Diving Behaviour Analysis in R[*]

# An Introduction to the `diveMove` Package

Sebastián P. Luque[†]

## Contents

## 1 Introduction

Remarkable developments in technology for electronic data collection and archival have increased researchers' ability to study the behaviour of aquatic animals while reducing the effort involved and impact on study animals. For example, interest in the study of diving behaviour led to the development of minute time-depth recorders (TDRs) that can collect more than 15 MB of data on depth, velocity, light levels, and other parameters as animals move through their habitat. Consequently, extracting useful information from TDRs has become a time-consuming and tedious task. Therefore, there is an increasing need for efficient software to automate these tasks, without compromising the freedom to control critical aspects of the procedure.

There are currently several programs available for analyzing TDR data to study diving behaviour. The large volume of peer-reviewed literature based on results from these programs attests to their usefulness. However, none of them are in the free software domain, to the best of my knowledge, with all the disadvantages it entails. Therefore, the main motivation for writing `diveMove` was to provide an R package for diving behaviour analysis allowing for more flexibility and access to intermediate calculations. The advantage of this approach is that researchers have all the elements they need at their disposal to take the analyses beyond the standard information returned by the program.

The purpose of this article is to outline the functionality of `diveMove`, demonstrating its most useful features through an example of a typical diving behaviour analysis session. Further information can be obtained by reading the vignette that is included in the package (`vignette("diveMove")`) which is currently under development, but already shows basic usage of its main functions. `diveMove` is available from `CRAN`, so it can easily be installed using `install.packages()`.

## 2 Features

`diveMove` offers functions to perform the following tasks:

---

[*]An earlier version of this introduction to `diveMove` has been published in R News (Luque 2007)

[†]Contact: `spluque@gmail.com`. Comments for improvement are very welcome!

- Identification of wet vs. dry periods, defined by consecutive readings with or without depth measurements, respectively, lasting more than a user-defined threshold. Depending on the sampling protocol programmed in the instrument, these correspond to wet vs. dry periods, respectively. Each period is individually identified for later retrieval.

- Calibration of depth readings, which is needed to correct for shifts in the pressure transducer. This can be done using a `tcltk` graphical user interface (GUI) for chosen periods in the record, or by providing a value determined a priori for shifting all depth readings.

- Identification of individual dives, with their different phases (descent, bottom, and ascent), using various criteria provided by the user. Again, each individual dive and dive phase is uniquely identified for future retrieval.

- Calibration of speed readings using the method described by Blackwell et al. (1999), providing a unique calibration for each animal and deployment. Arguments are provided to control the calibration based on given criteria. Diagnostic plots can be produced to assess the quality of the calibration.

- Summary of time budgets for wet vs. dry periods.

- Dive statistics for each dive, including maximum depth, dive duration, bottom time, post-dive duration, and summaries for each dive phases, among other standard dive statistics.

- `tcltk` plots to conveniently visualize the entire dive record, allowing for zooming and panning across the record. Methods are provided to include the information obtained in the points above, allowing the user to quickly identify what part of the record is being displayed (period, dive, dive phase).

Additional features are included to aid in analysis of movement and location data, which are often collected concurrently with *TDR* data. They include calculation of distance and speed between successive locations, and filtering of erroneous locations using various methods. However, `diveMove` is primarily a diving behaviour analysis package, and other packages are available which provide more extensive animal movement analysis features (e.g. `trip`).

The tasks described above are possible thanks to the implementation of three formal S4 classes to represent TDR data. Classes *TDR* and *TDRspeed* are used to represent data from TDRs with and without speed sensor readings, respectively. The latter class inherits from the former, and other concurrent data can be included with either of these objects. A third formal class (*TDRcalibrate*) is used to represent data obtained during the various intermediate steps described above. This structure greatly facilitates the retrieval of useful information during analyses.

# 3 Preliminary Procedures

As with other packages in R, to use the package we load it with the funtion `library`:

```
> library(diveMove)
```

This makes the objects in the package available in the current R session. A short overview of the most important functions can be seen by running the examples in the package's help page:

```
> example(diveMove)
```

## Data Preparation

`TDR` data are essentially a time-series of depth readings, possibly with other concurrent parameters, typically taken regularly at a user-defined interval. Depending on the instrument and manufacturer, however, the files obtained may contain various errors, such as repeated lines, missing sampling intervals, and invalid data. These errors are better dealt with using tools other than R, such as `awk` and its variants, because such stream editors use much less memory than R for this type of problems, especially with the typically large files obtained from `TDR`s. Therefore, `diveMove` currently makes no attempt to fix these errors. Validity checks for the TDR classes, however, do test for time series being in increasing order.

Most `TDR` manufacturers provide tools for downloading the data from their `TDR`s, but often in a proprietary format. Fortunately, some of these manufacturers also offer software to convert the files from their proprietary format into a portable format,

such as comma-separated-values (csv). At least one of these formats can easily be understood by R, using standard functions, such as `read.table()` or `read.csv()`. diveMove provides constructors for its two main formal classes to read data from files in one of these formats, or from simple data frames.

# 4 How to Represent TDR Data?

*TDR* is the simplest class of objects used to represent TDR data in diveMove. This class, and its *TDR-speed* subclass, stores information on the source file for the data, the sampling interval, the time and depth readings, and an optional data frame containing additional parameters measured concurrently. The only difference between *TDR* and *TDRspeed* objects is that the latter ensures the presence of a speed vector in the data frame with concurrent measurements. These classes have the following slots:

**file:** character,

**dtime:** numeric,

**time:** POSIXct,

**depth:** numeric,

**concurrentData:** data.frame

Once the TDR data files are free of errors and in a portable format, they can be read into a data frame, using e.g.:

```
> ff <- system.file(file.path("data",
+     "dives.csv"), package = "diveMove")
> tdrXcsv <- read.csv(ff, sep = ";")
```

and then put into one of the *TDR* classes using the function `createTDR()`. Note, however, that this approach requires knowledge of the sampling interval and making sure that the data for each slot are valid:

```
> ddtt.str <- paste(tdrXcsv$date,
+     tdrXcsv$time)
> ddtt <- strptime(ddtt.str,
+     format = "%d/%m/%Y %H:%M:%S")
> time.posixct <- as.POSIXct(ddtt,
+     tz = "GMT")
> tdrX <- createTDR(time = time.posixct,
+     depth = tdrXcsv$depth,
+     concurrentData = tdrXcsv[,
+         -c(1:3)], dtime = 5,
+     file = ff)
> tdrX <- createTDR(time = time.posixct,
+     depth = tdrXcsv$depth,
```
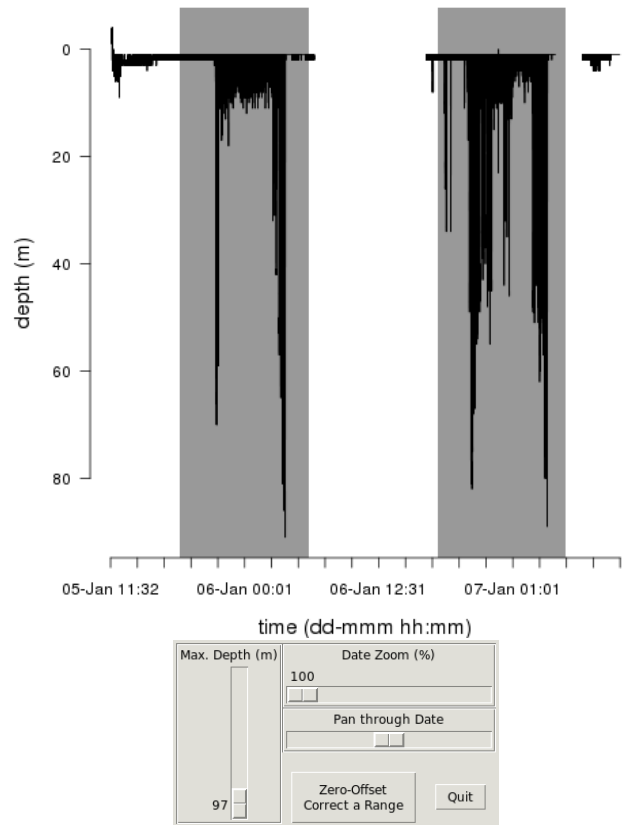


**Figure 1.** The *plotTDR()* method for *TDR* objects produces an interactive plot of the data, allowing for zooming and panning.

```
+     concurrentData = tdrXcsv[,
+         -c(1:3)], dtime = 5,
+     file = ff, speed = TRUE)
```

If the files are in *.csv format, these steps can be automated using the `readTDR()` function to create an object of one of the formal classes representing TDR data (*TDRspeed* in this case), and immediately begin using the methods provided:

```
> tdrX <- readTDR(ff, speed = TRUE,
+     sep = ";", na.strings = "",
+     as.is = TRUE)
> plotTDR(tdrX)
```

Several arguments for `readTDR()` allow mapping of data from the source file to the different slots in diveMove's classes, the time format in the input and the time zone attribute to use for the time readings.

Various methods are available for displaying TDR objects, including `show()`, which provides an informative summary of the data in the object, extractors and replacement methods for all the slots. There is a `plotTDR()` method (Figure 1) for both

*TDR* and *TDRspeed* objects. The *interact* argument allows for suppression of the `tcltk` interface. Information on these methods is available from *methods?TDR*.

*TDR* objects can easily be coerced to data frame (`as.data.frame()` method), without losing information from any of the slots. *TDR* objects can additionally be coerced to *TDRspeed*, whenever it makes sense to do so, using an *as.TDRspeed()* method.

# 5 Identification of Activities at Various Scales

One of the first steps of dive analysis is to identify dry and wet periods in the record. This is done with function `calibrateDepth()`. Wet periods are those with depth readings, dry periods are those without them. However, records may have aberrant missing depth that should not define dry periods, as they are usually of very short duration[1]. Likewise, there may be periods of wet activity that are too short to be compared with other wet periods, and need to be excluded from further analyses. These aspects can be controlled by setting the arguments *dry.thr* and *wet.thr* to appropriate values.

The next step involves correcting depth for shifts in the pressure transducer, so that surface readings correspond to zero. Such shifts are usually constant for an entire deployment period, but there are cases where the shifts vary within a particular deployment, so shifts remain difficult to detect and dives are often missed. Therefore, a visual examination of the data is often the only way to detect the location and magnitude of the shifts. Visual adjustment for shifts in depth readings is tedious, but has many advantages which may save time during later stages of analysis. These advantages include increased understanding of the data, and early detection of obvious problems in the records, such as instrument malfunction during certain intervals, which should be excluded from analysis.

Function `calibrateDepth()` takes a *TDR* object to perform three basic tasks: 1. identify wet and dry periods, 2. zero-offset correct (ZOC) the data, and 3. identify all dives in the record and their phases.

---

[1]They may result from animals resting at the surface of the water long enough to dry the sensors.

ZOC can be done using one of three methods: "visual", "offset", and "filter". The first one ("visual") is the default method, which let's the user perform the correction interactively, using the `tcltk` package:

```
> dcalib <- calibrateDepth(tdrX)
```

This command brings up a plot with `tcltk` controls allowing to zoom in and out, as well as pan across the data, and adjust the `depth` scale. Thus, an appropriate time window with a unique surface depth value can be displayed. This allows the user to select a `depth` scale that is small enough to resolve the surface value using the mouse. Clicking on the ZOC button waits for two clicks: i) the coordinates of the first click define the starting time for the window to be ZOC'ed, and the depth corresponding to the surface, ii) the second click defines the end time for the window (i.e. only the x coordinate has any meaning). This procedure can be repeated as many times as needed. If there is any overlap between time windows, then the last one prevails. However, if the offset is known a priori, method "offset" lets the user specify this value as the argument *offset* to `calibrateDepth()`. For example, preliminary inspection of object `tdrX` would have revealed a 3 m offset, and we could have simply called (without plotting):

```
> dcalib <- calibrateDepth(tdrX,
+     zoc.method = "offset",
+     offset = 3)
```

A third method ("filter") implements a smoothing/filtering mechanism where running quantiles can be applied to depth measurements sequentially, using `.depth.filter`. It relies on the `caTools` package. This method is still under development, but reasonable results can already be achieved by applying two filters, the first one using a running median with a narrow window to denoise the time series, followed by a running low quantile using a wide time window. The integer vector given as argument $k$ specifies the width of the moving window(s), where $k_i$ is the width for the $i^{th}$ filter in units of the sampling interval of the *TDR* object. Similarly, the integer vector given as argument *probs* specifies the quantile for each filter, where $probs_i$ is the quantile for the $i^{th}$ filter. Smoothing/filtering can be performed within specified minimum and maximum depth bounds using argument *depth.bounds*[2],

---

[2]Defaults to the depth range

in cases where surface durations are relatively brief separated by long periods of deep diving. These cases usually require large windows, and using depth bounds helps to stabilize the surface signal.

```
> dcalib <- calibrateDepth(tdrX,
+     zoc.method = "filter",
+     k = c(3, 5760), probs = c(0.5,
+         0.02), na.rm = TRUE)
```

Once the whole record has been zero-offset corrected, remaining depths below zero, are set to zero, as these are assumed to indicate values at the surface.

Finally, `calibrateDepth()` identifies all dives in the record, according to a minimum depth criterion given as its *dive.thr* argument. The value for this criterion is typically determined by the resolution of the instrument and the level of noise close to the surface. Thus, dives are defined as departures from the surface to maximal depths below *dive.thr* and the subsequent return to the surface. Each dive may subsequently be referred to by an integer number indicating its position in the time series.

