R Package Development by Means of Literate Programming `(noweb)`

Bernhard Pfaff

January 19, 2017

1 Introduction

The topic of this article is to elucidate the creation of the R package **hiker** by means of literate programming (henceforth: LP). LP has been introduced by Knuth (1984). Its constituent characteristic is that the source code of a program as well as the documentation thereof resides in the same file. To stay in the example of the **hiker** package: within the file you are currently reading, *i.e.*, `hiker.Rnw`, the whole R source code of the package as well as an explanation of it, is provided.[1] This file can then be processed by the facilities offered in the R package **Rnoweb** (see Ihaka, 2013). **Rnoweb** is a reimplementation of the `noweb` software (see Ramsey, 1994) written entirely in R. Creating the whole R package **hiker** by mere processing (tangling of the R code) of this file, is however not entirely true. Given that R packages must meet certain requirements with respect to the directory structure and the documentation of the provided functions, methods and classes, some additional tools become necessary. With respect to setting up the package structure and saving the produced R source files into the package's sub-directory `/R`, a `Makefile` is employed (the content of this file is provided in the appendix of this article, see section 5.5). The generation of the `man`-pages is accomplished by utilizing the facilities provided in the **roxygen2** and **devtools** packages (see Wickham et al., 2015; Wickham and Chang, 2016, respectively).

What is the package **hiker**(pronounce as 'hike—R') about? Within **hiker** the routines for detecting local peaks/troughs of a time series as proposed by Palshikar (2009) have been implemented. The knowledge of these points and the time spans in between these local maxima/minima can be utilized for analyzing the behavior of predictor variables in up-/down markets of a financial time series, *e.g.*, a stock price, for instance. Surely, these phases could in principal be detected by mere eyeball-econometrics, but this approach becomes burdensome if local peaks/troughs of many time series must be determined. The package itself is purely written in R and S4-classes and methods are employed.

In the remainder of this article, the proposed algorithms are presented in the subsequent section. The package structure, the defined S4-classes with the available methods are discussed in Section 3. In the Appendix of this paper the documentation of the package in the `roxygen`-format is provided. This

[1] This file and the requirements for generating the R package **hiker** are made available on github as project lp4rp.

organization has the advantage that the focus is shifted on the R code snippets only and is not interspersed with the documentation of the classes and methods and functions provided by the package.

2 Detecting Peaks/Troughs

2.1 Notation

A uniformly sampled time series $\mathbf{y} = \{y_1, \ldots, y_i, \ldots, y_T\}$ with T data points is considered. The detection of peak/trough points is achieved by a function $S(i, y_i, T)$ that returns for data point y_i a score value.[2]. If this score value surpasses a user-provided threshold value θ, *i.e*, $S(i, y_i, T) \geq \theta$ then the point is considered as a local peak/trough.

Furthermore, in case local peak/trough points appear closely together with respect to time (clustered), then these points can be classified as a burst or bust, respectively.

2.2 Algorithms

In Palshikar (2009) five different score functions S have been suggested. All have in common, that a centered window of size $2*k+1$ around y_i is considered. That is, for a positive integer k the k right neighbors $N^+(i, k, T) = \{y_{i+1}, \ldots, y_{i+k}\}$ and the k left neighbors $N^-(i, k, T) = \{\{y_{i-k}, \ldots, y_{i-1}\}$ are employed for assessing y_i as a local peak/trough. The union of $N^-(i, k, T)$ and $N^+(i, k, T)$ is defined as $N(i, k, T) = N^-(i, k, T) \cdot N^+(i, k, T)$ and if the center point is included as $N'(i, k, T) = N^-(i, k, T) \cdot y_i \cdot N^+(i, k, T)$.

The first function, S_1, computes the score value as the average of the maximum differences between y_i with its left and right neighbors. The function is defined as:

$$S_1 = \frac{\max(y_i - y_{i-1}, \ldots, y_i - y_{i-k}) + \max(y_i - y_{i+1}, \ldots, y_i - y_{i+k})}{2} \tag{1}$$

For a span of a time series $\mathbf{y}$ of length $2k+1$, denoted by `x`, the equation (1) can be casted in R as:

2 ⟨*score-maxdiff* 2⟩≡

```
scmaxdiff <- function(x, k){
    cp <- k + 1L
    lmax <- max(x[cp] - head(x, k))
    rmax <- max(x[cp] - tail(x, k))
    (lmax + rmax) / 2.0
}
```

Defines:
`scmaxdiff`, used in chunks 5–7.

[2]It suffices to provide a score function for peaks only. Trough points can be detected by using the negative values of the series $\mathbf{y}$.

Instead of using the maximum differences of y_i with its k left and right neighbors as in (1), an alternative is to compute the mean differences and evaluate the average thereof:

$$S_2 = \frac{\frac{(y_i - y_{i-1}, \ldots, y_i - y_{i-k})}{k} + \frac{(y_i - y_{i+1}, \ldots, y_i - y_{i+k})}{k}}{2} \tag{2}$$

This equation can be casted in R as:

3a ⟨*score-diffmean* 3a⟩≡

```
scdiffmean <- function(x, k){
    cp <- k + 1L
    ldmean <- x[cp] - mean(head(x, k))
    rdmean <- x[cp] - mean(tail(x, k))
    (ldmean + rdmean) / 2.0
}
```

Defines:
`scdiffmean`, used in chunks 5–7.

Another variation of score computation that has been proposed by Palshikar (2009) is to consider the differences to the mean of the k left and right neighbors, that is:

$$S_3 = \frac{(y_i - \frac{(y_{i-1}, \ldots, y_{i-k})}{k}) + (y_i - \frac{(y_{i+1}, \ldots, y_{i+k})}{k})}{2} \tag{3}$$

The equation (3) can be casted as R function `scavgdiff()` for instance as follows:

3b ⟨*score-avgdiff* 3b⟩≡

```
scavgdiff <- function(x, k){
    cp <- k + 1L
    lmean <- mean(x[cp] - head(x, k))
    rmean <- mean(x[cp] - tail(x, k))
    (lmean + rmean) / 2.0
}
```

Defines:
`scavgdiff`, used in chunks 5–7.

The fourth proposed score function differs from the previous three in the sense that it does take the differences between y_i and its neighbors explicitly into account, but tries to capture its information content by means of relative entropy. The entropy of a vector A with elements $A = \{a_1, \ldots, a_m\}$ is given as:

$$H_w(A) = \sum_{i=1}^{M} (-p_w(a_i) \log(p_w(a_i))) \tag{4}$$

where $p_w(a_i)$ is an estimate of the density value at a_i. The score function is now based on computing the entropies of $H(N((k, i, T))$ and $H(N'(k, i, T))$. Hereby, the densities can be determined by means of a kernel density estimator. The score function is then defined as the difference of the entropies:

$$S_4 = H(N((k, i, T)) - H(N'((k, i, T)) \tag{5}$$

This concept is implemented in the function `scentropy()`. The empirical density is computed by calling `density()`. The ellipsis argument of `scentropy()` is passed down to this function and hereby allowing the user to employ other than the default arguments of `density()`.

4 ⟨*score-entropy* 4⟩≡

```
scentropy <- function(x, k, ...){
    cp <- k + 1L
    dfull <- density(x, ...)$y
    hfull <- sum(-dfull * log(dfull))
    dexct <- density(x[-cp], ...)$y
    hexct <- sum(-dexct * log(dexct))
    hfull - hexct
}
```

Defines:
`scentropy`, used in chunks 5–7.

