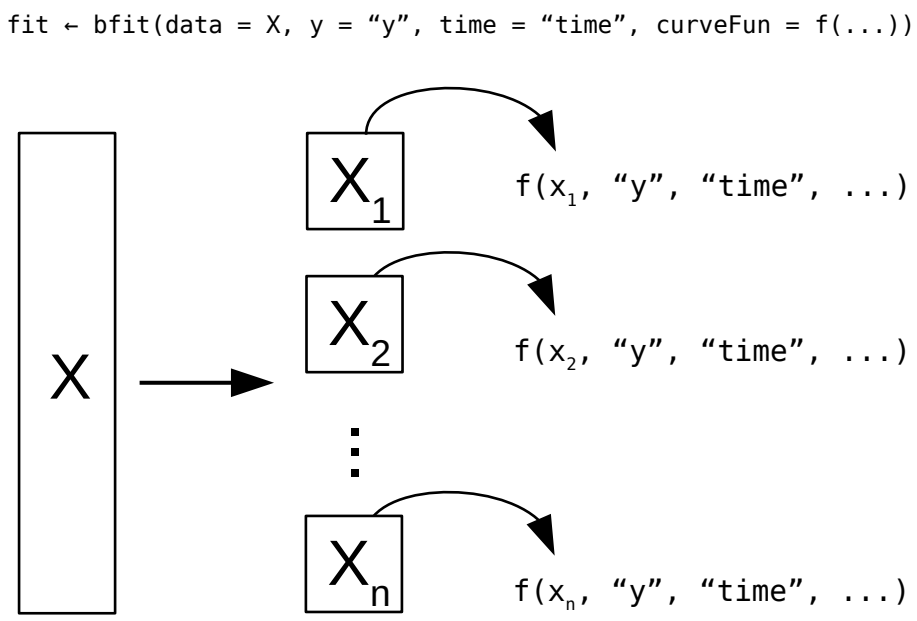

Figure 8: A call to the function `bfit` with data `X` and outcome and time variables `"y"` and `"time"`. `bfit` splits the dataset `X` by subject/group and passes each individual `data.frame` into the curve function `f()`, along with time and outcome character vectors as well as any other arguments passed 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.

their own needs. In this section we will detail how the curve function is used in `bdots` and how users can write their own.

To begin, it is important to understand 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 (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 `nlme::gnls`. Doing so requires both (1) a formula to which we fit the data and (2) starting parameter estimates. Providing the both of these is the role of the curve function. Figure 8 provides an illustration of this process.

We offer an explicit example of a curve function by constructing a curve function for fitting a straight line to the observed data. Curve functions require a minimum set of components, which we describe in enough detail to be suitable as a template. This function, along with its enumerated components, are given 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 formals, or the collection of arguments to be passed to the function. 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 parameters 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(params = c(intercept = 0, slope = 1))
   ```

   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 specified. 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 user's 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 `nlme::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). In the event of a problem, we return `NULL` so that the subject is ignored by the fitter (this does not preempt attempting to manually refit with `brefit`). 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 `nlme::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, replacing `.(y)` and `.(time)` with the appropriate names based on the character vectors provided in the function's formals. 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 function to be fit to their data, even if it is not included in `bdots`.

References

[1] Michael Seedorff, Jacob Oleson, and Bob McMurray. Detecting when timeseries differ: Using the bootstrapped differences of timeseries (bdots) to analyze visual world paradigm data (and more). *Journal of memory and language*, 102:55–67, 2018.

[2] Jacob J Oleson, Joseph E Cavanaugh, Bob McMurray, and Grant Brown. Detecting time-specific differences between temporal nonlinear curves: Analyzing data from the visual world paradigm. *Statistical methods in medical research*, 26(6): 2708–2725, 2017.

[3] José Pinheiro, Douglas Bates, and R Core Team. *nlme: Linear and Nonlinear Mixed Effects Models*, 2025. URL `https://CRAN.R-project.org/package=nlme`. R package version 3.1-167.

[4] Mengshi Li, Dijie Liu, Dongyoul Lee, Somya Kapoor, Katherine N. Gibson-Corley, Thomas P. Quinn, Edwin A. Sagastume, Sarah L. Mott, Susan A. Walsh, Michael R. Acevedo, Frances L. Johnson, and Michael K. Schultz. Enhancing the efficacy of melanocortin 1 receptor-targeted radiotherapy by pharmacologically upregulating the receptor in metastatic melanoma. *Molecular Pharmaceutics*, 16(9):3904–3915, July 2019. ISSN 1543-8392. doi: 10.1021/acs.molpharmaceut.9b00512.

[5] Bob McMurray, Jamie Klein-Packard, and J. Bruce Tomblin. A real-time mechanism underlying lexical deficits in developmental language disorder: Between-word inhibition. *Cognition*, 191:104000, oct 2019. doi: 10.1016/j.cognition.2019.06.012.