Dive phases are also identified at this last stage, and is done using a smoothing spline model of the dive and its first derivative. Detection of dive phases is controlled by four arguments: a critical quantile for rates of vertical descent (*descent.crit.q*), a critical quantile for rates of ascent (*ascent.crit.q*), a smoothing parameter (*smooth.par*), and a factor (*knot.factor*) that multiplies the duration of the dive to obtain the number of knots at which to evaluate the derivative of the smoothing spline. The first two arguments are used to define the rate of descent below which the descent phase is deemed to have ended, and the rate of ascent above which the ascent phase is deemed to have started, respectively. The rates are obtained by evaluating the derivative of the smoothing at a number of knots placed regularly throughout the dive. Descent is deemed to have ended at the *first* minimum derivative, and the nearest input time observation is considered to indicate the end of descent. The sign of the comparisons is reversed for detecting the ascent.

A more refined call to `calibrateDepth()` for object `tdrX` may be:

```
> dcalib <- calibrateDepth(tdrX,
+     dive.thr = 3, zoc.method = "offset",
+     offset = 3, descent.crit.q = 0.01,
```

```
+     ascent.crit.q = 0, knot.factor = 20)
```

The result (value) of this function is an object of class *TDRcalibrate*, where all the information obtained during the tasks described above are stored.

# 6 How to Represent Calibrated TDR Data?

Objects of class TDRcalibrate contain the following slots, which store information during the major procedures performed by `calibrateDepth()`:

**call:** TDR. The call used to generate the object.

**tdr:** TDR. The object which was calibrated.

**gross.activity:** *list*. This list contains four components with details on wet/dry activities detected, such as start and end times, durations, and identifiers and labels for each activity period. Five activity categories are used for labelling each reading, indicating dry (L), wet (W), underwater (U), diving (D), and brief wet (Z) periods. However, underwater and diving periods are collapsed into wet activity at this stage (see below).

**dive.activity:** *data.frame*. This data frame contains three components with details on the diving activities detected, such as numeric vectors identifying to which dive and post-dive interval each reading belongs to, and a factor labelling the activity each reading represents. Compared to the `gross.activity` slot, the underwater and diving periods are discerned here.

**dive.phases:** *factor*. This identifies each reading with a particular dive phase. Thus, each reading belongs to one of descent, descent/bottom, bottom, bottom/ascent, and ascent phases. The descent/bottom and bottom/ascent levels are useful for readings which could not unambiguously be assigned to one of the other levels.

**dive.models:** *list*. This list contains all the details of the modelling process used to identifies dive phases. Each member of this list consists of objects of class *diveModel*, for which important methods are available.

**dry.thr:** *numeric*.

**wet.thr:** *numeric.*

**dive.thr:** *numeric.* These last three slots store information given as arguments to `calibrateDepth()`, documenting criteria used during calibration.

**speed.calib.coefs:** *numeric.* If the object calibrated was of class *TDRspeed*, then this is a vector of length 2, with the intercept and the slope of the speed calibration line (see below).

All the information contained in each of these slots is easily accessible through extractor methods for objects of this class (see class?TDRcalibrate). An appropriate *show()* method is available to display a short summary of such objects, including the number of dry and wet periods identified, and the number of dives detected.

The *TDRcalibrate plotTDR()* method for these objects allows visualizing the major wet/dry activities throughout the record (Figure 2):

```
> plotTDR(dcalib, concurVars = c("speed",
+     "light"), surface = TRUE)
```

The `dcalib` object contains a *TDRspeed* object in its `tdr` slot, and speed is plotted by default in this case. Additional measurements obtained concurrently can also be plotted using the *concurVars* argument. Titles for the depth axis and the concurrent parameters use separate arguments; the former uses *ylab.depth*, while the latter uses *concurVarTitles*. Convenient default values for these are provided. The *surface* argument controls whether post-dive readings should be plotted; it is `FALSE` by default, causing only dive readings to be plotted which saves time plotting and re-plotting the data. All plot methods use the underlying `plotTD()` function, which has other useful arguments that can be passed from these methods.

A more detailed view of the record can be obtained by using a combination of the *diveNo* and the *labels* arguments to this *plotTDR()* method. This is useful if, for instance, closer inspection of certain dives is needed. The following call displays a plot of dives 2 through 8 (Figure 3):

```
> plotTDR(dcalib, diveNo = 2:8,
+     labels = "dive.phase")
```

The *labels* argument allows the visualization of the identified dive phases for all dives selected. The
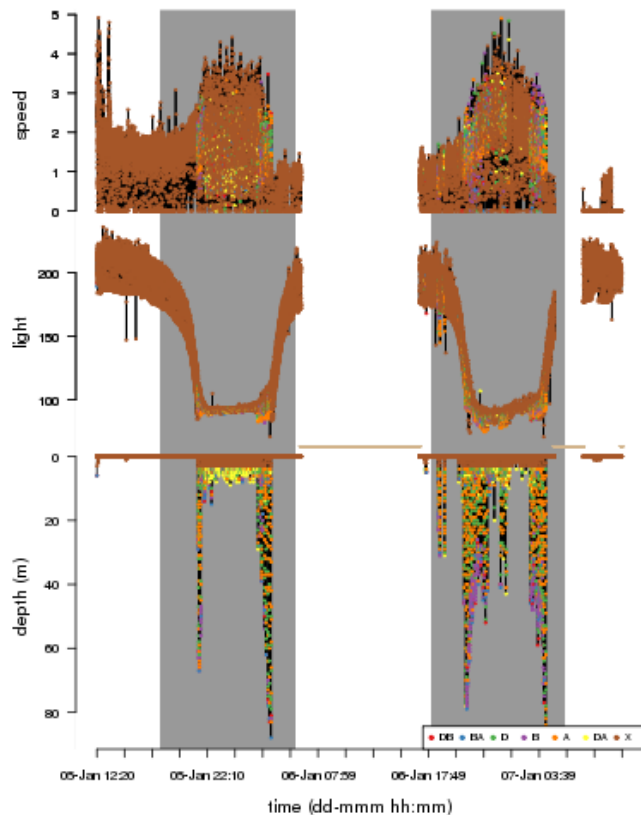


**Figure 2.** The `plotTDR()` method for *TDRcalibrate* objects displays information on the major activities identified throughout the record (wet/dry periods here).

same information can also be obtained with the *extractDive()* method for *TDRcalibrate* objects:

```
> extractDive(dcalib, diveNo = 2:8)
```

Other useful extractors include: *getGAct()* and *getDAct()*. These methods extract the whole `gross.activity` and `dive.activity`, respectively, if given only the `TDRcalibrate` object, or a particular component of these slots, if supplied a string with the name of the component. For example: `getGAct(dcalib, "trip.act")` would retrieve the factor identifying each reading with a wet/dry activity and `getDAct(dcalib, "dive.activity")` would retrieve a more detailed factor with information on whether the reading belongs to a dive or a brief aquatic period. Below is a demonstration of these methods.

*getTDR()*: This method simply takes the *TDRcalibrate* object as its single argument and extracts the *TDR* object[3]:

---

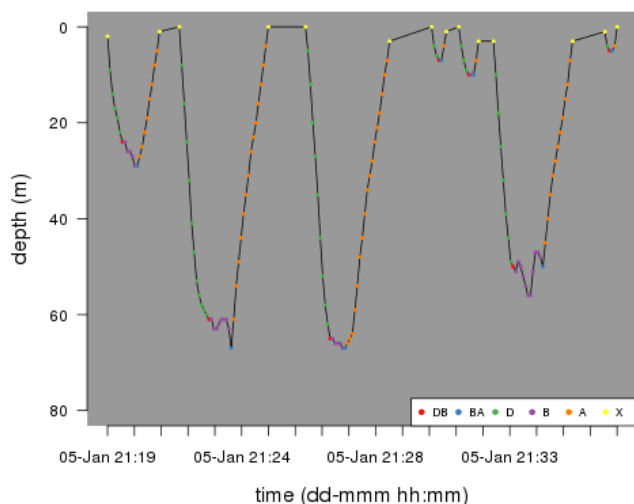[3]In fact, a *TDRspeed* object in this example

**Figure 3.** The *plotTDR()* method for *TDRcalibrate* objects can also display information on the different activities during each dive record (descent=D, descent/bottom=DB, bottom=B, bottom/ascent=BA, ascent=A, X=surface).

```
> getTDR(dcalib)

Time-Depth Recorder data -- Class TDRspeed object
  Source File         : dives.csv
  Sampling Interval (s): 5
  Number of Samples   : 34199
  Sampling Begins     : 2002-01-05 11:32:00
  Sampling Ends       : 2002-01-07 11:01:50
  Total Duration (d)  : 1.979
  Measured depth range : [0, 88]
  Other variables     :  light temperature speed
```

*getGAct()*: There are two methods for this generic, allowing access to a list with details about all wet/dry periods found. One of these extracts the entire *list* (output omitted for brevity):

```
> getGAct(dcalib)
```

The other provides access to particular elements of the *list*, by their name. For example, if we are interested in extracting only the vector that tells us to which period number every row in the record belongs to, we would issue the command:

```
> getGAct(dcalib, "phase.id")
```

Other elements that can be extracted are named "activity", "begin", and "end", and can be extracted in a similar fashion. These elements correspond to the activity performed for each reading (see ?detPhase for a description of the labels for each activity), the beginning and ending time for each period, respectively.

*getDAct()*: This generic also has two methods; one to extract an entire data frame with details about all dive and postdive periods found (output omitted):

```
> getDAct(dcalib)
```

The other method provides access to the columns of this data frame, which are named "dive.id", "dive.activity", and "postdive.id". Thus, providing any one of these strings to getDAct, as a second argument will extract the corresponding column.

*getDPhaseLab()*: This generic function extracts a factor identifying each row of the record to a particular dive phase (see ?detDive for a description of the labels of the factor identifying each dive phase). Two methods are available; one to extract the entire factor, and the other to select particular dive(s), by its (their) index number, respectively (output omitted):

```
> getDPhaseLab(dcalib)
> getDPhaseLab(dcalib, 20)

> dphases <- getDPhaseLab(dcalib,
+     c(100:300))
```

The latter method is useful for visually inspecting the assignment of points to particular dive phases. More information about the dive phase identification procedure can be gleaned by using the *plotDiveModel* (Figure 4):

```
> plotDiveModel(dcalib, diveNo = 260)
```

Another generic function that allows the subsetting of the original *TDR* object to a single a dive or group of dives' data:

```
> sealX <- extractDive(dcalib,
+     diveNo = c(100:300))
> sealX

Time-Depth Recorder data -- Class TDRspeed object
  Source File         : dives.csv
  Sampling Interval (s): 5
  Number of Samples   : 1758
  Sampling Begins     : 2002-01-05 23:40:20
  Sampling Ends       : 2002-01-06 23:04:45
  Total Duration (d)  : 0.9753
  Measured depth range : [0, 88]
  Other variables     :  light temperature speed
```

As can be seen, the function `extractDive` takes a *TDRcalibrate* object and a vector indicating the dive numbers to extract, and returns a *TDR* object containing the subsetted data. Once a subset of data has been selected, it is possible to plot them and
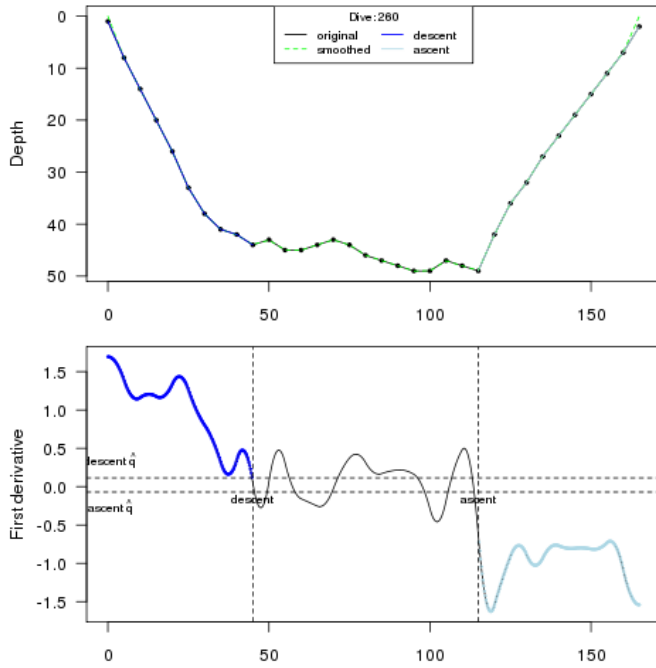
**Figure 4.** Details of the process of identification of dive phases shown by *plotDiveModel*, which has methods for objects of class *TDRcalibrate* and *diveModel*.

pass the factor labelling dive phases as the argument *phaseCol* to the *plot* method[4]:

```
> plotTDR(sealX, phaseCol = dphases)
```

With the information obtained during this calibration procedure, it is possible to calculate dive statistics for each dive in the record.

## 7 Dive Summaries

A table providing summary statistics for each dive can be obtained with the function `diveStats()` (Figure 5).

`diveStats()` returns a data frame with the final summaries for each dive (Figure 5), providing the following information:

- The time of start of the dive, the end of descent, and the time when ascent began.

- The total duration of the dive, and that of the descent, bottom, and ascent phases.

- The vertical distance covered during the descent, the bottom (a measure of the level of "wiggling", i.e. up and down movement performed during the bottom phase), and the vertical distance covered during the ascent.

- The maximum depth attained.

- The duration of the post-dive interval.

A summary of time budgets of wet vs. dry periods can be obtained with `timeBudget()`, which returns a data frame with the beginning and ending times for each consecutive period (Figure 5). It takes a *TDRcalibrate* object and another argument (*ignoreZ*) controlling whether aquatic periods that were briefer than the user-specified threshold[5] should be collapsed within the enclosing period of dry activity.