Finally, a moment-based score function has been put forward in the article by Palshikar. Hereby, the first and second moment of $N((k, i, T))$ are computed and a t-type statistic can be computed as $(y_i - m)/s$. If this statistic surpasses a provided threshold h, then the data point is considered as a local peak/trough.

$$S_5 = \begin{cases} 1 & (y_i - m)/s \geq h \\ 0 & \text{else} \end{cases} \tag{6}$$

This type of scoring algorithm is implemented as function `scttype()` below:

5a ⟨*score-ttype* 5a⟩≡

```
scttype <- function(x, k, tval){
    cp <- k + 1L
    m <- mean(x[-cp])
    s <- sd(x[-cp])
    tstat <- (x[cp] - m) / s
    if ( abs(tstat) < tval ){
        tstat <- 0
    }
    tstat
}
```

Defines:
`scttype`, used in chunks 5–7.

Incidentally, an ensemble forecast of these five algorithms can be utilized for local peak/trough classification. Hereby, one could either use a hybrid approach, whereby only those data points are considered as peak/trough points, if all five methods coincide. This concept is casted in the function `schybrid()`. Hereby, the signs of all five scoring algorithm are tested for equality.

5b ⟨*score-hybrid* 5b⟩≡

```
schybrid <- function(x, k, tval, ...){
    s <- c(sign(scmaxdiff(x, k)),
           sign(scavgdiff(x, k)),
           sign(scdiffmean(x, k)),
           sign(scentropy(x, k, ...)),
           sign(scttype(x, k, tval)))
    val <- unique(s)
    if ( length(val) < 2 ){
        return(s[1])
    } else {
        return(0)
    }
}
```

Defines:
`schybrid`, used in chunk 7.
Uses `scavgdiff` 3b, `scdiffmean` 3a, `scentropy` 4, `scmaxdiff` 2, and `scttype` 5a.

It is also conceivable to base the classification on a majority vote. For instance, if three out of the five algorithm classify a data point as a local peak/trough, then this is taken as sufficient evidence. This approach is defined in the function `scvote()` below. The count of same 'votes' is set by the argument `confby`. Its default value is 3, *i.e.*, a simple majority. For `confby = 5` the function would return the same classification as `schybrid()` does.

6 ⟨*score-vote* 6⟩≡

```
scvote <- function(x, k, tval, confby = 3, ...){
    s <- c(sign(scmaxdiff(x, k)),
           sign(scavgdiff(x, k)),
           sign(scdiffmean(x, k)),
           sign(scentropy(x, k, ...)),
           sign(scttype(x, k, tval)))
    pos <- rep(1, 5)
    zer <- rep(0, 5)
    neg <- rep(-1, 5)
    spos <- sum(s == pos)
    szer <- sum(s == zer)
    sneg <- sum(s == neg)
    v <- c(spos, szer, sneg)
    idx <- which(v >= confby)
    vals <- c(1, 0, -1)
    if ( length(idx) > 0 ){
        return(vals[idx])
    } else {
        return(0)
    }
}
```

Defines:
`scvote`, used in chunk 7.
Uses `scavgdiff` 3b, `scdiffmean` 3a, `scentropy` 4, `scmaxdiff` 2, and `scttype` 5a.

2.3 Combining score methods

What has been accomplished so far, are the function definitions of the five proposed heuristics and combinations thereof with respect to a sub-sample a time series of length $2k + 1$. These functions are now combined into a single `score()` function as shown in the subsequent R code chunk. The selection of the kind of score computation is conducted by calling the `switch()` function.

7 ⟨*score-wrapper* 7⟩≡

```
score <- function(x, k,
                  scoreby = c("vote", "avg", "diff", "max", "ent",
                              "ttype", "hybrid"),
                  tval = 1.0, confby = 3, ...){
    scoreby <- match.arg(scoreby)
    ans <- switch(scoreby,
                  vote = scvote(x, k, tval, confby, ...),
                  avg = scavgdiff(x, k),
                  diff = scdiffmean(x, k),
                  max = scmaxdiff(x, k),
                  ent = scentropy(x, k, ...),
                  ttype = scttype(x, k, tval),
                  hybrid = schybrid(x, k, tval, ...)
                  )
    ans
}
```

Defines:
`score`, used in chunk 8b.
Uses `scavgdiff` 3b, `scdiffmean` 3a, `scentropy` 4, `schybrid` 5b, `scmaxdiff` 2, `scttype` 5a, and `scvote` 6.

The functions for computing scores are grouped together in the R file `score.R` as shown in the next chunk. The function definitions are interspersed with the roxygen tags, which will be parsed to the `Rd`-file `score.Rd`. Incidentally, only code chunk with names ending in `.R` are treated as files that should be tangled and saved under the given code chunk's name.

8a ⟨*score.R* 8a⟩≡

```
⟨man-func-score 30⟩
⟨score-wrapper 7⟩
#' @rdname score
⟨score-maxdiff 2⟩
#' @rdname score
⟨score-diffmean 3a⟩
#' @rdname score
⟨score-avgdiff 3b⟩
#' @rdname score
⟨score-entropy 4⟩
#' @rdname score
⟨score-ttype 5a⟩
#' @rdname score
⟨score-hybrid 5b⟩
#' @rdname score
⟨score-vote 6⟩
```

This code is written to file `score.R`.

@

So far the wrapper function `score()` has been created, by which a single point is assessed for being a local maximum or minimum. For analyzing a whole time series for its local extrema, this routine can then be applied to each data point and its left/right neighbors. This task is accomplished with the function `hiker()` as defined next.

8b ⟨*hiker-func* 8b⟩≡

```
hiker <- function(y, k,
                  scoreby = c("vote", "avg", "diff", "max", "ent",
                              "ttype", "hybrid"),
                  tval = 1.0, confby = 3, ...){
⟨hiker-check 9⟩
    ## rolling centered window for peak scores
    s <- rollapply(y, width = ms, FUN = score,
                   k = k, scoreby = scoreby, tval = tval, ...)
⟨hiker-output 10a⟩
}
```

Uses `score` 7.

@

The arguments of the function are `y` for the time series object, `k` for the count of left/right neighbors, and `scoreby` for the selection of the scoring method. The arguments `tval` and `confby` belong the scoring concepts 'ttype' and 'hybrid', respectively, and the ellipsis argument is passed down to the call of `scentropy()` for `scoreby = 'ent'`.

The function body consists of three parts. First, the provided arguments are checked for their validity (as shown in the following code chunk). The computation of the scores is accomplished with the `rollapply()` function of the package **zoo** (see Zeileis and Grothendieck, 2005). Finally, the returned object is created.

