Package 'clampSeg'

May 25, 2018

Title Idealisation of Patch Clamp Recordings

Version 1.0-4

Depends R (>= 3.0.0)

Imports stepR(>= 2.0.0), Rcpp (>= 0.12.3), stats, methods

LinkingTo Rcpp

Suggests testthat, R.cache (>= 0.10.0)

Description Allows for idealisation of patch clamp recordings by implementing the non-parametric JUmp Local dEconvolution Segmentation (JULES) filter, see F. Pein, I. Tecuapetla-Gómez, O. Schütte, C. Steinem, and A. Munk (2017) <arXiv:1706.03671>.

License GPL-3

Encoding UTF-8

LazyData true

NeedsCompilation yes

Author Pein Florian [aut, cre],
Thomas Hotz [ctb],
Inder Tecuapetla-Gómez [ctb],
Timo Aspelmeier [ctb]

Maintainer Pein Florian <fpein@uni-goettingen.de>

Repository CRAN

Date/Publication 2018-05-25 15:58:41 UTC

R topics documented:

clampSeg-package *Idealisation of patch clamp recordings*

Description

Allows for idealisation (fitting) of patch clamp (ion channel) recordings by implementing the **JU**mp **L**ocal d**E**convolution **S**egmentation (JULES) filter (*Pein et al.*, 2017) in the function `jules`. This non-parametric (model-free) segmentation method combines statistical multiresolution techniques with local deconvolution for idealising patch clamp recordings. In particular, also flickering (events on small time scales) can be detected and idealised.

Details

The main function of this package is `jules` which implements the **JU**mp **L**ocal d**E**convolution **S**egmentation (JULES) filter (*Pein et al.*, 2017). It reconstructs the signal underlying the data which is assumed to be a step (piecewise constant) function, e.g. constant conductance levels are assumed. The signal is pertubed by (Gaussian) white noise and convolved with a lowpass filter, resulting in a smooth signal pertubed by correlated noise with known correlation structure. The recorded data points are modelled as sampled (digitised) recordings of this process. For more details on this model see (*Pein et al.*, 2017, section II). A small example of such a recording, 3 seconds of a gramicidin A recording, is given by `gramA`.

The filter can be created by the function `lowpassFilter`, currently only Bessel filters are supported. The critical value `q` in (*Pein et al.*, 2017, (7)), the main parameter of JULES, can either be given by the user or be obtained by the function `getCritVal`, automatically called if required, in an universal manner by Monte-Carlo simulations such that (7) is a level `alpha`-test. The critical value `q`, or alternatively the significance level `alpha`, balances the risk of over- and underfitting. By default a small significance level of `alpha = 0.05` is chosen to guarantee that additional artificial changes are only be detected with a small probability. The critical value `q` and the Monte-Carlo simulations depend on the number of data points and the filter.

Since a Monte-Carlo simulation lasts potentially much longer (up to several hours or days if the number of observations is in the millions) than the main calculations, multiple possibilities for saving and loading the simulations are offered. Simulations can either be saved in the workspace in the variable `critValStepRTab` or persistently on the file system for which the package `R.cache` is used. Moreover, storing in and loading from variables and RDS files is supported. The simulation, saving and loading can be controlled by the argument `option`. By default simulations will be saved in the workspace and on the file system. For more details and for how simulation can be removed see the documentation of `getCritVal`.

The detection and estimation step of JULES can be obtained separately by the functions `stepDetection` and `deconvolveLocally`, respectively.

References

Pein, F., Tecuapetla-Gómez, I., Schütte, O., Steinem, C., Munk, A. (2017) Fully-automatic multiresolution idealization for filtered ion channel recordings: flickering event detection. *arXiv*:1706.03671.

Hotz, T., Schütte, O., Sieling, H., Polupanow, T., Diederichsen, U., Steinem, C., and Munk, A. (2013) Idealizing ion channel recordings by a jump segmentation multiresolution filter. *IEEE Transactions on NanoBioscience* **12**(4), 376–386.