These summaries are the primary goal of `diveMove`, but they form the basis from which more elaborate and customized analyses are possible, depending on the particular research problem. These include investigation of descent/ascent rates based on the depth profiles, and bout structure analysis. Some of these will be implemented in the future.

In the particular case of *TDRspeed* objects, however, it may be necessary to calibrate the speed readings before calculating these statistics.

## 8 Calibrating Speed Sensor Readings

Calibration of speed sensor readings is performed using the procedure described by Blackwell et al. (1999). Briefly the method rests on the principle that for any given rate of depth change, the lowest measured speeds correspond to the steepest descent angles, i.e. vertical descent/ascent. In this case, measured speed and rate of depth change are expected to be equal. Therefore, a line drawn through the bottom edge of the distribution of observations in a plot of measured speed vs. rate of depth change would provide a calibration line. The calibrated speeds, therefore, can be calculated by reverse estimation of rate of depth change from the regression line.

---

[4]The function that the method uses is actually `plotTD`, so all the possible arguments can be studied by reading the help page for `plotTD`

[5]This corresponds to the value given as the *wet.thr* argument to `calibrateDepth()`.

```
> tdrXSumm1 <- diveStats(dcalib)
> names(tdrXSumm1)

 [1] "begdesc"           "enddesc"           "begasc"            "desctim"
 [5] "botttim"           "asctim"            "divetim"           "descdist"
 [9] "bottdist"          "ascdist"           "bottdep.mean"      "bottdep.median"
[13] "bottdep.sd"        "maxdep"            "desc.tdist"        "desc.mean.speed"
[17] "desc.angle"        "bott.tdist"        "bott.mean.speed"   "asc.tdist"
[21] "asc.mean.speed"    "asc.angle"         "postdive.dur"      "postdive.tdist"
[25] "postdive.mean.speed" "descD.min"       "descD.1stqu"       "descD.median"
[29] "descD.mean"        "descD.3rdqu"       "descD.max"         "descD.sd"
[33] "bottD.min"         "bottD.1stqu"       "bottD.median"      "bottD.mean"
[37] "bottD.3rdqu"       "bottD.max"         "bottD.sd"          "ascD.min"
[41] "ascD.1stqu"        "ascD.median"       "ascD.mean"         "ascD.3rdqu"
[45] "ascD.max"          "ascD.sd"

> tbudget <- timeBudget(dcalib, ignoreZ = TRUE)
> head(tbudget, 4)

  phaseno activity                 beg                 end
1       1        L 2002-01-05 11:32:00 2002-01-05 11:39:40
2       2        W 2002-01-05 11:39:45 2002-01-06 06:30:00
3       3        L 2002-01-06 06:30:05 2002-01-06 17:01:10
4       4        W 2002-01-06 17:01:15 2002-01-07 05:00:30

> trip.labs <- stampDive(dcalib, ignoreZ = TRUE)
> tdrXSumm2 <- data.frame(trip.labs, tdrXSumm1)
> names(tdrXSumm2)

 [1] "trip.no"           "trip.type"         "beg"               "end"
 [5] "begdesc"           "enddesc"           "begasc"            "desctim"
 [9] "botttim"           "asctim"            "divetim"           "descdist"
[13] "bottdist"          "ascdist"           "bottdep.mean"      "bottdep.median"
[17] "bottdep.sd"        "maxdep"            "desc.tdist"        "desc.mean.speed"
[21] "desc.angle"        "bott.tdist"        "bott.mean.speed"   "asc.tdist"
[25] "asc.mean.speed"    "asc.angle"         "postdive.dur"      "postdive.tdist"
[29] "postdive.mean.speed" "descD.min"       "descD.1stqu"       "descD.median"
[33] "descD.mean"        "descD.3rdqu"       "descD.max"         "descD.sd"
[37] "bottD.min"         "bottD.1stqu"       "bottD.median"      "bottD.mean"
[41] "bottD.3rdqu"       "bottD.max"         "bottD.sd"          "ascD.min"
[45] "ascD.1stqu"        "ascD.median"       "ascD.mean"         "ascD.3rdqu"
[49] "ascD.max"          "ascD.sd"
```

**Figure 5.** Per-dive summaries can be obtained with functions `diveStats()`, and a summary of time budgets with `timeBudget()`. `diveStats()` takes a *TDRcalibrate* object as a single argument (object `dcalib` above, see text for how it was created).

`diveMove` implements this procedure with function `calibrateSpeed()`. This function performs the following tasks:

1. Subset the necessary data from the record. By default only data corresponding to depth changes $> 0$ are included in the analysis, but higher constraints can be imposed using the $z$ argument. A further argument limiting the data to be used for calibration is *bad*, which is a vector with the minimum *rate* of depth change and minimum speed readings to include in the calibration. By default, values $> 0$ for both parameters are used.

2. Calculate the binned bivariate kernel density and extract the desired contour. Once the proper data were obtained, a bivariate normal kernel density grid is calculated from the relationship between measured speed and rate of depth change (using the `KernSmooth` package). The choice of bandwidths for the binned kernel density is made using *bw.nrd*. The *contour.level* argument to `calibrateSpeed()` controls which particular contour should be extracted from the density grid. Since the interest is in defining a regression line passing through the lower densities of the grid, this value should be relatively low (it is set to 0.1 by default).

3. Define the regression line passing through the lower edge of the chosen contour. A quantile regression through a chosen quantile is used for this purpose. The quantile can be specified using the *tau* argument, which is passed to the rq() function in package `quantreg`. *tau* is set to 0.1 by default.

4. Finally, the speed readings in the *TDR* object are calibrated.

As recognized by Blackwell et al. (1999), the advantage of this method is that it calibrates the instrument based on the particular deployment conditions (i.e. controls for effects of position of the instrument on the animal, and size and shape of the instrument, relative to the animal's morphometry, among others). However, it is possible to supply the coefficients of this regression if they were estimated separately; for instance, from an experiment. The argument *coefs* can be used for this purpose, which is then assumed to contain the intercept and the slope of the line. `calibrateSpeed()` returns a *TDRcalibrate* object, with calibrated speed readings
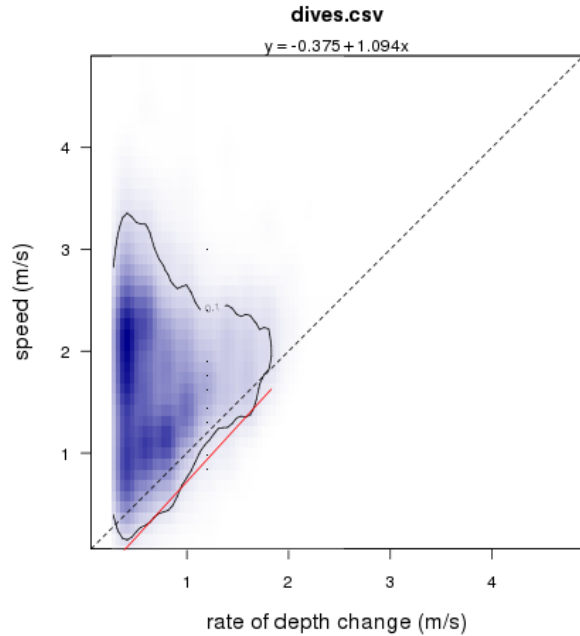


**Figure 6.** The relationship between measured speed and rate of depth change can be used to calibrate speed readings. The line defining the calibration for speed measurements passes through the bottom edge of a chosen contour, extracted from a bivariate kernel density grid.

included in its `tdr` slot, and the coefficients used for calibration.

For instance, to calibrate speed readings using the 0.1 quantile regression of measured speed vs. rate of depth change, based on the 0.1 contour of the bivariate kernel densities, and including only changes in depth $> 1$, measured speeds and rates of depth change $> 0$:

```
> vcalib <- calibrateSpeed(dcalib,
+     tau = 0.1, contour.level = 0.1,
+     z = 1, bad = c(0, 0),
+     cex.pts = 0.2)
```

This call produces the plot shown in Figure 6, which can be suppressed by the use of the logical argument *plot*. Calibrating speed readings allows for the meaningful interpretation of further parameters calculated by `diveStats()`, whenever a *TDRspeed* object was found in the *TDRcalibrate* object:

- The total distance travelled, mean speed, and diving angle during the descent and ascent phases of the dive.

- The total distance travelled and mean speed

during the bottom phase of the dive, and the post-dive interval.

# 9 Bout Detection

Diving behaviour often occurs in bouts for several species, so `diveMove` implements procedures for defining bout ending criteria (Langton et al. 1995; Luque and Guinet 2007). Please see `?bouts2.mle` and `?bouts2.nls` for examples.

# 10 Summary

The `diveMove` package provides tools for analyzing diving behaviour, including convenient methods for the visualization of the typically large amounts of data collected by `TDR`s. The package's main strengths are its ability to:

1. identify wet vs. dry periods,

2. calibrate depth readings,

3. identify individual dives and their phases,

4. summarize time budgets,

5. calibrate speed sensor readings,

6. provide basic summaries for each dive identified in `TDR` records, and

7. provide tools for identification of dive bout end criteria.

Formal `S4` classes are supplied to efficiently store `TDR` data and results from intermediate analysis, making the retrieval of intermediate results readily available for customized analysis. Development of the package is ongoing, and feedback, bug reports, or other comments from users are very welcome.

## Acknowledgements

# References

S. Blackwell, C. A. Haverl, B. J. Le Boeuf, and D. P. Costa. A method for calibrating swim-speed recorders. *Mar Mamm Sci*, 15(3):894–905, 1999.

S. D. Langton, D. Collett, and R. M. Sibly. Splitting behaviour into bouts; a maximum likelihood approach. *Behaviour*, 132:781–799, 1995.

S. P. Luque. Diving behaviour analysis in R. *R News*, 7:8–14, 2007.

S. P. Luque and C. Guinet. A maximum likelihood approach for identifying dive bouts improves accuracy, precision, and objectivity. *Behaviour*, 144: 1315–1332, 2007.

# diveMove

October 6, 2010

## R topics documented:

---

diveMove-package          *Dive Analysis and Calibration*

---

#### Description

This package is a collection of functions for visualizing, and analyzing depth and speed data from
time-depth recorders TDRs. These can be used to zero-offset correct depth, calibrate speed, and
divide the record into different phases, or time budget. Functions are provided for calculating
summary dive statistics for the whole record, or at smaller scales within dives.

#### Author(s)

Sebastian P. Luque <spluque@gmail.com>

#### See Also

A vignette with a guide to this package is available by doing vignette("diveMove"). TDR-
class, calibrateDepth, calibrateSpeed, timeBudget, stampDive.

#### Examples

```
## read in data and create a TDR object
(sealX <- readTDR(system.file(file.path("data", "dives.csv"),
                              package="diveMove"),
                  speed=TRUE, sep=";", na.strings="", as.is=TRUE))

if (dev.interactive(orNone=TRUE)) plotTDR(sealX)  # interactively pan and zoom

## detect periods of activity, and calibrate depth, creating
## a "TDRcalibrate" object
if (dev.interactive(orNone=TRUE)) dcalib <- calibrateDepth(sealX)
## Use the "offset" ZOC method to zero-offset correct depth at 3 m
(dcalib <- calibrateDepth(sealX, zoc.method="offset", offset=3))

if (dev.interactive(orNone=TRUE)) {
    ## plot all readings and label them with the phase of the record
    ## they belong to, excluding surface readings
    plotTDR(dcalib, surface=FALSE)
    ## plot the first 300 dives, showing dive phases and surface readings
    plotTDR(dcalib, diveNo=seq(300), surface=TRUE)
}

## calibrate speed (using changes in depth > 1 m and default remaining arguments)
(vcalib <- calibrateSpeed(dcalib, z=1))

## Obtain dive statistics for all dives detected
dives <- diveStats(vcalib)
head(dives)

## Attendance table
```

```
att <- timeBudget(vcalib, FALSE) # taking trivial aquatic activities into account
att <- timeBudget(vcalib, TRUE)  # ignoring them
## Add trip stamps to each dive
stamps <- stampDive(vcalib)
sumtab <- data.frame(stamps, dives)
head(sumtab)
```

---

austFilter                *Filter satellite locations*

---

### Description

Apply a three stage algorithm to eliminate erroneous locations, based on the procedure outlined in Austin et al. (2003).

### Usage

```
austFilter(time, lon, lat, id=gl(1, 1, length(time)),
           speed.thr, dist.thr, window=5)
grpSpeedFilter(x, speed.thr, window=5)
rmsDistFilter(x, speed.thr, window=5, dist.thr)
```

### Arguments

| | |
|---|---|
| time | POSIXct object with dates and times for each point. |
| lon | numeric vectors of longitudes, in decimal degrees. |
| lat | numeric vector of latitudes, in decimal degrees. |
| id | A factor grouping points in different categories (e.g. individuals). |
| speed.thr | numeric scalar: speed threshold (m/s) above which filter tests should fail any given point. |
| dist.thr | numeric scalar: distance threshold (km) above which the last filter test should fail any given point. |
| window | integer: the size of the moving window over which tests should be carried out. |
| x | 3-column matrix with column 1: POSIXct vector; column 2: numeric longitude vector; column 3: numeric latitude vector. |

### Details

These functions implement the location filtering procedure outlined in Austin et al. (2003). grpSpeedFilter and rmsDistFilter can be used to perform only the first stage or the second and third stages of the algorithm on their own, respectively. Alternatively, the three filters can be run in a single call using austFilter.