9 ⟨*hiker-check* 9⟩≡

```
y <- as.zoo(y)
## checking arguments
k <- as.integer(abs(k))
ms <- 2 * k + 1L
if ( is.null(dim(y)) ){
    yname <- "series"
    n <- length(y)
    if ( n < ms ) {
        stop(paste("Sample size of 'y' is ", n,
                   " and k = ", k, ".\n", sep = ""))
        }
} else {
    n <- nrow(y)
    yname <- colnames(y)[1]
    if ( n < ms ) {
        stop(paste("Sample size of 'y' is ", n,
                   " and k = ", k, ".\n", sep = ""))
    }
    if ( ncol(y) > 1 ) {
        stop("Provide univariate time series of S3-class 'zoo'.\n")
    }
}
if ( (confby < 3) || (confby > 5) ){
    stop("\nArgument 'confby' must be integer and in set {3, 4, 5}.\n")
}
scoreby <- match.arg(scoreby)
```

@

Within the check section of the function body, the object `y` is first coerced to a `zoo` object and the count of neighbors is coerced to a positive integer. Next, the size of the sub-sample for computing the scores is assigned to the object `ms`. The remaining part consists of checks whether the series is univariate and its length is at least $2 \times k + 1$. Finally, the scoring method is determined from the argument `scoreby` by means of the `match.arg()` function.

In the final part of the function body of `hiker()`, the returned object is created as depicted in the subsequent R code chunk.

10a ⟨*hiker-output* 10a⟩≡

```
## merging time series and scores
ans <- merge(y, s)
colnames(ans) <- c("Series", "Scores")
des <- switch(scoreby,
              vote = "majority vote",
              avg = "average of averaged differences",
              diff = "average of mean differences",
              max = "average of maximum differences",
              ent = "difference of entropies",
              ttype = "t-type statistic",
              hybrid = "hybrid")
new("HikeR", ys = ans, k = k, scoreby = des, yname = yname)
```

@

First, the time series object `y` is merged with the series of the computed scores and assigned to `ans`. Next the type of scoring method is stored as character string to `des`. Finally, an object of S4-class `HikeR` is created and returned. The definition of this formal class is provided in section 3.2.

Given that `hiker()` is the core function of the package, it will be stored in its own `hiker.R` file, as commanded by the next chunk.

10b ⟨*hiker.R* 10b⟩≡

⟨*man-func-hiker* 31a⟩
⟨*hiker-func* 8b⟩

This code is written to file `hiker.R`.

@

3 Package structure

3.1 Preliminaries

R packages must contain a `DESCRIPTION` and a `NAMESPACE` file. Both reside in the package's root directory. As mentioned earlier, only chunks with names ending in '`.R`' are saved to disk by the tangling function of **Rnoweb**. Therefore, the content of the description file is first, casted in the `DESCRIPTION.R` file as shown below and is then moved by a directive in the `Makefile` to the root directory of the package **hiker** without the file's suffix.

11a ⟨*DESCRIPTION.R* 11a⟩≡

```
Package: hiker
Title: Local Peak and Trough of a Time Series
Version: 0.0.0.9000
Authors@R: person("Bernhard", "Pfaff", email = "bernhard@pfaffikus.de",
                  role = c("aut", "cre"))
Description: Methods for detecting local peaks and troughs of a time series.
Depends: R (>= 3.3.1), zoo, methods
License: GPL-3
Encoding: UTF-8
LazyData: true
```

This code is written to file `DESCRIPTION.R`.

@

With respect to the creation of the `NAMESPACE` file, matters are a bit different. The import/export directives contained in this file can be written as `roxygen` tags and the assembly of the tags and the saving of the file under the package's root directory is conducted by the `document()` function of **devtools**. Hence, it suffices to include the tags in a named chunk with suffix '`.R`'. In the case **hiker** the import directives are included in the R-file `Allclasses.R`. As is indicated by the file name, the definitions of the S4-classes will also reside in this file. The skeleton of this this file is provided in the subsequent chunk.

11b ⟨*Allclasses.R* 11b⟩≡

⟨*NAMESPACE* 12a⟩
⟨*HikerClass* 12b⟩
⟨*PtbbClass* 12c⟩

This code is written to file `Allclasses.R`.

@

The import directives for the required packages and functions are then given as:

12a ⟨*NAMESPACE* 12a⟩≡

```
#' @import methods
NULL
#' @import zoo
NULL
#' @importFrom graphics plot
NULL
#' @importFrom stats density sd na.omit start end smooth
NULL
#' @importFrom utils head tail
NULL

# Setting old (aka S3) classes
setOldClass("zoo")
```

@

3.2 S4-classes and generics

Two S4-classes are defined. The first, `HikeR`, has been mentioned briefly in the discussion of the core function `hiker()`. The class consists of four slots, as shown in the subsequent code chunk. Objects of the slot `ys` are of class `zoo` and contain the time series and the score value. The slot `k` is of type `integer` and holds the count of left/right neighbors. The character slot `scoreby` provides a literal description of the employed scoring algorithm and the slot `yname` is a character string of the time series name.

12b ⟨*HikerClass* 12b⟩≡

```
⟨man-class-HikeR 31b⟩
setClass("HikeR", slots = list(ys = "zoo",
                               k = "integer",
                               scoreby = "character",
                               yname = "character"))
```

@

The `HikeR`-class does not contain explicitly the information whether a certain data point is classified as a peak or trough, or does belong to a certain phase of the time series progression. This lack of explicit information is filled by the S4-class `PTBB`; it is a short-hand for 'PeaksTroughsBurstBust'. Its slot `pt` is of informal class `zoo` and is a logical whether a data point does belong to one the above mentioned points/phases. The slot `type` is a literal description thereof and the employed threshold value for making this classification is available in the slot `h`. The class definition is provided in the subsequent code chunk.

12c ⟨*PtbbClass* 12c⟩≡

```
⟨man-class-PTBB 32a⟩
setClass("PTBB", slots = list(pt = "zoo",
                              type = "character",
                              h = "numeric"))
```

@

Next, the following generic functions are defined in the file `Allgenerics.R`. The methods of these generics are discussed in the section 3.3.

13 ⟨*Allgenerics.R* 13⟩≡

```
# generic for extracting peaks
setGeneric("peaks", function(object, ...) standardGeneric("peaks"))
# generic for extracting troughs
setGeneric("troughs", function(object, ...) standardGeneric("troughs"))
# generic for extracting bursts
setGeneric("bursts", function(object, ...) standardGeneric("bursts"))
# generic for extracting busts
setGeneric("busts", function(object, ...) standardGeneric("busts"))
# generic for computing ridges
setGeneric("ridges", function(object, ...) standardGeneric("ridges"))
# generic for computing phases
setGeneric("phases", function(object, ...) standardGeneric("phases"))
# generic for extracting topeaks
setGeneric("topeaks", function(object, ...) standardGeneric("topeaks"))
# generic for extracting totroughs
setGeneric("totroughs", function(object, ...) standardGeneric("totroughs"))
# generic for computing runs
setGeneric("runs", function(x) standardGeneric("runs"))
```

This code is written to file `Allgenerics.R`.

@

3.3 Methods for S4-class 'HikeR'

In this section the S4-methods for objects of class `HikeR` are discussed. The provided methods are for showing `show()`, summarizing `summary()`, retrieval of peaks `peaks()` and troughs `troughs()` for this type of objects. Furthermore, the concept of bursts phases (close occurrence of peaks with respect to time) and busts (close occurrence of troughs with respect to time) are defined as methods `bursts()` and `busts()`, respectively. Additional methods for characterising the progression of a time series, such as 'ridges', 'phases', 'to-peaks' and 'to-troughs' are provided, too. Finally, a `plot()`-method is available whereby the user can hightlight/shade the local optima and the phases in between them. All of these methods are contained in the file `hiker-methods.R`. The skeleton of this file is provided next.