Frick, K., Munk, A. and Sieling, H. (2014) Multiscale change-point inference. With discussion and rejoinder by the authors. *Journal of the Royal Statistical Society, Series B* **76**(3), 495–580.

Pein, F., Sieling, H. and Munk, A. (2016) Heterogeneous change point inference. *Journal of the Royal Statistical Society, Series B*, early view.

See Also

jules, critVal, lowpassFilter, gramA, deconvolveLocally, stepDetection

Examples

```
## idealisation of the gramicidin A recordings given by gramA
# the used filter
filter <- lowpassFilter(type = "bessel", param = list(pole = 4L, cutoff = 1e3 / 1e4),
                        sr = 1e4)

# the corresponding time points
time <- 9 + seq(along = gramA) / filter$sr

# plot of the data as in (Pein et al., 2017, figure 1 lower panel)
plot(time, gramA, pch = ".", col = "grey30", ylim = c(20, 50),
     ylab = "Conductance in pS", xlab = "Time in s")

# idealisation by JULES
# this call requires a Monte-Carlo simulation
# and therefore might last a few minutes,
# progress of the Monte-Carlo simulation is reported
idealisation <- jules(gramA, filter = filter, startTime = 9, messages = 100)

# this second call should be much faster
# as the previous Monte-Carlo simulation will be loaded
jules(gramA, filter = filter, startTime = 9)

# add idealisation to the plot
lines(idealisation, col = "#FF0000", lwd = 3)

# much larger significance level alpha for a larger detection power,
# but also with the risk of detecting additional artefacts
# in this example much more changes are detected,
# most of them are probably artefacts, but for instance the event at 11.36972
# might be an additional small event that was missed before
jules(gramA, filter = filter, alpha = 0.9, startTime = 9)

# getCritVal was called in jules, can be called explicitly
# for instance outside of a for loop to save computation time
q <- getCritVal(length(gramA), filter = filter)
identical(jules(gramA, q = q, filter = filter, startTime = 9), idealisation)

# both steps of JULES can be called separately
fit <- stepDetection(gramA, filter = filter, startTime = 9)
identical(deconvolveLocally(fit, data = gramA, filter = filter, startTime = 9),
          idealisation)
```

```
# more detailed output
each <- jules(gramA, filter = filter, startTime = 9, output = "each")
every <- jules(gramA, filter = filter, startTime = 9, output = "every")

identical(idealisation, each$idealization)
idealisationEvery <- every$idealization[[3]]
attr(idealisationEvery, "noDeconvolution") <- attr(every$idealization,
                                                   "noDeconvolution")
identical(idealisation, idealisationEvery)

identical(each$fit, fit)
identical(every$fit, fit)


## zoom into a single event, (Pein et al., 2017, figure 2 lower left panel)
plot(time, gramA, pch = 16, col = "grey30", ylim = c(20, 50),
     xlim = c(10.40835, 10.4103), ylab = "Conductance in pS", xlab = "Time in s")

# relevant part of the idealisation
cps <- idealisation$leftEnd[8:9]
levels <- idealisation$value[7:9]
t <- seq(cps[1] - 0.0009, cps[2] + 0.0023, 1e-6)

# idealisation
lines(t, ifelse(t < cps[1], rep(levels[1], length(t)),
                ifelse(t < cps[2], rep(levels[2], length(t)),
                       rep(levels[3], length(t)))),
      col = "#FF0000", lwd = 3)

# idealisation convolved with the filter
lines(t, levels[1] * (1 - filter$truncatedStepfun(t - cps[1])) +
        levels[2] * (filter$truncatedStepfun(t - cps[1]) -
                       filter$truncatedStepfun(t - cps[2])) +
        levels[3] * filter$truncatedStepfun(t - cps[2]),
      col = "#770000", lwd = 3)

# fit prior to the deconvolution step
# does not fit the recorded data points appropriately
cps <- fit$leftEnd[8:9]
levels <- fit$value[7:9]
t <- seq(cps[1] - 0.0009, cps[2] + 0.0023, 1e-6)