The first stage of the filter is an iterative process which tests every point, except the first and last ($w/2$) - 1 (where $w$ is the window size) points, for travel velocity relative to the preceeding/following ($w/2$) - 1 points. If all $w$ - 1 speeds are greater than the specified threshold, the point is marked as

failing the first stage. In this case, the next point is tested, removing the failing point from the set of test points.

The second stage runs McConnell et al. (1992) algorithm, which tests all the points that passed the first stage, in the same manner as above. The root mean square of all *w* - 1 speeds is calculated, and if it is greater than the specified threshold, the point is marked as failing the second stage (see Warning section below).

The third stage is run simultaneously with the second stage, but if the mean distance of all *w* - 1 pairs of points is greater than the specified threshold, then the point is marked as failing the third stage.

The speed and distance threshold should be obtained separately (see `distSpeed`).

## Value

`grpSpeedFilter` returns a logical vector indicating those lines that passed the test.

`rmsDistFilter` and `austFilter` return a matrix with 2 or 3 columns, respectively, of logical vectors with values TRUE for points that passed each stage. For the latter, positions that fail the first stage fail the other stages too. The second and third columns returned by `austFilter`, as well as those returned by `rmsDistFilter` are independent of one another; i.e. positions that fail stage 2 do not necessarily fail stage 3.

## Warning

This function applies McConnell et al.'s filter as described in Freitas et al. (2008). According to the original description of the algorithm in McConnell et al. (1992), the filter makes a single pass through all locations. Austin et al. (2003) and other authors may have used the filter this way. However, as Freitas et al. (2008) noted, this causes locations adjacent to those flagged as failing to fail also, thereby rejecting too many locations. In diveMove, the algorithm was modified to reject only the "peaks" in each series of consecutive locations having root mean square speed higher than threshold.

## Author(s)

Sebastian P. Luque <spluque@gmail.com> and Andy Liaw.

## References

McConnell BJ, Chambers C, Fedak MA. 1992. Foraging ecology of southern elephant seals in relation to bathymetry and productivity of the Southern Ocean. *Antarctic Science* 4:393-398.

Austin D, McMillan JI, Bowen D. 2003. A three-stage algorithm for filtering erroneous Argos satellite locations. *Marine Mammal Science* 19: 371-383.

Freitas C, Lydersen, C, Fedak MA, Kovacs KM. 2008. A simple new algorithm to filter marine mammal ARGOS locations. Marine Mammal Science DOI: 10.1111/j.1748-7692.2007.00180.x

## See Also

`distSpeed`

**Examples**

```
locs <- readLocs(system.file(file.path("data", "sealLocs.csv"),
                             package="diveMove"), idCol=1, dateCol=2,
                 dtformat="%Y-%m-%d %H:%M:%S", classCol=3, lonCol=4,
                 latCol=5, sep=";")
ringy <- subset(locs, id == "ringy" & !is.na(lon) & !is.na(lat))

## Austin et al.'s group filter alone
grp <- grpSpeedFilter(ringy[, 3:5], speed.thr=1.1)

## McConnell et al.'s filter (root mean square test), and distance test alone
rms <- rmsDistFilter(ringy[, 3:5], speed.thr=1.1, dist.thr=300)

## Show resulting tracks
n <- nrow(ringy)
plot.nofilter <- function(main) {
    plot(lat ~ lon, ringy, type="n", main=main)
    with(ringy, segments(lon[-n], lat[-n], lon[-1], lat[-1]))
}
layout(matrix(1:4, ncol=2, byrow=TRUE))
plot.nofilter(main="Unfiltered Track")
plot.nofilter(main="Group Filter")
n1 <- length(which(grp))
with(ringy[grp, ], segments(lon[-n1], lat[-n1], lon[-1], lat[-1],
                            col="blue"))
plot.nofilter(main="Root Mean Square Filter")
n2 <- length(which(rms[, 1]))
with(ringy[rms[, 1], ], segments(lon[-n2], lat[-n2], lon[-1], lat[-1],
                                 col="red"))
plot.nofilter(main="Distance Filter")
n3 <- length(which(rms[, 2]))
with(ringy[rms[, 2], ], segments(lon[-n3], lat[-n3], lon[-1], lat[-1],
                                 col="green"))


## All three tests (Austin et al. procedure)
austin <- with(ringy, austFilter(time, lon, lat, speed.thr=1.1,
                                 dist.thr=300))
layout(matrix(1:4, ncol=2, byrow=TRUE))
plot.nofilter(main="Unfiltered Track")
plot.nofilter(main="Stage 1")
n1 <- length(which(austin[, 1]))
with(ringy[austin[, 1], ], segments(lon[-n1], lat[-n1], lon[-1], lat[-1],
                                    col="blue"))
plot.nofilter(main="Stage 2")
n2 <- length(which(austin[, 2]))
with(ringy[austin[, 2], ], segments(lon[-n2], lat[-n2], lon[-1], lat[-1],
                                    col="red"))
plot.nofilter(main="Stage 3")
n3 <- length(which(austin[, 3]))
with(ringy[austin[, 3], ], segments(lon[-n3], lat[-n3], lon[-1], lat[-1],
                                    col="green"))
```

---

bout-methods                  *Methods for Plotting and Extracting the Bout Ending Criterion*

---

**Description**

Plot results from fitted mixture of 2-process Poisson models, and calculate the bout ending criterion.

**Usage**

```
## S4 method for signature 'nls':
plotBouts(fit, ...)
## S4 method for signature 'mle':
plotBouts(fit, x, ...)
## S4 method for signature 'nls':
bec2(fit)
## S4 method for signature 'mle':
bec2(fit)
```

**Arguments**

| | |
|---|---|
| fit | nls or mle object. |
| x | numeric object with variable modelled. |
| ... | Arguments passed to the underlying plotBouts2.nls and plotBouts2.mle. |

**General Methods**

**plotBouts** signature(fit="nls"): Plot fitted 2-process model of log frequency vs the interval mid points, including observed data.

**plotBouts** signature(x="mle"): As the nls method, but models fitted through maximum likelihood method. This plots the fitted model and a density plot of observed data.

**bec2** signature(fit="nls"): Extract the estimated bout ending criterion from a fitted 2-process model.

**bec2** signature(fit="mle"): As the nls method, but extracts the value from a maximum likelihood model.

**Author(s)**

Sebastian P. Luque <spluque@gmail.com>

**References**

Langton, S.; Collett, D. and Sibly, R. (1995) Splitting behaviour into bouts; a maximum likelihood approach. Behaviour **132**, 9-10.

Luque, S. P. and Guinet, C. (2007) A maximum likelihood approach for identifying dive bouts improves accuracy, precision, and objectivity. Behaviour **144**, 1315-1332.

Mori, Y.; Yoda, K. and Sato, K. (2001) Defining dive bouts using a sequential differences analysis. Behaviour **138**, 1451-1466.

Sibly, R.; Nott, H. and Fletcher, D. (1990) Splitting behaviour into bouts. Animal Behaviour **39**, 63-69.

**See Also**

`bouts.mle`, `bouts2.nls` for examples.

---

| bout-misc | *Fit a Broken Stick Model on Log Frequency Data for identification of bouts of behaviour* |
|---|---|

---

**Description**

Application of methods described by Sibly et al. (1990) and Mori et al. (2001) for the identification of bouts of behaviour, based on sequential differences of a variable.

**Usage**

```
boutfreqs(x, bw, method=c("standard", "seq.diff"), plot=TRUE)
boutinit(lnfreq, x.break, plot=TRUE)
labelBouts(x, bec, bec.method=c("standard", "seq.diff"))
logit(p)
unLogit(logit)
```

**Arguments**

| | |
|---|---|
| x | numeric vector on which bouts will be identified based on "method". For `labelBouts` it can also be a matrix with different variables for which bouts should be identified. |
| bw | numeric scalar: bin width for the histogram. |
| method, bec.method | |
| | character: method used for calculating the frequencies: "standard" simply uses x, while "seq.diff" uses the sequential differences method. |
| plot | logical, whether to plot results or not. |
| lnfreq | data.frame with components *lnfreq* (log frequencies) and corresponding x (mid points of histogram bins). |
| x.break | numeric scalar: x value defining the break point for broken stick model, such that x < xlim is 1st process, and x >= xlim is 2nd one. |

| bec | numeric vector or matrix with values for the bout ending criterion which should be compared against the values in x for identifying the bouts. |
|---|---|
| p | numeric vector of proportions (0-1) to transform to the logit scale. |
| logit | numeric scalar: logit value to transform back to original scale. |

### Details

This follows the procedure described in Mori et al. (2001), which is based on Sibly et al. 1990. Currently, only a two process model is supported.

boutfreqs creates a histogram with the log transformed frequencies of *x* with a chosen bin width and upper limit. Bins following empty ones have their frequencies averaged over the number of previous empty bins plus one.

boutinit fits a "broken stick" model to the log frequencies modelled as a function of *x* (well, the midpoints of the binned data), using a chosen value to separate the two processes.

labelBouts labels each element (or row, if a matrix) of *x* with a sequential number, identifying which bout the reading belongs to.

logit and unLogit are useful for reparameterizing the negative maximum likelihood function, if using Langton et al. (1995).

### Value

boutfreqs returns a data frame with components *lnfreq* containing the log frequencies and *x*, containing the corresponding mid points of the histogram. Empty bins are excluded. A plot is produced as a side effect if argument plot is TRUE. See the Details section.

boutinit returns a list with components a1, lambda1, a2, and lambda2, which are starting values derived from broken stick model. A plot is produced as a side effect if argument plot is TRUE.

labelBouts returns a numeric vector sequentially labelling each row or element of *x*, which associates it with a particular bout.

unLogit and logit return a numeric vector with the (un)transformed arguments.

### Author(s)

Sebastian P. Luque <spluque@gmail.com>

### References

Langton, S.; Collett, D. and Sibly, R. (1995) Splitting behaviour into bouts; a maximum likelihood approach. Behaviour **132**, 9-10.

Luque, S.P. and Guinet, C. (2007) A maximum likelihood approach for identifying dive bouts improves accuracy, precision, and objectivity. Behaviour, **144**, 1315-1332.

Mori, Y.; Yoda, K. and Sato, K. (2001) Defining dive bouts using a sequential differences analysis. Behaviour, 2001 **138**, 1451-1466.

Sibly, R.; Nott, H. and Fletcher, D. (1990) Splitting behaviour into bouts. Animal Behaviour **39**, 63-69.

## See Also

[bouts2.nls](), [bouts.mle]().

## Examples

```
## Using the Example from '?diveStats':
utils::example("diveStats", package="diveMove",
               ask=FALSE, echo=FALSE)
postdives <- tdrX.tab$postdive.dur[tdrX.tab$trip.no == 2]
## Remove isolated dives
postdives <- postdives[postdives < 2000]
lnfreq <- boutfreqs(postdives, bw=0.1, method="seq.diff", plot=FALSE)
boutinit(lnfreq, 50)
```

---

bouts2MLE                    *Maximum Likelihood Model of mixture of 2 Poisson Processes*

---

## Description

Functions to model a mixture of 2 random Poisson processes to identify bouts of behaviour. This
follows Langton et al. (1995).

## Usage

```
bouts2.mleFUN(x, p, lambda1, lambda2)
bouts2.ll(x)
bouts2.LL(x)
bouts.mle(ll.fun, start, x, ...)
bouts2.mleBEC(fit)
plotBouts2.mle(fit, x, xlab="x", ylab="Log Frequency", bec.lty=2, ...)
plotBouts2.cdf(fit, x, draw.bec=FALSE, bec.lty=2, ...)
```

## Arguments

| | |
|---|---|
| x | numeric vector with values to model. |
| p, lambda1, lambda2 | |
| | numeric: parameters of the mixture of Poisson processes. |
| ll.fun | function returning the negative of the maximum likelihood function that should be maximized. This should be a valid minuslogl argument to mle. |
| start, ... | Arguments passed to mle. For plotBouts2.cdf, arguments passed to plot.ecdf. For plotBouts2.mle, arguments passed to curve. |
| fit | mle object. |
| xlab, ylab | character: titles for the x and y axes. |
| bec.lty | Line type specification for drawing the BEC reference line. |
| draw.bec | logical; do we draw the BEC? |

**Details**

For now only a mixture of 2 Poisson processes is supported. Even in this relatively simple case, it is very important to provide good starting values for the parameters.

One useful strategy to get good starting parameter values is to proceed in 4 steps. First, fit a broken stick model to the log frequencies of binned data (see `boutinit`), to obtain estimates of 4 parameters corresponding to a 2-process model (Sibly et al. 1990). Second, calculate parameter *p* from the 2 alpha parameters obtained from the broken stick model, to get 3 tentative initial values for the 2-process model from Langton et al. (1995). Third, obtain MLE estimates for these 3 parameters, but using a reparameterized version of the -log L2 function. Lastly, obtain the final MLE estimates for the 3 parameters by using the estimates from step 3, un-transformed back to their original scales, maximizing the original parameterization of the -log L2 function.

`boutinit` can be used to perform step 1. Calculation of the mixing parameter *p* in step 2 is trivial from these estimates. Function `bouts2.LL` is a reparameterized version of the -log L2 function given by Langton et al. (1995), so can be used for step 3. This uses a logit (see `logit`) transformation of the mixing parameter *p*, and log transformations for both density parameters *lambda1* and *lambda2*. Function `bouts2.ll` is the -log L2 function corresponding to the un-transformed model, hence can be used for step 4.

`bouts.mle` is the function performing the main job of maximizing the -log L2 functions, and is essentially a wrapper around `mle`. It only takes the -log L2 function, a list of starting values, and the variable to be modelled, all of which are passed to `mle` for optimization. Additionally, any other arguments are also passed to `mle`, hence great control is provided for fitting any of the -log L2 functions.