14a ⟨*HikerMethods.R* 14a⟩≡
⟨*HikeR-show* 14b⟩
⟨*HikeR-summary* 15a⟩
⟨*HikeR-peaks* 15b⟩
⟨*HikeR-troughs* 16a⟩
⟨*HikeR-bursts* 16b⟩
⟨*HikeR-busts* 17⟩
⟨*HikeR-ridges* 18⟩
⟨*HikeR-phases* 19⟩
⟨*HikeR-topeaks* 20⟩
⟨*HikeR-totroughs* 21⟩
⟨*HikeR-plot* 22a⟩

This code is written to file `HikerMethods.R`.

@

3.3.1 `show`-method

The `show`-method provides a brief overview of `HikeR`-objects. Its definition is given in the following chunk. The `roxygen` documentation is deferred to the chunk labeled `man-HikeR-show` and provided in the appendix 5.3.

14b ⟨*HikeR-show* 14b⟩≡

```
⟨man-HikeR-show 32b⟩
setMethod("show",
          signature(object = "HikeR"), function(object){
              cat(paste("Peak/trough score computed as: ",
                        object@scoreby, ".\n", sep = ""))
              cat(paste("Count of left/right neighbours: ", object@k,
                        ".\n", sep = ""))
              cat("\nSummary statistics of scores:\n")
              print(summary(object))
          }
)
```

@

First, the employed scoring-method and the count of left/right neighbors is printed *via* the `cat()` and `paste()` functions. Second, the `summary`-method is called, which provides descriptive statistics of the scores and is formally defined in the next section.

3.3.2 `summary`-method

The `summary`-method for numerics is applied to the second column of the slot `ys`, *i.e.*, the computed score values. These numbers are extracted from the `zoo` object by utilizing the `coredata()` function. Akin, to the `show`-method, the within-package documentation is deferred to the appendix.

15a ⟨*HikeR-summary* 15a⟩≡

```
⟨man-HikeR-summary 32c⟩
setMethod("summary",
          signature(object = "HikeR"),
          function (object, ...){
              summary(na.omit(coredata(object@ys[, 2])))
          }
)
```

@

3.3.3 `peaks`-method

The `peaks`-method is the first in a series of methods for classifying the data points in a time series. This and the following methods returns an object of formal class `PTBB`. Given that a peak is defined as a score value greater than a threshold (the default is `h = 0`), the core of the method is the logical statement in the first line of the function's body, whether this is true. The remaining slot entries of `PTBB` are set directly in the to `new()`.

15b ⟨*HikeR-peaks* 15b⟩≡

```
⟨man-HikeR-peaks 32d⟩
setMethod("peaks",
    signature(object = "HikeR"),
    function (object, h = 0) {
        ans <- object@ys[, 2] > h
        new("PTBB", pt = ans, type = "peak", h = h)
    }
)
```

@

As before, the `roxygen` documention of this method is listed in the appendix 5.3.

3.3.4 `troughs`-method

The mirror of peak detection is the trough classification. The method's body is hence defined as the flip-side of the former method, whereby the logical statement is casted now with respect to score values that are less than the threshold value. The default for the threshold is again set to `h = 0`. The computed logical array is now of type '`trough`' and the object of class `PTBB` is returned in the last line by invoking `new()`.

16a ⟨*HikeR-troughs* 16a⟩≡

```
⟨man-HikeR-troughs 32e⟩
setMethod("troughs",
    signature(object = "HikeR"),
    function (object, h = 0) {
        ans <- object@ys[, 2] < h
        new("PTBB", pt = ans, type = "trough", h = h)
    }
)
```

@

3.3.5 `bursts`-method

Bursts are characterized by clusters of local peaks. Whether the sequential occurrence of peaks do constitute a burst-phase, does depend on the maximum allowed distance between two consecutive peak points. This distance is depicted in the `burst`-method by the argument `b` with a default value of `k`, *i.e.* the count of considered left/right neighbors.

16b ⟨*HikeR-bursts* 16b⟩≡

```
⟨man-HikeR-bursts 32f⟩
setMethod("bursts",
    signature(object = "HikeR"),
    function (object, h = 0, b = object@k) {
        lpts <- object@ys[, 2] > h
        lidx <- which(lpts == TRUE)
        nidx <- length(lidx)
        if ( nidx > 1 ){
            for ( i in 2:nidx ){
                didx <- lidx[i] - lidx[i - 1]
                if ( didx <= b ){
                    lpts[(lidx[i]):(lidx[i - 1])] <- TRUE
                }
            }
        }
        ans <- zoo(lpts, order.by = index(object@ys))
        new("PTBB", pt = ans, type = "burst", h = h)
})
```

@

Within the definition of the method's body, first the peak points with respect to a threshold `h` are determined and assigned to the object `lpts`. Next, the index `lidx` of these occurrences is created and its length is saved in the object `nidx`. This information is then used in the following `if`-clause, to check whether any local peak points are existent in the first place, a necessary condition for the existence of burst phases. Within the `for`-loop the distance between two consecutive index entries is computed and compared to the bandwidth `b`. If the difference is less or equal to `b`, then the intermittent index entries are set to `TRUE`. Finally, an object of S4-class `PTBB` is created that holds as a logical the detected burst-phases.

3.3.6 `busts`-method

A bust episode is characterized by a concentration of trough points with respect to time. As such, it is the flip-side of burst phases. Hence, the skeleton and definition of the `bust`-method differs only from the `burst`-method with respect to the detection of local minimum values. Its definition is provided in the subsequent code chunk.

17 ⟨*HikeR-busts* 17⟩≡

```
⟨man-HikeR-busts 33a⟩
setMethod("busts",
    signature(object = "HikeR"),
    function (object, h = 0, b = object@k) {
        lpts <- object@ys[, 2] < h
        lidx <- which(lpts == TRUE)
        nidx <- length(lidx)
        if ( nidx > 1 ){
            for ( i in 2:nidx ){
                didx <- lidx[i] - lidx[i - 1]
                if ( didx <= b ){
                    lpts[(lidx[i]):(lidx[i - 1])] <- TRUE
                }
            }
        }
        ans <- zoo(lpts, order.by = index(object@ys))
        new("PTBB", pt = ans, type = "bust", h = h)
})
```

@

3.3.7 `ridges`-method

Data points of a time series can be classified as members of a ridge-like progression, if they exclusively neither belong to a burst or bust phase. This is given in a broader side-way movement. However, it can be possible burst- and bust phases are interlaced with each other (oscillating progression) and this case is also subsumed as a ridge. The `ridge`-method defined below, employs the previously discussed `burst` and `bust`-methods. The two logical arrays of the burst (object `burstp`) and bust (object `bustp`) points are then combined in the matrix `bbp`. The above stated rule is implemented by computing the row sums thereof and checking whether this is different from unity, in which case a data point is classified as being a member of a ridge-phase (object `ans`). The method does return an object of S4-class `PTBB` and the slot `type` is set to 'ridge'.