In practice, step 3 does not pose major problems using the reparameterized -log L2 function, but it might be useful to use method "L-BFGS-B" with appropriate lower and upper bounds. Step 4 can be a bit more problematic, because the parameters are usually on very different scales. Therefore, it is almost always the rule to use method "L-BFGS-B", again bounding the parameter search, as well as passing a `control` list with proper `parscale` for controlling the optimization. See `Note` below for useful constraints which can be tried.

**Value**

`bouts.mle` returns an object of class `mle`.

`bouts2.mleBEC` and `bouts2.mleFUN` return a numeric vector.

`bouts2.LL` and `bouts2.ll` return a function.

`plotBouts2.mle` and `plotBouts2.cdf` return nothing, but produce a plot as side effect.

**Note**

In the case of a mixture of 2 Poisson processes, useful values for lower bounds for the `bouts.LL` reparameterization are `c(-2, -5, -10)`. For `bouts2.ll`, useful lower bounds are `rep(1e-08, 3)`. A useful parscale argument for the latter is `c(1, 0.1, 0.01)`. However, I have only tested this for cases of diving behaviour in pinnipeds, so these suggested values may not be useful in other cases.

**Author(s)**

Sebastian P. Luque <spluque@gmail.com>

**References**

Langton, S.; Collett, D. and Sibly, R. (1995) Splitting behaviour into bouts; a maximum likelihood approach. Behaviour **132**, 9-10.

Luque, S.P. and Guinet, C. (2007) A maximum likelihood approach for identifying dive bouts improves accuracy, precision, and objectivity. Behaviour, **144**, 1315-1332.

Sibly, R.; Nott, H. and Fletcher, D. (1990) Splitting behaviour into bouts. Animal Behaviour **39**, 63-69.

**See Also**

mle, optim, logit, unLogit for transforming and fitting a reparameterized model.

**Examples**

```
## Using the Example from '?diveStats':
utils::example("diveStats", package="diveMove",
               ask=FALSE, echo=FALSE)
postdives <- tdrX.tab$postdive.dur[tdrX.tab$trip.no == 2]
postdives.diff <- abs(diff(postdives))

## Remove isolated dives
postdives.diff <- postdives.diff[postdives.diff < 2000]
lnfreq <- boutfreqs(postdives.diff, bw=0.1, plot=FALSE)
startval <- boutinit(lnfreq, 50)
p <- startval$a1 / (startval$a1 + startval$a2)

## Fit the reparameterized (transformed parameters) model
init.parms <- list(p=logit(p), lambda1=log(startval$lambda1),
                   lambda2=log(startval$lambda2))
bout.fit1 <- bouts.mle(bouts2.LL, start=init.parms, x=postdives.diff,
                       method="L-BFGS-B", lower=c(-2, -5, -10))
coefs <- as.vector(coef(bout.fit1))

## Un-transform and fit the original parameterization
init.parms <- list(p=unLogit(coefs[1]), lambda1=exp(coefs[2]),
                   lambda2=exp(coefs[3]))
bout.fit2 <- bouts.mle(bouts2.ll, x=postdives.diff, start=init.parms,
                       method="L-BFGS-B", lower=rep(1e-08, 3),
                       control=list(parscale=c(1, 0.1, 0.01)))
plotBouts(bout.fit2, postdives.diff)

## Plot cumulative frequency distribution
plotBouts2.cdf(bout.fit2, postdives.diff)

## Estimated BEC
bec2(bout.fit2)
```

---

bouts2NLS                     *Fit mixture of 2 Poisson Processes to Log Frequency data*

---

### Description

Functions to model a mixture of 2 random Poisson processes to histogram-like data of log frequency vs interval mid points. This follows Sibly et al. (1990) method.

### Usage

```
bouts2.nlsFUN(x, a1, lambda1, a2, lambda2)
bouts2.nls(lnfreq, start, maxiter)
bouts2.nlsBEC(fit)
plotBouts2.nls(fit, lnfreq, bec.lty, ...)
```

### Arguments

| | |
|---|---|
| x | numeric vector with values to model. |
| a1, lambda1, a2, lambda2 | |
| | numeric: parameters from the mixture of Poisson processes. |
| lnfreq | data.frame with named components *lnfreq* (log frequencies) and corresponding *x* (mid points of histogram bins). |
| start, maxiter | |
| | Arguments passed to nls. |
| fit | nls object. |
| bec.lty | Line type specification for drawing the BEC reference line. |
| ... | Arguments passed to plot.default. |

### Details

bouts2.nlsFUN is the function object defining the nonlinear least-squares relationship in the model. It is not meant to be used directly, but is used internally by bouts2.nls.

bouts2.nls fits the nonlinear least-squares model itself.

bouts2.nlsBEC calculates the BEC from a list object, as the one that is returned by nls, representing a fit of the model. plotBouts2.nls plots such an object.

### Value

bouts2.nlsFUN returns a numeric vector evaluating the mixture of 2 Poisson process.

bouts2.nls returns an nls object resulting from fitting this model to data.

bouts2.nlsBEC returns a number corresponding to the bout ending criterion derived from the model.

plotBouts2.nls plots the fitted model with the corresponding data.

**Author(s)**

Sebastian P. Luque <spluque@gmail.com>

**References**

Sibly, R.; Nott, H. and Fletcher, D. (1990) Splitting behaviour into bouts Animal Behaviour **39**, 63-69.

**See Also**

bouts.mle for a better approach.

**Examples**

```
## Using the Example from '?diveStats':
utils::example("diveStats", package="diveMove",
               ask=FALSE, echo=FALSE)
## Postdive durations
postdives <- tdrX.tab$postdive.dur[tdrX.tab$trip.no == 2]
postdives.diff <- abs(diff(postdives))
## Remove isolated dives
postdives.diff <- postdives.diff[postdives.diff < 2000]

## Construct histogram
lnfreq <- boutfreqs(postdives.diff, bw=0.1, plot=FALSE)
startval <- boutinit(lnfreq, 50)

## Fit the 2 process model
bout.fit1 <- bouts2.nls(lnfreq, start=startval, maxiter=500)
summary(bout.fit1)
plotBouts(bout.fit1)

## Estimated BEC
bec2(bout.fit1)
```

---

calibrateDepth          *Calibrate Depth and Generate a "TDRcalibrate" object*

---

**Description**

Detect periods of major activities in a TDR record, calibrate depth readings, and generate a TDRcalibrate object essential for subsequent summaries of diving behaviour.

**Usage**

```
calibrateDepth(x, dry.thr=70, wet.thr=3610, dive.thr=4,
               zoc.method=c("visual", "offset", "filter"), ...,
               interp.wet=FALSE, smooth.par=0.1, knot.factor=3,
               descent.crit.q=0, ascent.crit.q=0)
```

**Arguments**

| | |
|---|---|
| x | An object of class `TDR` for `calibrateDepth` or an object of class `TDRcalibrate` for `calibrateSpeed`. |
| dry.thr | numeric: dry error threshold in seconds. Dry phases shorter than this threshold will be considered as wet. |
| wet.thr | numeric: wet threshold in seconds. At-sea phases shorter than this threshold will be considered as trivial wet. |
| dive.thr | numeric: threshold depth below which an underwater phase should be considered a dive. |
| zoc.method | character string to indicate the method to use for zero offset correction. One of "visual", "offset", or "filter" (see 'Details'). |
| ... | Arguments required for ZOC methods `filter` (`k`, `probs`, `depth.bounds` (defaults to range), `na.rm` (defaults to TRUE)) and `offset` (`offset`). |
| interp.wet | logical: if TRUE (default is FALSE), then an interpolating spline function is used to impute NA depths in wet periods (*after ZOC*). *Use with caution*: it may only be useful in cases where the missing data pattern in wet periods is restricted to shallow depths near the beginning and end of dives. This pattern is common in some satellite-linked TDRs. |
| smooth.par | numeric scalar representing amount of smoothing (argument `spar` in `smooth.spline`. |
| knot.factor | numeric scalar that multiplies the duration of the dive (used to construct the time predictor for the derivative). |
| descent.crit.q | |
| | numeric: critical quantile of rates of descent below which descent is deemed to have ended. |
| ascent.crit.q | |
| | numeric: critical quantile of rates of ascent above which ascent is deemed to have started. |

**Details**

This function is really a wrapper around `.detPhase`, `.detDive`, and `.zoc` which perform the work on simplified objects. It performs wet/dry phase detection, zero-offset correction of depth, and detection of dives, as well as proper labelling of the latter.

The procedure starts by first creating a factor with value "L" (dry) for rows with NAs for `depth` and value "W" (wet) otherwise. It subsequently calculates the duration of each of these phases of activity. If the duration of a dry phase ("L") is less than `dry.thr`, then the values in the factor for that phase are changed to "W" (wet). The duration of phases is then recalculated, and if the duration of a phase of wet activity is less than `wet.thr`, then the corresponding value for the

factor is changed to "Z" (trivial wet). The durations of all phases are recalculated a third time to provide final phase durations.

Zero-offset correction of depth is performed at this stage (see 'ZOC' below).

Some instruments produce a peculiar pattern of missing data near the surface, at the beginning and/or end of dives. The argument `interp.wet` may help to rectify this problem by using an interpolating spline function to impute the missing data, constraining the result to a minimum depth of zero. Please note that this optional step is performed after ZOC and before identifying dives, so that interpolation is performed through dry phases coded as wet because their duration was briefer than `dry.thr`. Therefore, `dry.thr` must be chosen carefully to avoid interpolation through legitimate dry periods.

The next step is to detect dives whenever the zero-offset corrected depth in an underwater phase is below the specified dive threshold. A new factor with finer levels of activity is thus generated, including "U" (underwater), and "D" (diving) in addition to the ones described above.

Once dives have been detected and assigned to a period of wet activity, phases within dives are identified using the descent, ascent and wiggle criteria (see 'Detection of dive phases' below). This procedure generates a factor with levels "D", "DB", "B", "BA", "A", "DA", and "X", breaking the input into descent, descent/bottom, bottom, bottom/ascent, ascent, and non-dive, respectively.

**Value**

An object of class `TDRcalibrate`.

**ZOC**

This procedure is required to correct drifts in the pressure transducer of TDR records and noise in depth measurements. Three methods are available to perform this correction.

Method "visual" calls `plotTDR`, which plots depth and, optionally, speed vs. time with the ability of zooming in and out on time, changing maximum depths displayed, and panning through time. The button to zero-offset correct sections of the record allows for the collection of 'x' and 'y' coordinates for two points, obtained by clicking on the plot region. The first point clicked represents the offset and beginning time of section to correct, and the second one represents the ending time of the section to correct. Multiple sections of the record can be corrected in this manner, by panning through the time and repeating the procedure. In case there's overlap between zero offset corrected windows, the last one prevails.

Method "offset" can be used when the offset is known in advance, and this value is used to correct the entire time series. Therefore, offset=0 specifies no correction.

Method "filter" implements a smoothing/filtering mechanism where running quantiles can be applied to depth measurements in a recursive manner, using `.depth.filter`. It relies on function `runquantile` from the caTools package. The method calculates the first running quantile defined by `probs[1]` on a moving window of size `k[1]`. The next running quantile, defined by `probs[2]` and `k[2]`, is applied to the smoothed/filtered depth measurements from the previous step, and so on. The corrected depth measurements (d) are calculated as:

$$d = d_0 - d_n$$

where $d_0$ is original depth and $d_n$ is the last smoothed/filtered depth. This method is under development, but reasonable results can be achieved by applying two filters (see 'Examples'). The default

`na.rm=TRUE` works well when there are no level shifts between non-NA phases in the data, but `na.rm=FALSE` is better in the presence of such shifts. In other words, there is no reason to pollute the moving window with NAs when non-NA phases can be regarded as a continuum, so splicing non-NA phases makes sense. Conversely, if there are level shifts between non-NA phases, then it is better to retain NA phases to help the algorithm recognize the shifts while sliding the window(s). The search for the surface can be limited to specified bounds during smoothing/filtering, so that observations outside these bounds are interpolated using the bounded smoothed/filtered series.

Once the whole record has been zero-offset corrected, remaining depths below zero, are set to zero, as these are assumed to indicate values at the surface.

**Detection of dive phases**

The process for each dive begins by taking all observations below the dive detection threshold, and setting the beginning and end depths to zero, at time steps prior to the first and after the last, respectively. The latter ensures that descent and ascent derivatives are non-negative and non-positive, respectively, so that the end and beginning of these phases are not truncated. A smoothing spline is used to model the dive and its derivative to investigate its changes in vertical rate. This method requires at least 4 observations (see `smooth.spline`), so the time series is linearly interpolated at equally spaced time steps if this limit is not achieved in the current dive. Wiggles at the beginning and end of the dive are assumed to be zero offset correction errors, so depth observations at these extremes are interpolated between zero and the next observations when this occurs.

A set of knots is established to fit the smoothing spline by using an regular time sequence with beginning and end equal to the extremes of the input sequence, and with length equal to $N \times knot.factor$. The first derivate of the spline is evaluated at the same set of knots to calculate the vertical rate throughout the dive and determine the end of descent and beginning of ascent. Equivalent procedures are used for detecting descent and ascent phases.

The quantile corresponding to (`descent.crit.q`) of all the positive derivatives (rate of descent) at the beginning of the dive is used as threshold for determining the end of descent. Descent is deemed to have ended at the *first* minimum derivative, and the nearest input time observation is considered to indicate the end of descent. The sign of the comparisons is reversed for detecting the ascent. If observed depth to the left and right of the derivative defining the ascent are the same, the right takes precedence.

The particular dive phase categories are subsequently defined using simple set operations.

**Author(s)**

Sebastian P. Luque <spluque@gmail.com>

**References**

Luque, S.P. and Fried, R. (Submitted) Recursive filtering for zero offset correction of diving depth time series.

**See Also**

TDRcalibrate, .zoc, .depthFilter, .detPhase, .detDive, plotTDR.

**Examples**

```
data(divesTDR)
divesTDR

## Consider a 3 m offset, a dive threshold of 3 m, the 1% quantile for
## critical vertical rates, and a set of knots 20 times as long as the
## observed time steps.
(dcalib <- calibrateDepth(divesTDR, dive.thr=3, zoc.method="offset",
                          offset=3, descent.crit.q=0.01, ascent.crit.q=0,
                          knot.factor=20))
## Or ZOC algorithmically with method="filter":
## Not run: ## This can take a while due to large window needed for 2nd
## filter in this dataset
(dcalib <- calibrateDepth(divesTDR, dive.thr=3, zoc.method="filter",
                          k=c(3, 5760), probs=c(0.5, 0.02), na.rm=TRUE,
                          descent.crit.q=0.01, ascent.crit.q=0,
                          knot.factor=20))

## End(Not run)
```

---

calibrateSpeed        *Calibrate and build a "TDRcalibrate" object*

---

**Description**

These functions create a TDRcalibrate object which is necessary to obtain dive summary statistics.

**Usage**

```
calibrateSpeed(x, tau=0.1, contour.level=0.1, z=0, bad=c(0, 0),
               main=slot(getTDR(x), "file"), coefs, plot=TRUE,
               postscript=FALSE, ...)
```

**Arguments**

| | |
|---|---|
| x | An object of class TDR for calibrateDepth or an object of class TDRcalibrate for calibrateSpeed. |
| tau | numeric scalar: quantile on which to regress speed on rate of depth change; passed to rq. |
| contour.level | |
| | numeric scalar: the mesh obtained from the bivariate kernel density estimation corresponding to this contour will be used for the quantile regression to define the calibration line. |
| z | numeric scalar: only changes in depth larger than this value will be used for calibration. |

| bad | numeric vector of length 2 indicating that only rates of depth change and speed greater than the given value should be used for calibration, respectively. |
|---|---|
| coefs | numeric: known speed calibration coefficients from quantile regression as a vector of length 2 (intercept, slope). If provided, these coefficients are used for calibrating speed, ignoring all other arguments, except x. |
| main, ... | Arguments passed to rqPlot. |
| plot | logical: whether to plot the results. |
| postscript | logical: whether to produce postscript file output. |

### Details

This calibrates speed readings following the procedure outlined in Blackwell et al. (1999).

### Value

An object of class TDRcalibrate.

### Author(s)

Sebastian P. Luque <spluque@gmail.com>

### References

Blackwell S, Haverl C, Le Boeuf B, Costa D (1999). A method for calibrating swim-speed recorders. Marine Mammal Science 15(3):894-905.

### See Also

TDRcalibrate

### Examples

```
## Continuing the Example from '?calibrateDepth':
utils::example("calibrateDepth", package="diveMove",
               ask=FALSE, echo=FALSE)
dcalib # the 'TDRcalibrate' that was created

## Calibrate speed using only changes in depth > 2 m
vcalib <- calibrateSpeed(dcalib, z=2)
vcalib
```

---

distSpeed *Calculate distance and speed between locations*

---

**Description**

Calculate distance, time difference, and speed between pairs of points defined by latitude and longitude, given the time at which all points were measured.

**Usage**

```
distSpeed(pt1, pt2)
```

**Arguments**

| | |
|---|---|
| pt1 | A matrix or data frame with three columns; the first a POSIXct object with dates and times for all points, the second and third numeric vectors of longitude and latitude for all points, respectively, in decimal degrees. |
| pt2 | A matrix with the same size and structure as pt1. |

**Value**

A matrix with three columns: distance (km), time difference (s), and speed (m/s).

**Author(s)**

Sebastian P. Luque <spluque@gmail.com>

**Examples**

```
locs <- readLocs(system.file(file.path("data", "sealLocs.csv"),
                             package="diveMove"), idCol=1, dateCol=2,
                 dtformat="%Y-%m-%d %H:%M:%S", classCol=3, lonCol=4,
                 latCol=5, sep=";")

## Travel summary between successive standard locations
locs.std <- subset(locs, subset=class == "0" | class == "1" |
                   class == "2" | class == "3" &
                   !is.na(lon) & !is.na(lat))
locs.std.tr <- by(locs.std, locs.std$id, function(x) {
    distSpeed(x[-nrow(x), 3:5], x[-1, 3:5])
})
lapply(locs.std.tr, head)

## Particular quantiles from travel summaries
lapply(locs.std.tr, function(x) {
    quantile(x[, 3], seq(0.90, 0.99, 0.01), na.rm=TRUE) # speed
})
lapply(locs.std.tr, function(x) {
```

```
    quantile(x[, 1], seq(0.90, 0.99, 0.01), na.rm=TRUE) # distance
})

## Travel summary between two arbitrary sets of points
distSpeed(locs[c(1, 5, 10), 3:5], locs[c(25, 30, 35), 3:5])
```

diveModel-class    *Class "diveModel" for representing a model for identifying dive phases*

### Description

Details of model used to identify the different phases of a dive.

### Objects from the Class

Objects can be created by calls of the form `new("diveModel", ...)`.

'diveModel' objects contain all relevant details of the process to identify phases of a dive. Objects of this class are typically generated during depth calibration, using `calibrateDepth`, more specifically `.cutDive`.

### Slots

label.matrix: Object of class `"matrix"`. A 2-column character matrix with row numbers matching each observation to the full `TDR` object, and a vector labelling the phases of each dive.

dive.spline: Object of class `"smooth.spline"`. Details of cubic smoothing spline fit (see `smooth.spline`).

spline.deriv: Object of class `"list"`. A list with the first derivative of the smoothing spline (see `predict.smooth.spline`).

descent.crit: Object of class `"numeric"`. The index of the observation at which the descent was deemed to have ended (from initial surface observation).

ascent.crit: Object of class `"numeric"`. the index of the observation at which the ascent was deemed to have ended (from initial surface observation).

descent.crit.rate: Object of class `"numeric"`. The rate of descent corresponding to the critical quantile used.

ascent.crit.rate: Object of class `"numeric"`. The rate of ascent corresponding to the critical quantile used.

### Author(s)

Sebastian P. Luque <spluque@gmail.com>

### See Also

`getDiveDeriv`, `plotDiveModel`

## Examples

```
showClass("diveModel")
```

---

dives                          *Sample of TDR data from a fur seal*

---

### Description

This data set is meant to show a typical organization of a TDR *.csv file, suitable as input for readTDR, or to construct a TDR object. divesTDR is an example TDR object.

### Format

A comma separated value (csv) file with 69560 TDR readings with the following columns:

**date** Date
**time** Time
**depth** Depth in m
**light** Light level
**temperature** Temperature in degrees Celsius
**speed** Speed in m/s

The data are also provided as a TDR object (*.RData format) for convenience.

### Details

The data are a subset of an entire TDR record, so they are not meant to make valid inferences from this particular individual/deployment.

divesTDR is a TDR object representation of the data in dives.

### Source

Sebastian P. Luque, Christophe Guinet, John P.Y. Arnould

### See Also

readTDR, diveStats.

### Examples

```
dives <- read.csv(system.file(file.path("data", "dives.csv"),
                              package="diveMove"),
                  sep=";", na.strings="")
str(dives)
```

---

diveStats                          *Per-dive statistics*

---

**Description**

Calculate dive statistics in TDR records.

**Usage**

```
diveStats(x, depth.deriv=TRUE)
oneDiveStats(x, interval, speed=FALSE)
stampDive(x, ignoreZ=TRUE)
```

**Arguments**

| | |
|---|---|
| x | A TDRcalibrate-class object for diveStats and stampDive, and a data frame containing a single dive's data (a factor identifying the dive phases, a POSIXct object with the time for each reading, a numeric depth vector, and a numeric speed vector) for oneDiveStats. |
| depth.deriv | logical: should depth derivative statistics be calculated? |
| interval | numeric scalar: sampling interval for interpreting x. |
| speed | logical: should speed statistics be calculated? |
| ignoreZ | logical: whether trips should be numbered considering all aquatic activities ("W" and "Z") or ignoring "Z" activities. |

**Details**

diveStats calculates various dive statistics based on time and depth for an entire TDR record. oneDiveStats obtains these statistics from a single dive, and stampDive stamps each dive with associated trip information.

**Value**

A data.frame with one row per dive detected (durations are in s, and linear variables in m):

| | |
|---|---|
| begdesc | A POSIXct object, specifying the start time of each dive. |
| enddesc | A POSIXct object, as begdesc indicating descent's end time. |
| begasc | A POSIXct object, as begdesc indicating the time ascent began. |
| desctim | Descent duration of each dive. |
| botttim | Bottom duration of each dive. |
| asctim | Ascent duration of each dive. |
| divetim | Dive duration. |
| descdist | Numeric vector with last descent depth. |

| | |
|---|---|
| bottdist | Numeric vector with the sum of absolute depth differences while at the bottom of each dive; measure of amount of "wiggling" while at bottom. |
| ascdist | Numeric vector with first ascent depth. |
| bottdep.mean | Mean bottom depth. |
| bottdep.median | |
| | Median bottom depth. |
| bottdep.sd | Standard deviation of bottom depths. |
| maxdep | Numeric vector with maximum depth. |
| desc.tdist | Numeric vector with descent total distance, estimated from speed. |
| desc.mean.speed | |
| | Numeric vector with descent mean speed. |
| desc.angle | Numeric vector with descent angle, from the surface plane. |
| bott.tdist | Numeric vector with bottom total distance, estimated from speed. |
| bott.mean.speed | |
| | Numeric vector with bottom mean speed. |
| asc.tdist | Numeric vector with ascent total distance, estimated from speed. |
| asc.mean.speed | |
| | Numeric vector with ascent mean speed. |
| asc.angle | Numeric vector with ascent angle, from the bottom plane. |
| postdive.dur | Postdive duration. |
| postdive.tdist | |
| | Numeric vector with postdive total distance, estimated from speed. |
| postdive.mean.speed | |
| | Numeric vector with postdive mean speed. |

If `depth.deriv=TRUE`, 21 additional columns with the minimum, first quartile, median, mean, third quartile, maximum, and standard deviation of the depth derivative for each phase of the dive. The number of columns also depends on the value of speed.

`stampDive` returns a data.frame with trip number, trip type, and start and end times for each dive.

**Author(s)**

Sebastian P. Luque <spluque@gmail.com>

**See Also**

calibrateDepth, .detPhase, TDRcalibrate-class

**Examples**

```
## Continuing the Example from '?calibrateDepth':
utils::example("calibrateDepth", package="diveMove",
               ask=FALSE, echo=FALSE)
dcalib # the 'TDRcalibrate' that was created
```

```
tdrX <- diveStats(dcalib)
stamps <- stampDive(dcalib, ignoreZ=TRUE)
tdrX.tab <- data.frame(stamps, tdrX)
summary(tdrX.tab)
```

---

extractDive-methods

*Extract Dives from "TDR" or "TDRcalibrate" Objects*

---

#### Description

Extract data corresponding to a particular dive(s), referred to by number.

#### Usage

```
## S4 method for signature 'TDR,numeric,numeric':
extractDive(obj, diveNo, id)
## S4 method for signature 'TDRcalibrate,numeric,missing':
extractDive(obj, diveNo)
```

#### Arguments

| | |
|---|---|
| obj | TDR object. |
| diveNo | numeric vector or scalar with dive numbers to extract. |
| id | numeric vector or scalar of dive numbers from where diveNo should be chosen. |

#### Value

An object of class TDR or TDRspeed.

#### Author(s)

Sebastian P. Luque <spluque@gmail.com>

#### Examples

```
## Continuing the Example from '?calibrateDepth':
utils::example("calibrateDepth", package="diveMove",
               ask=FALSE, echo=FALSE)
dcalib # the 'TDRcalibrate' that was created

diveX <- extractDive(divesTDR, 9, getDAct(dcalib, "dive.id"))
plotTDR(diveX, interact=FALSE)

diveX <- extractDive(dcalib, 5:10)
plotTDR(diveX, interact=FALSE)
```

```
plotDiveModel-methods
```
*Methods for plotting models of dive phases*

### Description

Methods for function `plotDiveModel`.