18 ⟨*HikeR-ridges* 18⟩≡

```
⟨man-HikeR-ridges 33b⟩
setMethod("ridges",
    signature(object = "HikeR"),
    function (object, h = 0, b = object@k) {
        N <- nrow(object@ys)
        k <- object@k
        bustp <- busts(object, h = h, b = b)@pt
        burstp <- bursts(object, h = h, b = b)@pt
        bbp <- cbind(bustp, burstp)
        ans <- zoo(FALSE, order.by = index(bustp))
        ridx <- which ( (rowSums(bbp) > 1) | (rowSums(bbp) < 1) )
        ans[ridx] <- TRUE
        ans[1:object@k] <- NA
        ans[(N - k + 1):N] <- NA
        new("PTBB", pt = ans, type = "ridge", h = h)
})
```

@

3.3.8 `phases`-method

The `phases`-method encompasses the previous three methods and classifies the phases of a time series into the categories 'burst', 'bust' and 'ridge'. The definition of this method is provided in the subsequent code chunk. First, a numeric `ans` is defined with a length equal to the size of the time series. Next, each of the three methods are called and the detected busts, bursts and ridges are assigned as character strings to the corresponding positions in `ans`. Finally, the `ans` numeric is casted as an object of informal class `zoo` and utilized in the creation of the `PTBB` S4-class obejct.

19 ⟨*HikeR-phases* 19⟩≡

```
⟨man-HikeR-phases 33c⟩
setMethod("phases",
    signature(object = "HikeR"),
    function (object, h = 0, b = object@k) {
        N <- nrow(object@ys)
        ans <- rep(NA, N)
        bustp <- busts(object, h = h, b = b)@pt
        ans[which(bustp == TRUE)] <- "bust"
        burstp <- bursts(object, h = h, b = b)@pt
        ans[which(burstp == TRUE)] <- "burst"
        ridgep <- ridges(object, h = h, b = b)@pt
        ans[which(ridgep == TRUE)] <- "ridge"
        ans <- factor(ans)
        ans <- zoo(ans, order.by = index(object@ys))
        new("PTBB", pt = ans, type = "phase", h = h)
})
```

@

3.3.9 `topeaks`-method

The `topeaks`-method can be utilized for detecting broader phases of up-movements in a time series. These singled out episodes could then be exploited for analysis the behavior of predictors in those periods. The flip side of this method is the detection of 'to-trough' periods and can be utilized in a similar fashion (this method is discussed in the next section).

The definition of the `topeaks`-method is shown below. First, a `zoo`-object is assigned to `ans` and all entries are set to `TRUE`. In the next two lines, the first and last k entries of the series are set to `NA`, motivated by the fact that the first and last left/right neighbors of a time series cannot be classified as a peak. This fact is due to the symmetric filtering of the time series for peaks. In the following lines the peak and trough points are determined and the positions thereof are assigned to the objects `pidx` and `tidx`, respectively. The length of each object, *i.e.*, the count, is assigned to `npidx` for the peaks and `ntidx` for the local trough points. Next sanity checks are conducted. If neither peak nor trough points are existent, then the method returns `NULL` with a corresponding warning.

20 ⟨*HikeR-topeaks* 20⟩≡

```
⟨man-HikeR-topeaks 33d⟩
setMethod("topeaks",
    signature(object = "HikeR"),
    function (object, h = 0) {
        N <- nrow(object@ys)
        k <- object@k
        ans <- zoo(rep(TRUE, N), order.by = index(object@ys))
        ans[1:k] <- NA
        ans[(N - k + 1):N] <- NA
        peakp <- peaks(object, h)@pt
        pidx <- which(peakp == TRUE)
        npidx <- length(pidx)
        troup <- troughs(object, h)@pt
        tidx <- which(troup == TRUE)
        ntidx <- length(tidx)
        if ( npidx == 0 ){
            warning("\nNo local peak points.\n")
            return(NULL)
        }
        if ( ntidx == 0 ){
            warning("\nNo local trough points.\n")
            return(NULL)
        }
        ## if trough comes first, set prior points to FALSE
        if ( tidx[1] < pidx[1] ){
            ans[(k + 1):tidx[1]] <- FALSE
        }
        for ( i in 1:ntidx ) {
            previouspeaks <- which(pidx < tidx[i])
```

```
            countpreviouspeaks <- length(previouspeaks)
            if ( countpreviouspeaks > 0 ){
                maxpos <- which.max(object@ys[pidx[previouspeaks], 1])
                ans[(pidx[maxpos] + 1):tidx[i]] <- FALSE
                pidx <- pidx[-c(1:countpreviouspeaks)]
            }
        }
        new("PTBB", pt = ans, type = "topeak", h = h)
})
```

@

In the remaining part of the method's body, the 'to-peaks' periods are set. First, it is checked whether the first trough point occurs earlier in time than a peak point. In this case, the entries of `ans` until this trough point are set to `FALSE`. In the following for-loop, the indices of the peaks that appeared earlier in a time than a trough point are assigned to `previouspeaks`. If this vector has length greater than zero, then the maximum index entry of the peaks (the last one prior to the i-th trough point) is assigned to `maxpos`. The entries in `ans` starting from the next period of this last peak point until the i-th trough are then set to `FALSE`. For house-keeping purposes the assessed peak-points in `pidx` are then removed from this index and a new loop-iterations commences. Finally, the method returns an `PTBB` S4-class object.

3.3.10 `totroughs`-method

For the detection of 'to-trough' phases the method `tottrough` is provided as shown below. Its definition is pretty straightforward, given that the 'to-trough' phases of a series are the complement of the 'to-peaks' periods. Having said this, the chest of this method is first to utilize the previously discussed `topeaks` method and then inverts the slot `pt` of the returned `PTBB` object.

21 ⟨*HikeR-totroughs* 21⟩≡

```
⟨man-HikeR-totroughs 33e⟩
setMethod("totroughs",
    signature(object = "HikeR"),
    function (object, h = 0) {
        ans <- topeaks(object, h)
        ans@pt <- !ans@pt
        ans@type <- "tottrough"
        ans
})
```

@

3.3.11 `plot`-method

Finally, a `plot`-method is defined for objects of S4-class `HikeR`. The skeleton is provided in the subsequent code chunk. Given that quite a few methods for the detection of local minima and maxima and the detection of certain phases have been defined for this kind of objects, a visualization thereof becomes necessary. The function's closure is endowed with arguments, such that basically all information can be graphically represented. In particular, the arguments `type` and `phase` are the two work-horses for the creation of the relevant plots. The remaining arguments can be utilized for custom adjustments with respect to the labeling and coloring of the plots. Each of the plot-types are discussed in the subsequent paragraphs.

22a ⟨*HikeR-plot* 22a⟩≡

```
⟨man-HikeR-plot 33f⟩
setMethod("plot",
    signature(x = "HikeR", y = "missing"),
    function (x, type = c("series", "score", "both", "zoo"),
              h = 0, b = x@k, phase = c("none", "pt", "bb"),
              main = NULL, sub = NULL,
              pt.peak = list(col = "darkgreen", pch = 19, cex = 0.8),
              pt.trough = list(col = "darkred", pch = 19, cex = 0.8),
              area.se = list(col = "lightgray"),
              area.pb = list(col = "seagreen", density = 20),
              area.tb = list(col = "red2", density = 20),
              ...){
        ⟨plotinit 22b⟩
        ⟨plotscore 23⟩
        ⟨plotseries 24⟩
        ⟨plotboth 28a⟩
        ⟨plotzoo 28b⟩
    }
)
```

@

In the first part of the method's body, the arguments are matched, where applicable and the size of the time series is determined, as well as the creation of an index (object `xidx`). Finally, a default sub-title is defined by which the employed scoring method and the count of left/right neighbors is set.