### Usage

```
## S4 method for signature 'diveModel,missing':
plotDiveModel(x, diveNo)

## S4 method for signature 'numeric,numeric':
plotDiveModel(x, y, times.s, depths.s, d.crit, a.crit,
              diveNo=1, times.deriv, depths.deriv,
              d.crit.rate, a.crit.rate)

## S4 method for signature 'TDRcalibrate,missing':
plotDiveModel(x, diveNo)
```

### Arguments

| | |
|---|---|
| x | A diveModel (diveMode,missing method), numeric vector of time step observations (numeric,numeric method), or TDRcalibrate object (TDRcalibrate,numeric method). |
| diveNo | integer representing the dive number selected for plotting. |
| y | numeric vector with depth observations at each time step. |
| times.s | numeric vector with time steps used to generate the smoothing spline (i.e. the knots, see diveModel. |
| depths.s | numeric vector with smoothed depth (see diveModel). |
| d.crit | integer denoting the index where descent ends in the observed time series (see diveModel). |
| a.crit | integer denoting the index where ascent begins in the observed time series (see diveModel). |
| times.deriv | numeric vector representing the time steps where the derivative of the smoothing spline was evaluated diveModel. |
| depths.deriv | numeric vector representing the derivative of the smoothing spline evaluated at times.deriv diveModel. |
| d.crit.rate | numeric scalar: vertical rate of descent corresponding to the quantile used (see diveModel. |
| a.crit.rate | numeric scalar: vertical rate of ascent corresponding to the quantile used (see diveModel. |

**Methods**

All methods produce a double panel plot. The top panel shows the depth against time, the cubic spline smoother, the identified descent and ascent phases (which form the basis for identifying the rest of the dive phases), while the bottom panel shows the first derivative of the smooth trace.

signature(x = "diveModel", y = "missing") Given a diveModel object and (possibly) the dive number that it corresponds to, the plot shows the model data.

signature(x = "numeric", y = "numeric") This is the main method, which requires all aspects of the model to be provided.

signature(x = "TDRcalibrate", y = "missing") Given a TDRcalibrate object and a dive number to extract from it, this method plots the observed data and the model. The intended use of this method is through plotTDR when what="dive.model".

**Author(s)**

Sebastian P. Luque <spluque@gmail.com>

**See Also**

diveModel

**Examples**

```
## Continuing the Example from '?calibrateDepth':
utils::example("calibrateDepth", package="diveMove",
               ask=FALSE, echo=FALSE)

## 'diveModel' method
dm <- getDiveModel(dcalib, 100)
plotDiveModel(dm, diveNo=100)

## 'TDRcalibrate' method
plotDiveModel(dcalib, diveNo=100)
```

---

plotTDR-methods      *Methods for plotting objects of class "TDR" and "TDRcalibrate"*

---

**Description**

Main plotting method for objects of these classes. Plot and optionally set zero-offset correction windows in TDR records, with the aid of a graphical user interface (GUI), allowing for dynamic selection of offset and multiple time windows to perform the adjustment.

**Usage**

```
## S4 method for signature 'POSIXt,numeric':
plotTDR(x, y, concurVars=NULL, xlim=NULL, depth.lim=NULL,
        xlab="time (dd-mmm hh:mm)", ylab.depth="depth (m)",
        concurVarTitles=deparse(substitute(concurVars)),
        xlab.format="%d-%b %H:%M", sunrise.time="06:00:00",
        sunset.time="18:00:00", night.col="gray60",
        dry.time=NULL, phase.factor=NULL, plot.points=FALSE,
        interact=TRUE, key=TRUE, cex.pts=0.4, ...)
## S4 method for signature 'TDR,missing':
plotTDR(x, y, concurVars, concurVarTitles, ...)
## S4 method for signature 'TDRcalibrate,missing':
plotTDR(x, y, what=c("phases", "dive.model"),
        diveNo=seq(max(getDAct(x, "dive.id"))), ...)
```

**Arguments**

| | |
|---|---|
| x | POSIXct object with date and time, [TDR], or [TDRcalibrate] object. |
| y | numeric vector with depth in m. |
| concurVars | matrix with additional variables in each column to plot concurrently with depth. For the (TDR,missing) and (TDRcalibrate,missing) methods, a [character] vector naming additional variables from the concurrentData slot to plot, if any. |
| xlim | POSIXct or numeric vector of length 2, with lower and upper limits of time to be plotted. |
| depth.lim | numeric vector of length 2, with the lower and upper limits of depth to be plotted. |
| xlab, ylab.depth | |
| | character strings to label the corresponding y-axes. |
| concurVarTitles | |
| | character vector of titles to label each new variable given in *concurVars*. |
| xlab.format | character: format string for formatting the x axis; see [strptime]. |
| sunrise.time, sunset.time | |
| | character string with time of sunrise and sunset, respectively, in 24 hr format. This is used for shading night time. |
| night.col | color for shading night time. |
| dry.time | subset of time corresponding to observations considered to be dry. |
| phase.factor | factor dividing rows into sections. |
| plot.points | logical: whether to plot points. |
| interact | logical: whether to provide interactive tcltk controls and access to the associated ZOC functionality. |
| key | logical: whether to draw a key. |
| cex.pts | Passed to [points] to set the relative size of points to plot (if any). |

| ... | For the (POSIXt, numeric) method, arguments passed to [par](#) for all methods; useful defaults las=1, bty="n", and mar (the latter depending on whether additional concurrent data will be plotted) are provided, but they can be overridden. For other methods, except (TDRcalibrate, missing), these can be any of the arguments for the (POSIXt, numeric) method. For (TDRcalibrate, missing), these are arguments for the appropriate methods. |
|---|---|
| diveNo | numeric vector or scalar with dive numbers to plot. |
| what | character: what aspect of the [TDRcalibrate](#) to plot, which selects the method to use for plotting. |

### Details

This function is used primarily to correct drifts in the pressure transducer of TDR records and noise in depth measurements via method="visual" in [calibrateDepth](#).

### Value

If called with the interact argument set to TRUE, returns a list (invisibly) with as many components as sections of the record that were zero-offset corrected, each consisting of two further lists with the same components as those returned by [locator](#).

### Methods

**plotTDR** signature(x="TDR", y="numeric"): interactive graphical display of time-depth data, with zooming and panning capabilities.

**plotTDR** signature(x="TDR", y="missing"): As method above.

**plotTDR** signature(x="TDRcalibrate", y="missing"): plot selected aspects of [TDRcalibrate](#) object. Currently, two aspects have plotting methods:

- phases (Optional arguments: concurVars, surface) Plots all dives, labelled by the activity phase they belong to. It produces a plot consisting of one or more panels; the first panel shows depth against time, and additional panels show other concurrent data in the object. Optional argument concurVars is a character vector indicating which additional components from the concurrentData slot to plot, if any. Optional argument surface is a logical: whether to plot surface readings.
- dive.model Plots the dive model for the selected dive number (diveNo argument).

### Author(s)

Sebastian P. Luque <spluque@gmail.com>, with many ideas from CRAN package sfsmisc.

### See Also

[calibrateDepth](#), [.zoc](#)

## Examples

```
## Continuing the Example from '?calibrateDepth':
utils::example("calibrateDepth", package="diveMove",
               ask=FALSE, echo=FALSE)
## Use interact=TRUE (default) to set an offset interactively
## Plot the 'TDR' object
plotTDR(getTime(divesTDR), getDepth(divesTDR), interact=FALSE)
plotTDR(divesTDR, interact=FALSE)

## Plot different aspects of the 'TDRcalibrate' object
plotTDR(dcalib, interact=FALSE)
plotTDR(dcalib, diveNo=19:25, interact=FALSE)
plotTDR(dcalib, what="dive.model", diveNo=25)
if (dev.interactive(orNone=TRUE)) {
    ## Add surface observations and interact
    plotTDR(dcalib, surface=TRUE)
    ## Plot one dive
    plotTDR(dcalib, diveNo=200)
}
```

---

readLocs                *Read comma-delimited file with location data*

---

## Description

Read a delimited (*.csv) file with (at least) time, latitude, longitude readings.

## Usage

```
readLocs(locations, loc.idCol, idCol, dateCol, timeCol=NULL,
         dtformat="%m/%d/%Y %H:%M:%S", tz="GMT",
         classCol, lonCol, latCol, alt.lonCol=NULL, alt.latCol=NULL, ...)
```

## Arguments

| | |
|---|---|
| locations | character: a string indicating the path to the file to read, or a `data.frame` available in the search list. Provide the entire path if the file is not on the current directory. |
| loc.idCol | integer: column number containing location ID. If missing, a `loc.id` column is generated with sequential integers as long as the input. |
| idCol | integer: column number containing an identifier for locations belonging to different groups. If missing, an id column is generated with number one repeated as many times as the input. |
| dateCol | integer: column number containing dates, and, optionally, times. |
| timeCol | integer: column number containing times. |

| dtformat | character: a string specifying the format in which the date and time columns, when pasted together, should be interpreted (see [strptime](#)) in file. |
| tz | character: a string indicating the time zone for the date and time readings. |
| lonCol | integer: column number containing longitude readings. |
| latCol | integer: column number containing latitude readings. |
| classCol | integer: column number containing the ARGOS rating for each location. |
| alt.lonCol | integer: column number containing alternative longitude readings. |
| alt.latCol | integer: Column number containing alternative latitude readings. |
| ... | Passed to [read.csv](#) |

### Details

The file must have a header row identifying each field, and all rows must be complete (i.e. have the same number of fields). Field names need not follow any convention.

### Value

A data frame.

### Author(s)

Sebastian P. Luque <spluque@gmail.com>

### Examples

```
locs <- readLocs(system.file(file.path("data", "sealLocs.csv"),
                             package="diveMove"), idCol=1, dateCol=2,
                 dtformat="%Y-%m-%d %H:%M:%S", classCol=3,
                 lonCol=4, latCol=5, sep=";")

summary(locs)
```

---

readTDR                          *Read comma-delimited file with "TDR" data*

---

### Description

Read a delimited (*.csv) file containing time-depth recorder (*TDR*) data from various TDR models. Return a TDR or TDRspeed object. createTDR creates an object of one of these classes from other objects.

## Usage

```
readTDR(file, dateCol=1, timeCol=2, depthCol=3, speed=FALSE,
        subsamp=5, concurrentCols=4:6,
        dtformat="%d/%m/%Y %H:%M:%S", tz="GMT", ...)
createTDR(time, depth, concurrentData=data.frame(), speed=FALSE, dtime, file)
```

## Arguments

| | |
|---|---|
| `file` | character: a string indicating the path to the file to read. |
| `dateCol` | integer: column number containing dates, and optionally, times. |
| `timeCol` | integer: column number with times. |
| `depthCol` | integer: column number containing depth readings. |
| `speed` | logical: whether speed is included in one of the columns of concurrentCols. |
| `subsamp` | numeric scalar: subsample rows in `file` with subsamp interval, in s. |
| `concurrentCols` | |
| | integer vector of column numbers to include as concurrent data collected. |
| `dtformat` | character: a string specifying the format in which the date and time columns, when pasted together, should be interpreted (see [strptime](#)). |
| `tz` | character: a string indicating the time zone assumed for the date and time readings. |
| `...` | Passed to [read.csv](#) |
| `time` | A POSIXct object with date and time readings for each reading. |
| `depth` | numeric vector with depth readings. |
| `concurrentData` | |
| | data.frame with additional, concurrent data collected. |
| `dtime` | numeric scalar: sampling interval used in seconds. If missing, it is calculated from the `time` argument. |

## Details

The input file is assumed to have a header row identifying each field, and all rows must be complete (i.e. have the same number of fields). Field names need not follow any convention. However, depth and speed are assumed to be in m, and $m \cdot s^{-1}$, respectively, for further analyses.

If *speed* is TRUE and concurrentCols contains a column named speed or velocity, then an object of class [TDRspeed](#) is created, where speed is considered to be the column matching this name.

## Value

An object of class [TDR](#) or [TDRspeed](#).

## Note

Although [TDR](#) and [TDRspeed](#) classes check that time stamps are in increasing order, the integrity of the input must be thoroughly verified for common errors present in text output from TDR devices such as duplicate records, missing time stamps and non-numeric characters in numeric fields. These errors are much more efficiently dealt with outside of GNU R using tools like GNU awk or GNU sed, so [diveMove](#) does not currently attempt to fix these errors.

**Author(s)**

Sebastian P. Luque <spluque@gmail.com>

**Examples**

```
readTDR(system.file(file.path("data", "dives.csv"),
                    package="diveMove"),
        speed=TRUE, sep=";", na.strings="", as.is=TRUE)

## Or more pedestrian
tdrX <- read.csv(system.file(file.path("data", "dives.csv"),
                             package="diveMove"),
                 sep=";", na.strings="", as.is=TRUE)
date.time <- paste(tdrX$date, tdrX$time)
tdr.time <- as.POSIXct(strptime(date.time, format="%d/%m/%Y %H:%M:%S"),
                       tz="GMT")
createTDR(tdr.time, tdrX$depth, concurrentData=data.frame(speed=tdrX$speed),
          file="dives.csv", speed=TRUE)
```

---

| rqPlot | *Plot of quantile regression for speed calibrations* |
|---|---|

---

**Description**

Plot of quantile regression for assessing quality of speed calibrations

**Usage**

```
rqPlot(rddepth, speed, z, contours, rqFit, main="qtRegression",
       xlab="rate of depth change (m/s)", ylab="speed (m/s)",
       colramp=colorRampPalette(c("white", "darkblue")),
       col.line="red", cex.pts=1)
```

**Arguments**

| | |
|---|---|
| speed | numeric vector with speed in m/s. |
| rddepth | numeric vector with rate of depth change. |
| z | list with the bivariate kernel density estimates (1st component the x points of the mesh, 2nd the y points, and 3rd the matrix of densities). |
| contours | list with components: `pts` which should be a matrix with columns named `x` and `y`, `level` a number indicating the contour level the points in `pts` correspond to. |
| rqFit | object of class "rq" representing a quantile regression fit of rate of depth change on mean speed. |
| main | character: string with title prefix to include in ouput plot. |

| | |
|---|---|
| xlab, ylab | character vectors with axis labels. |
| colramp | function taking an integer n as an argument and returning n colors. |
| col.line | color to use for the regression line. |
| cex.pts | numeric: value specifying the amount by which to enlarge the size of points. |

### Details

The dashed line in the plot represents a reference indicating a one to one relationship between speed and rate of depth change. The other line represent the quantile regression fit.

### Author(s)

Sebastian P. Luque <spluque@gmail.com>

### See Also

diveStats

---

| sealLocs | *Ringed and Gray Seal ARGOS Satellite Location Data* |
|---|---|

---

### Description

Satellite locations of a gray (Stephanie) and a ringed (Ringy) seal caught and released in New York.

### Format

A data frame with the following information:

**id** String naming the seal the data come from.

**time** The date and time of the location.

**class** The ARGOS location quality classification.

**lon, lat** x and y geographic coordinates of each location.

### Source

WhaleNet Satellite Tracking Program http://whale.wheelock.edu/Welcome.html.

### See Also

readLocs, distSpeed.

### Examples

```
sealLocs <- read.csv(system.file(file.path("data", "sealLocs.csv"),
                                 package="diveMove"), sep=";")
str(sealLocs)
```

TDR-accessors                    *Coerce, Extractor, and Replacement methods for class "TDR" objects*

### Description

Basic methods for manipulating objects of class TDR.

### Show Method

**show** signature(object="TDR"): print an informative summary of the data.

### Coerce Methods

**as.data.frame** signature(x="TDR"): Coerce object to data.frame. This method returns a
data frame, with attributes "file" and "dtime" indicating the source file and the interval between
samples.

**as.data.frame** signature(x="TDRspeed"): Coerce object to data.frame. Returns an object
as for TDR objects.

**as.TDRspeed** signature(x="TDR"): Coerce object to TDRspeed class.

### Extractor Methods

**[** signature(x="TDR"): Subset a TDR object; these objects can be subsetted on a single index
*i*. Selects given rows from object.

**getDepth** signature(x = "TDR"): depth slot accessor.

**getCCData** signature(x="TDR", y="missing"): concurrentData slot accessor.

**getCCData** signature(x="TDR", y="character"): access component named *y* in *x*.

**getDtime** signature(x = "TDR"): sampling interval accessor.

**getFileName** signature(x="TDR"): source file name accessor.

**getTime** signature(x = "TDR"): time slot accessor.

**getSpeed** signature(x = "TDRspeed"): speed accessor for TDRspeed objects.

### Replacement Methods

**depth<-** signature(x="TDR"): depth replacement.

**speed<-** signature(x="TDR"): speed replacement.

**ccData<-** signature(x="TDR"): concurrent data frame replacement.

### Author(s)

Sebastian P. Luque <spluque@gmail.com>

### See Also

extractDive, plotTDR.

## Examples

```
data(divesTDR)

## Retrieve the name of the source file
getFileName(divesTDR)
## Retrieve concurrent temperature measurements
temp <- getCCData(divesTDR, "temperature"); head(temp)
temp <- getCCData(divesTDR); head(temp)

## Coerce to a data frame
dives.df <- as.data.frame(divesTDR)
head(dives.df)

## Replace speed measurements
newspeed <- getSpeed(divesTDR) + 2
speed(divesTDR) <- newspeed
```

---

TDR-class  *Classes "TDR" and "TDRspeed" for representing TDR information*

---

### Description

These classes store information gathered by time-depth recorders.

### Details

Since the data to store in objects of these clases usually come from a file, the easiest way to construct such objects is with the function readTDR to retrieve all the necessary information. The methods listed above can thus be used to access all slots.

### Objects from the Class

Objects can be created by calls of the form new("TDR", ...) and new("TDRspeed", ...).

'TDR' objects contain concurrent time and depth readings, as well as a string indicating the file the data originates from, and a number indicating the sampling interval for these data. 'TDRspeed' extends 'TDR' objects containing additional concurrent speed readings.

### Slots

In class *TDR*:

file: Object of class 'character', string indicating the file where the data comes from.

dtime: Object of class 'numeric', sampling interval in seconds.

time: Object of class POSIXct, time stamp for every reading.

depth: Object of class 'numeric', depth (m) readings.

concurrentData: Object of class data.frame, optional data collected concurrently.

Class 'TDRspeed' must also satisfy the condition that a component of the concurrentData slot is named speed or velocity, containing the measured speed, a vector of class 'numeric' containing speed measurements in (m/s) readings.

**Author(s)**

Sebastian P. Luque <spluque@gmail.com>

**See Also**

readTDR, TDRcalibrate.

---

TDRcalibrate-accessors
                    *Methods to Show and Extract Basic Information from "TDRcalibrate"*
                    *Objects*

---

**Description**

Show and extract information from TDRcalibrate objects.

**Usage**

```
## S4 method for signature 'TDRcalibrate,missing':
getDAct(x)
## S4 method for signature 'TDRcalibrate,character':
getDAct(x, y)
## S4 method for signature 'TDRcalibrate,missing':
getDPhaseLab(x)
## S4 method for signature 'TDRcalibrate,numeric':
getDPhaseLab(x, diveNo)
## S4 method for signature 'TDRcalibrate,missing':
getDiveModel(x)
## S4 method for signature 'TDRcalibrate,numeric':
getDiveModel(x, diveNo)
## S4 method for signature 'diveModel':
getDiveDeriv(x, phase=c("all", "descent", "bottom", "ascent"))
## S4 method for signature 'TDRcalibrate':
getDiveDeriv(x, diveNo, phase=c("all", "descent", "bottom", "ascent"))
## S4 method for signature 'TDRcalibrate,missing':
getGAct(x)
## S4 method for signature 'TDRcalibrate,character':
getGAct(x, y)
```

## Arguments

| | |
|---|---|
| x | [TDRcalibrate](#) object. |
| diveNo | numeric vector with dive numbers to extract information from. |
| y | string; "dive.id", "dive.activity", or "postdive.id" in the case of getDAct, to extract the numeric dive ID, the factor identifying activity phases (with underwater and diving levels possibly represented), or the numeric postdive ID, respectively. In the case of getGAct it should be one of "phase.id", "activity", "begin", or "end", to extract the numeric phase ID for each observation, a factor indicating what major activity the observation corresponds to (where diving and underwater levels are not represented), or the beginning and end times of each phase in the record, respectively. |
| phase | character vector indicating phase of the dive for which to extract the derivative. |

## Value

The extractor methods return an object of the same class as elements of the slot they extracted.

## Show Methods

**show** signature(object="TDRcalibrate"): prints an informative summary of the data.

**show** signature(object="diveModel"): prints an informative summary of a dive model.

## Extractor Methods

**getDAct** signature(x="TDRcalibrate", y="missing"): this accesses the dive.activity slot of [TDRcalibrate](#) objects. Thus, it extracts a data frame with vectors identifying all readings to a particular dive and postdive number, and a factor identifying all readings to a particular activity.

**getDAct** signature(x="TDRcalibrate", y="character"): as the method for missing y, but selects a particular vector to extract. See [TDRcalibrate](#) for possible strings.

**getDPhaseLab** signature(x="TDRcalibrate", diveNo="missing"): extracts a factor identifying all readings to a particular dive phase. This accesses the dive.phases slot of [TDRcalibrate](#) objects, which is a factor.

**getDPhaseLab** signature(x="TDRcalibrate", diveNo="numeric"): as the method for missing y, but selects data from a particular dive number to extract.

**getDiveModel** signature(x="TDRcalibrate", diveNo="missing"): extracts a list with all dive phase models. This accesses the dive.models slot of [TDRcalibrate](#) objects.

**getDiveModel** signature(x="TDRcalibrate", diveNo="numeric"): as the method for missing diveNo, but selects data from a particular dive number to extract.

**getDiveDeriv** signature(x="TDRcalibrate"): extracts the derivative (list) of the dive model (smoothing spline) from the dive.models slot of [TDRcalibrate](#) objects for one or all phases of a dive.

**getDiveDeriv** signature(x="diveModel"): as the method for TDRcalibrate, but selects data from one or all phases of a dive.

**getGAct** signature(x="TDRcalibrate", y="missing"): this accesses the gross.activity slot of TDRcalibrate objects, which is a named list. It extracts elements that divide the data into major wet and dry activities.

**getGAct** signature(x="TDRcalibrate", y="character"): as the method for missing y, but extracts particular elements.

**getTDR** signature(x="TDRcalibrate"): this accesses the tdr slot of TDRcalibrate objects, which is a TDR object.

**getSpeedCoef** signature(x="TDRcalibrate"): this accesses the speed.calib.coefs slot of TDRcalibrate objects; the speed calibration coefficients.

### Author(s)

Sebastian P. Luque <spluque@gmail.com>

### See Also

diveModel, plotDiveModel, plotTDR.

### Examples

```
## Continuing the Example from '?calibrateDepth':
utils::example("calibrateDepth", package="diveMove",
               ask=FALSE, echo=FALSE)
dcalib # the 'TDRcalibrate' that was created

## Beginning times of each successive phase in record
getGAct(dcalib, "begin")

## Factor of dive IDs
dids <- getDAct(dcalib, "dive.id")
table(dids[dids > 0]) # samples per dive

## Factor of dive phases for given dive
getDPhaseLab(dcalib, 19)
## Full dive model
(dm <- getDiveModel(dcalib, 19))
str(dm)

## Derivatives
getDiveDeriv(dcalib, diveNo=19)
(derivs.desc <- getDiveDeriv(dcalib, diveNo=19, phase="descent"))
(derivs.bott <- getDiveDeriv(dcalib, diveNo=19, phase="bottom"))
(derivs.asc <- getDiveDeriv(dcalib, diveNo=19, phase="ascent"))
if (require(lattice)) {
    fl <- c("descent", "bottom", "ascent")
    bwplot(~ derivs.desc$y + derivs.bott$y + derivs.asc$y,
           outer=TRUE, allow.multiple=TRUE, layout=c(1, 3),
           xlab=expression(paste("Vertical rate (", m %.% s^-1, ")")),
           strip=strip.custom(factor.levels=fl))
}
```

TDRcalibrate-class    *Class "TDRcalibrate" for dive analysis*

**Description**

This class holds information produced at various stages of dive analysis. Methods are provided for extracting data from each slot.

**Details**

This is perhaps the most important class in diveMove, as it holds all the information necessary for calculating requested summaries for a TDR.

**Objects from the Class**

Objects can be created by calls of the form new("TDRcalibrate", ...). The objects of this class contain information necessary to divide the record into sections (e.g. dry/water), dive/surface, and different sections within dives. They also contain the parameters used to calibrate speed and criteria to divide the record into phases.

**Slots**

call: Object of class call.

The matched call to the function that created the object.

tdr: Object of class TDR.

This slot contains the time, zero-offset corrected depth, and possibly a data frame. If the object is also of class "TDRspeed", then the data frame might contain calibrated or uncalibrated speed. See readTDR and the accessor function getTDR for this slot.

gross.activity: Object of class 'list'.

This slot holds a list of the form returned by .detPhase, composed of 4 elements. It contains a vector (named phase.id) numbering each major activity phase found in the record, a factor (named activity) labelling each row as being dry, wet, or trivial wet activity. These two elements are as long as there are rows in tdr. This list also contains two more vectors, named begin and end: one with the beginning time of each phase, and another with the ending time; both represented as POSIXct objects. See .detPhase.

dive.activity: Object of class 'data.frame'.

This slot contains a data.frame of the form returned by .detDive, with as many rows as those in tdr, consisting of three vectors named: dive.id, which is an integer vector, sequentially numbering each dive (rows that are not part of a dive are labelled 0), dive.activity is a factor which completes that in activity above, further identifying rows in the record belonging to a dive. The third vector in dive.activity is an integer vector sequentially numbering each postdive interval (all rows that belong to a dive are labelled 0). See .detDive, and getDAct to access all or any one of these vectors.

dive.phases: Object of class 'factor'. This slot is a factor that labels each row in the record as
 belonging to a particular phase of a dive. It has the same form as the "phase.labels" component
 of the list returned by .labDivePhase.

dive.models: Object of class 'list'. This slot contains the details of the process of dive phase
 identification for each dive. It has the same form as the dive.models component of the
 list returned by .labDivePhase. It has as many components as there are dives in the TDR
 object, each of them of class diveModel.

dry.thr: Object of class 'numeric' the temporal criteria used for detecting dry periods that
 should be considered as wet.

wet.thr: Object of class 'numeric' the temporal criteria used for detecting periods wet that
 should not be considered as foraging time.

dive.thr: Object of class 'numeric' the criteria used for defining a dive.

speed.calib.coefs: Object of class 'numeric' the intercept and slope derived from the speed
 calibration procedure. Defaults to c(0, 1) meaning uncalibrated speeds.

### Author(s)

Sebastian P. Luque <spluque@gmail.com>

### See Also

TDR for links to other classes in the package. TDRcalibrate-methods for the various methods
available.

---

timeBudget-methods *Describe the Time Budget of Major Activities from "TDRcalibrate"*
 *object.*

---

### Description

Summarize the major activities recognized into a time budget.

### Usage

```
## S4 method for signature 'TDRcalibrate,logical':
timeBudget(obj, ignoreZ)
```

### Arguments

| | |
|---|---|
| obj | TDRcalibrate object. |
| ignoreZ | logical: whether to ignore trivial aquatic periods. |

### Details

Ignored trivial aquatic periods are collapsed into the enclosing dry period.

## Value

A data frame with components:

| | |
|---|---|
| `phaseno` | A numeric vector numbering each period of activity. |
| `activity` | A factor labelling the period with the corresponding activity. |
| `beg, end` | [POSIXct] objects indicating the beginning and end of each period. |

## Author(s)

Sebastian P. Luque <spluque@gmail.com>

## See Also

[calibrateDepth]

## Examples

```
## Continuing the Example from '?calibrateDepth':
utils::example("calibrateDepth", package="diveMove",
               ask=FALSE, echo=FALSE)
dcalib # the 'TDRcalibrate' that was created

timeBudget(dcalib, TRUE)
```

# Index