22b ⟨*plotinit* 22b⟩≡

```
type <- match.arg(type)
phase <- match.arg(phase)
N <- nrow(x@ys)
xidx <- 1:N
if ( is.null(sub) ){
    sub <- paste("Score method: ", x@scoreby,
                 ", k = ", x@k, sep = "")
}
```

@

A bar chart of the scores is produced if `type = 'score'` has been chosen. The commands for creating this type of plot are shown in the code chunk below. First, the range of the scores is determined, followed by the creation of a default main title. Next, an empty plot is created with limit margins according to the range of the score values. Rectangular areas are superimposed on this device for the first and last `k` data points, *i.e.*, the count of neighbors. Next, the score values are plotted as a bars (`type = 'h'`). The ellipsis argument is passed down to the `graphics::lines()` function. A horizontal line for the threshold value is superimposed in red. The drawing of a box and the labeling of the axis complete the plot of the score values.

23 ⟨*plotscore* 23⟩≡

```
if ( type == "score" ){
    srange <- range(na.omit(x@ys[, 2]))
    if ( is.null(main) ){
        main <- paste("Scores of time series:", x@yname)
    }
    plot(c(1, N), srange, axes = FALSE,
         main = main,
         sub = sub,
         xlab = "Time", ylab = "Score", type = "n")
    do.call(graphics::rect, c(list(xleft = 1,
                                   ybottom = srange[1],
                                   xright = x@k,
                                   ytop = srange[2]),
                              area.se))
    do.call(graphics::rect, c(list(xleft = N - x@k + 1,
                                   ybottom = srange[1],
                                   xright = N,
                                   ytop = srange[2]),
                              area.se))
    graphics::lines(coredata(x@ys[, 2]), type = "h", ...)
    graphics::abline(h = h, col = "red")
    graphics::box()
    graphics::axis(1, at = xidx, labels = index(x@ys), tick = FALSE)
    idx <- pretty(srange)
    graphics::axis(2, at = idx, labels = idx)
}
```

@

The default plot-type is a series plot (`type = 'series'`). The code thereof is shown in the subsequent code chunk. First, the range of the time series is determined and depending on the `phase` argument a default title is created unless one is provided by the user. Akin to the score plot, an empty plot device is created next with rectangular areas at the start and end of the sample period. The appearance of these is controlled by the list-type argument `area.se`. In the following code section, the time series is plotted with the `lines()` function from the graphics package and the peak/trough points are then determined and superimposed on the time series chart as points. The shape and color of these points can be adjusted by the arguments `pt.peak` and `pt.trough`, respectively.

24 ⟨*plotseries* 24⟩≡

```
if ( type == "series" ){
    yrange <- range(na.omit(x@ys[, 1]))
    if ( is.null(main) ){
        if ( phase == "none" ){
            main <- paste("Peaks and Troughs of:", x@yname)
        } else if ( phase == "pt" ){
            main <- paste("To-Peak and To-Trough Phases of:", x@yname)
        }
        else if ( phase == "bb") {
            main <- paste("Burst and Bust Phases of:", x@yname)
        }
    }
    plot(c(1, N), yrange, axes = FALSE,
         main = main, sub = sub,
         xlab = "Time", ylab = "", type = "n")
    do.call(graphics::rect, c(list(xleft = 1,
                                   ybottom = yrange[1],
                                   xright = x@k,
                                   ytop = yrange[2]),
                              area.se))
    do.call(graphics::rect, c(list(xleft = N - x@k + 1,
                                   ybottom = yrange[1],
                                   xright = N,
                                   ytop = yrange[2]),
                              area.se))
    graphics::lines(coredata(x@ys[, 1]), ...)
    peakidx <- which(peaks(x, h = h)@pt == TRUE)
    ynum <- coredata(x@ys[index(x@ys)[peakidx], 1])
    do.call(graphics::points, c(list(y = ynum, x = peakidx),
                                pt.peak))
    troughidx <- which(troughs(x, h = h)@pt == TRUE)
    ynum <- coredata(x@ys[index(x@ys)[troughidx], 1])
    do.call(graphics::points, c(list(y = ynum, x = troughidx),
                                pt.trough))
    ⟨plotbb 26⟩
    ⟨plotpt 27⟩
    graphics::box()
```

```
    graphics::axis(1, at = xidx, labels = index(x@ys),
                   tick = FALSE)
    idx <- pretty(yrange)
    graphics::axis(2, at = idx, labels = idx)
}
```

@

Depending on the value of the `phase` argument, either burst/bust or to-peak/to-trough periods are overlaid on the figure. The code thereof is presented in the next to two listings. Finally, a box is drawn around the figure region and axis labeling is applied to the graphic.

The burst and bust phases, if any, are first established by calling the `phases` method. The slot `pt` of the assigned object `p` is then investigated for entries equal to the character string 'burst' and a new object of S4-class `PTBB` is created. The start and end points of these burst phases are determined by calling the `runs`-method on this object. A description of the `runs`-method is provided in the following section. If burst phases are existent, then these are drawn as rectangles. Hereby, the user can alter the appearance by the `plot` argument `area.pb`. In a similar manner, the bust phases are localized and displayed on the figure.

26 ⟨*plotbb* 26⟩≡

```
if ( phase  == "bb" ){
    p <- phases(x, h = h, b = b)
    pchar <- as.character(coredata(p@pt))
    burstz <- new("PTBB",
                  pt = zoo(pchar == "burst",
                           order.by = index(p@pt)),
                  type = "burst",
                  h = p@h)
    burstr <- runs(burstz)
    if ( !is.null(burstr) ) {
        xleft <- which(as.character(index(burstz@pt)) %in%
                       as.character(burstr[, "From"]))
        xright <- which(as.character(index(burstz@pt)) %in%
                        as.character(burstr[, "To"]))
        for (i in 1:length(xleft)) {
            do.call(graphics::rect, c(list(xleft = xleft[i],
                                           ybottom = yrange[1],
                                           xright = xright[i],
                                           ytop = yrange[2]),
                                      area.pb))
        }
    }
    bustz <- new("PTBB",
                 pt = zoo(pchar == "bust",
                          order.by = index(p@pt)),
                 type = "bust",
                 h = p@h)
    bustr <- runs(bustz)
    if ( !is.null(bustr) ) {
        xleft <- which(as.character(index(bustz@pt)) %in%
                       as.character(bustr[, "From"]))
        xright <- which(as.character(index(bustz@pt)) %in%
                        as.character(bustr[, "To"]))
        for (i in 1:length(xleft)) {
```

```
            do.call(graphics::rect, c(list(xleft = xleft[i],
                                           ybottom = yrange[1],
                                           xright = xright[i],
                                           ytop = yrange[2]),
                                      area.tb))
        }
    }
}
```

@

Superimposing the 'to-peaks' and 'to-trough' phases is similar to the shading of the burst/bust phases. The code chunk is shown below. First, the method `topeaks` is invoked and the runs thereof are assigned to the object `topr`. Similarly, the ranges of the 'to-trough' periods are established and assigned to the object `totr`. Within the if-clauses the start and end point for each of the to-peaks and to-trough periods are stored to the objects `xleft` and `xright` and the regions are then plotted on top of the figure.

27 ⟨*plotpt* 27⟩≡

```
if ( phase == "pt" ){
    top <- topeaks(x, h)
    topr <- runs(top)
    tot <- totroughs(x, h)
    totr <- runs(tot)
    if ( !is.null(topr) ) {
        xleft <- which(as.character(index(top@pt)) %in%
                       as.character(topr[, "From"]))
        xright <- which(as.character(index(top@pt)) %in%
                        as.character(topr[, "To"]))
        for (i in 1:length(xleft)) {
            do.call(graphics::rect, c(list(xleft = xleft[i],
                                           ybottom = yrange[1],
                                           xright = xright[i],
                                           ytop = yrange[2]),
                                      area.pb))
        }
    }
    if ( !is.null(totr) ) {
        xleft <- which(as.character(index(tot@pt)) %in%
                       as.character(totr[, "From"]))
        xright <- which(as.character(index(tot@pt)) %in%
                        as.character(totr[, "To"]))
        for (i in 1:length(xleft)) {
            do.call(graphics::rect, c(list(xleft = xleft[i],
                                           ybottom = yrange[1],
                                           xright = xright[i],
                                           ytop = yrange[2]),
                                      area.tb))
        }
    }
}
```

@

The third plot-type is stacked version of the series and the score plot. This type is returned when `type = 'both'` has been set. The code snippet is shown below.

28a ⟨*plotboth* 28a⟩≡

```
if ( type == "both" ){
    op <- graphics::par(no.readonly = TRUE)
    graphics::par(mfrow = c(2, 1))
    plot(x, type = "series", h = h, ...)
    plot(x, type = "score", h = h, ...)
    graphics::par(op)
}
```

@

Finally, the series itself can be plotted by setting `type = 'zoo'`. Hereby, the `plot`-method of `zoo` objects is employed. The ellipsis argument is passed down to the call of this method, such that the user can make adjustments to the figure's appearance.

28b ⟨*plotzoo* 28b⟩≡

```
if ( type == "zoo" ){
    plot(x@ys, ...)
}
```

@

3.4 runs-method for S4-class 'PTBB'

As briefly mentioned in the previous section the `runs`-method determines the ranges of consecutive `TRUE` values in the slot `pt` of `PTBB` objects. Herby, the function `rle()` is utilized. The returned array of this function contains the information about runs as `'1'` values. These runs are assigned to the object `runs`. In the following lines of the method's body, the left and right points for each run are determined and a three-column data frame is created that holds in the first two columns the start and end periods of each run and the type of the `PTBB` object in its third. Should no runs exist, then the method returns `NULL` with a descriptive warning.

29 ⟨*PtbbMethods.R* 29⟩≡

```
⟨man-PTBB-runs 34a⟩
setMethod("runs",
          signature(x = "PTBB"), function(x){
              if ( any(na.omit(x@pt)) ){
                  p <- as.numeric(coredata(x@pt))
                  run <- rle(p)
                  runs <- which(run$values == 1)
                  cumidx <- cumsum(run$lengths)
                  n <- length(cumidx)
                  xidxright <- cumidx
                  xidxleft <- c(1, cumidx + 1)[-c(n + 1)]
                  xidx <- cbind(xidxleft, xidxright)
                  idx <- xidx[runs, ]
                  runscount <- nrow(xidx)
                  runsidx <- 1:runscount
                  ans <- sapply(runsidx, function(i)
                      x@pt[xidx[i, 1]:xidx[i, 2]])
                  pidx <- which(unlist(lapply(ans, function(r)
                      is.element(TRUE, r[1]))))
                  anstrue <- ans[pidx]
                  per <- lapply(anstrue, function(i)
                      data.frame(start(i), end(i)))
                  ans <- data.frame(do.call("rbind", per))
                  ans[, "Type"] <- x@type
                  colnames(ans) <- c("From", "To", "Type")
                  return(ans)
              } else {
                  warning("No runs in PTBB-object detected.\n")
                  return(NULL)
              }
          }
)
```

This code is written to file `PtbbMethods.R`.

@

4 Summary

5 Appendix

5.1 Documentation of functions

30 ⟨*man-func-score* 30⟩≡

```
#' Basic scoring methods for local minima and maxima
#'
#' These are basic functions for evaluating the centre
#' point of a time series as local minimum or maximum.
#' Hereby, a score value is computed according to various methods.
#' If the score is positive, the centre point is tentatively
#' classified as a local peak.
#' Incidentally, negative scores indicate a local minima.
#'
#' @param x \code{numeric}, vector of length \code{2 * k + 1}.
#' @param k \code{integer}, the count of left/right neighbours.
#' @param scoreby \code{character}, the scoring method to be used.
#' @param tval \code{numeric}, factor for standard deviation band
#' if \code{scoreby = 'ttype'}.
#' @param confby \code{integer}, count of minimum vote,
#' values in the set \code{3:5}.
#' @param ... ellipsis argument.
#'
#' @name score
#' @family scores
#' @return \code{numeric}, the score value.
NULL

#' @rdname score
#' @export
```

@

31a ⟨*man-func-hiker* 31a⟩≡

```
#' Peak/trough scores of time series points
#'
#' This function computes the score value for each
#' data point of a time series. The first and last
#' \code{k} observations are set to \code{NA}.
#'
#' @inheritParams score
#' @param y \code{zoo}, univariate time series.
#' @return An object of S4-class \code{HikeR}.
#' @family scores
#'
#' @references Girish K. Palshikar. Simple Algorithms for
#' Peak Detection in Time-Series. In \emph{Proc. 1st Int. Conf.
#' Advanced Data Analysis,
#' Business Analytics and Intelligence}, 2009.
#'
#' @examples
#' TEX <- SP500[, "TEX"]
#' ans <- hiker(TEX, k = 8, scoreby = "hybrid", tval = 0.1)
#' ans
#' plot(ans)
#'
#' @export
```

@

5.2 Documentation of S4-classes

31b ⟨*man-class-HikeR* 31b⟩≡

```
#' S4 class \code{HikeR}
#'
#' Formal class for classifying local minima and maxima
#' of a time series.
#'
#' @slot ys \code{zoo}, time series with associated scores.
#' @slot k \code{integer}, count of left/right neigbours around centre point.
#' @slot scoreby \code{character}, scoring method.
#' @slot yname \code{character}, name of the series.
#' @exportClass HikeR
```

@

32a ⟨*man-class-PTBB* 32a⟩≡

```
#' S4 class \code{PTBB}
#'
#' Formal class for peaks, troughs, burst, busts and
#' intermittent phase of a time series.
#'
#' @slot pt \code{zoo}, logical: indicating peak/trough points.
#' @slot type \code{character}, type of point/phase.
#' @slot h \code{numeric}, the threshhold for score evaluation.
#' @exportClass PTBB
```

@

5.3 Documentation of S4-methods

32b ⟨*man-HikeR-show* 32b⟩≡

```
#' @rdname HikeR-class
#' @param object An object of S4 class \code{HikeR}.
#' @export
```

@

32c ⟨*man-HikeR-summary* 32c⟩≡

```
#' @rdname HikeR-class
#' @aliases summary
#' @param ... Ellipsis argument.
#' @export
```

@

32d ⟨*man-HikeR-peaks* 32d⟩≡

```
#' @rdname HikeR-class
#' @aliases peaks
#' @param h \code{numeric}, the threshold value for scores
#' to be considered as peaks/troughs.
#' @return Object of S4-class \code{PTBB}.
#' @export
```

@

32e ⟨*man-HikeR-troughs* 32e⟩≡

```
#' @rdname HikeR-class
#' @aliases troughs
#' @export
```

@

32f ⟨*man-HikeR-bursts* 32f⟩≡

```
#' @rdname HikeR-class
#' @aliases bursts
#' @param b \code{integer}, intermittent count of points between peaks.
#' @export
```

@

33a ⟨*man-HikeR-busts* 33a⟩≡

```
#' @rdname HikeR-class
#' @aliases busts
#' @export
```

@

33b ⟨*man-HikeR-ridges* 33b⟩≡

```
#' @rdname HikeR-class
#' @aliases ridges
#' @export
```

@

33c ⟨*man-HikeR-phases* 33c⟩≡

```
#' @rdname HikeR-class
#' @aliases phases
#' @export
```

@

33d ⟨*man-HikeR-topeaks* 33d⟩≡

```
#' @rdname HikeR-class
#' @aliases topeaks
#' @export
```

@

33e ⟨*man-HikeR-totroughs* 33e⟩≡

```
#' @rdname HikeR-class
#' @aliases totroughs
#' @export
```

@

33f ⟨*man-HikeR-plot* 33f⟩≡

```
#' @rdname HikeR-class
#' @aliases plot
#' @param x An object of S4 class \code{HikeR}.
#' @param type \code{character}, whether series, scores or both should be plotted.
#' @param phase \code{character}, whether burst/bust or topeak/totrough phases
#' should be drawn in series plot.
#' @param pt.peak \code{list}, named elements are passed to \code{graphics::points()}.
#' @param main \code{character}, main title of plot.
#' @param sub \code{character}, sub title of plot
#' @param pt.trough \code{list}, named elements are passed to \code{graphics::points()}.
#' @param area.se \code{list}, named elements are passed to \code{graphics::rect()}
#' for areas of pre- and post sample points.
#' @param area.pb \code{list}, named elements are passed to \code{graphics::rect()}
#' for 'to-peak' or 'burst' phases.
#' @param area.tb \code{list}, named elements are passed to \code{graphics::rect()}
#' for 'to-trough' or 'bust' phases.
#' @export
```

@

34a ⟨*man-PTBB-runs* 34a⟩≡

```
#' @rdname PTBB-class
#' @aliases runs
#' @param x An object of S4 class \code{PTBB}.
#' @export
```

@

5.4 Documentation of data set

34b ⟨*data.R* 34b⟩≡

```
#' Weekly price data of 476 S&P 500 constituents.
#'
#' The data set was used in the reference below. The authors adjusted
#' the price data for dividends and have removed stocks if two or
#' more consecutive missing values were found. In the remaining cases
#' the NA entries have been replaced by interpolated values.
#'
#'
#' @format A S3-class \code{zoo} object with 265 weekly observations
#' of 476 members of the S&P 500 index. The sample starts at 2003-03-03
#' and ends in 2008-03-24.
#'
#' @references Cesarone, F. and Scozzari, A. and Tardella, F.: Portfolio
#'     selection problems in practice: a comparison between linear and
#'     quadratic optimization models, Working Paper, Universita degli
#'     Studi Roma Tre, Universita Telematica delle Scienze Umane and
#'     Universita di Roma, July 2010.
#'     \url{http://arxiv.org/ftp/arxiv/papers/1105/1105.3594.pdf}
#'
#' @source \url{http://w3.uniroma1.it/Tardella/datasets.html},\cr
#' \url{ http://finance.yahoo.com/}
"SP500"
```

This code is written to file `data.R`.

@

5.5 Makefile

```
# Makefile for creating the R package hiker

PKGNAME := hiker
PKGVERS = $(shell sed -n "s/Version: *\([^ ]*\)/\1/p" ./$(PKGNAME)/DESCRIPTION)
PKGTAR = $(PKGNAME)_$(PKGVERS).tar.gz
TEXCMD := pdflatex
RFILES := Allclasses.R Allgenerics.R score.R hiker.R data.R HikerMethods.R PtbbMethods.R
DFILES := SP500.rda

all: deps tex pdf pkg check
tex: $(PKGNAME).tex
pdf: $(PKGNAME).pdf

deps:
	Rscript -e 'if (!require("devtools")) install.packages("devtools")'
	Rscript -e 'if (!require("Rnoweb")) install.packages("Rnoweb_1.1.tar.gz", repos = NULL, type="source")'

$(PKGNAME).tex: $(PKGNAME).Rnw
	Rscript -e 'library(Rnoweb); noweb("$(PKGNAME).Rnw", tangle = FALSE)'

$(PKGNAME).pdf: $(PKGNAME).tex
	$(TEXCMD) $<
	$(TEXCMD) $<
	bibtex $(PKGNAME).aux
	$(TEXCMD) $<
	$(TEXCMD) $<

pkg: $(PKGNAME).Rnw
	Rscript -e 'library(Rnoweb); noweb("$(PKGNAME).Rnw", weave = FALSE)'
# creating package skeleton
	if [ ! -d "$(PKGNAME)" ]; then mkdir $(PKGNAME);  fi
	if [ ! -d "$(PKGNAME)/R" ]; then mkdir $(PKGNAME)/R;  fi
	if [ ! -d "$(PKGNAME)/data" ]; then mkdir $(PKGNAME)/data;  fi
# handling R files
	find ./$(PKGNAME)/R/ -type f -delete
	mv DESCRIPTION.R $(PKGNAME)/DESCRIPTION
	mv $(RFILES) $(PKGNAME)/R/
	cp $(DFILES) $(PKGNAME)/data/
# handling man files
	if [ ! -d "$(PKGNAME)/man" ]; then mkdir $(PKGNAME)/man;  fi
	find ./$(PKGNAME)/man/ -type f -delete
	Rscript -e 'library(devtools); devtools::document(pkg = "./$(PKGNAME)")'
# building the source tarball
	R CMD build $(PKGNAME)

check: pkg
	R CMD check $(PKGTAR)

clean:
	$(RM) -r $(PKGNAME).Rcheck/
	$(RM) $(PKGNAME).aux $(PKGNAME).log $(PKGNAME).out $(PKGNAME).bbl $(PKGNAME).blg
```

6 Chunk Index

⟨*score-avgdiff* 3b⟩
⟨*score-diffmean* 3a⟩
⟨*score-entropy* 4⟩
⟨*score-hybrid* 5b⟩
⟨*score-maxdiff* 2⟩
⟨*score-ttype* 5a⟩
⟨*score-vote* 6⟩
⟨*score-wrapper* 7⟩
⟨*score.R* 8a⟩

7 Identifier Index

References

Ihaka, R. (2013). *Rnoweb: Simplified Literate Programming*. R package version 1.1.

Knuth, D. E. (1984). Literate programming. *The Computer Journal 27*(2), 97–111.

Palshikar, G. (2009). Simple algorithms for peak detection in time-series. In *First Int. Conf. Advanced Data Analysis, Business Analytics and Intelligence*, Ahmedabad, India.

Ramsey, N. (1994, September). Literate programming simplified. *IEEE Software 11*(5), 97–105.

Wickham, H. and W. Chang (2016). *devtools: Tools to Make Developing R Packages Easier*. R package version 1.12.0.

Wickham, H., P. Danenberg, and M. Eugster (2015). *roxygen2: In-Source Documentation for R*. R package version 5.0.1.

Zeileis, A. and G. Grothendieck (2005). zoo: S3 infrastructure for regular and irregular time series. *Journal of Statistical Software 14*(6), 1–27.