# fit
lines(t, ifelse(t < cps[1], rep(levels[1], length(t)),
                ifelse(t < cps[2], rep(levels[2], length(t)),
                       rep(levels[3], length(t)))),
      col = "blue", lwd = 3)

# fit convolved with the filter
lines(t, levels[1] * (1 - filter$truncatedStepfun(t - cps[1])) +
        levels[2] * (filter$truncatedStepfun(t - cps[1]) -
                       filter$truncatedStepfun(t - cps[2])) +
```

```
        levels[3] * filter$truncatedStepfun(t - cps[2]),
      col = "darkblue", lwd = 3)


## zoom into a single jump
plot(time, gramA, pch = 16, col = "grey30", ylim = c(20, 50),
     xlim = c(9.6476, 9.6496), ylab = "Conductance in pS", xlab = "Time in s")

# relevant part of the idealisation
cp <- idealisation$leftEnd[2]
levels <- idealisation$value[1:2]
t <- seq(cp - 0.0009, cp + 0.0023, 1e-6)

# idealisation
lines(t, ifelse(t < cp, rep(levels[1], length(t)), rep(levels[2], length(t))),
      col = "#FF0000", lwd = 3)

# idealisation convolved with the filter
lines(t, levels[1] * (1 - filter$stepfun(t - cp)) + levels[2] * filter$stepfun(t - cp),
      col = "#770000", lwd = 3)

# idealisation with a wrong filter
# does not fit the recorded data points appropriately
wrongFilter <- lowpassFilter(type = "bessel",
                             param = list(pole = 6L, cutoff = 0.2),
                             sr = 1e4)

# Monte-Carlo simulation depend on the number of observations and on the filter
# hence a simulation is required again (if called for the first time)
# to save some time the number of iterations is reduced to r = 1e3
# hence the critical value is computed with less precision
# In general, r = 1e3 is enough for a first impression
# for a detailed analysis r = 1e4 is suggested
idealisationWrong <- jules(gramA, filter = wrongFilter, startTime = 9,
                           r = 1e3, messages = 100)

# relevant part of the idealisation
cp <- idealisationWrong$leftEnd[2]
levels <- idealisationWrong$value[1:2]
t <- seq(cp - 0.0012, cp + 0.0023, 1e-6)

# idealisation
lines(t, ifelse(t < cp, rep(levels[1], length(t)), rep(levels[2], length(t))),
      col = "blue", lwd = 3)

# idealisation convolved with the filter
lines(t, levels[1] * (1 - filter$stepfun(t - cp)) + levels[2] * filter$stepfun(t - cp),
      col = "darkblue", lwd = 3)


# simulation for a larger number of observations can be used (nq = 3e4)
# does not require a new simulation as the simulation from above will be used
# (if the previous call was executed first)
```

```
jules(gramA[1:2.99e4], filter = wrongFilter, startTime = 9,
      nq = 3e4, r = 1e3, messages = 100)

# simulation of type "vectorIncreased" for n1 observations can only be reused
# for n2 observations if as.integer(log2(n1)) == as.integer(log2(n2))
# no simulation is required, since a simulation of type "matrixIncreased"
# will be loaded from the fileSystem
# this call also saves a simulation of type "vectorIncreased" in the workspace
jules(gramA[1:1e4], filter = filter, startTime = 9,
      nq = 3e4, messages = 100, r = 1e3)
# here a new simulation is required
# (if no appropriate simulation is saved from a call outside of this file)
jules(gramA[1:1e3], filter = filter, startTime = 9,
      nq = 3e4, messages = 100, r = 1e3,
      options = list(load = list(workspace = c("vector", "vectorIncreased"))))

# the above calls saved and (attempted to) load Monte-Carlo simulations
# in the following call the simulations will neither be saved nor loaded
jules(gramA, filter = filter, startTime = 9, messages = 100, r = 1e3,
      options = list(load = list(), save = list()))

# only simulations of type "vector" and "vectorInceased" will only be in and
# loaded from the workspace, but no simulations of type "matrix" and
# "matrixIncreased" on the file system
jules(gramA, filter = filter, startTime = 9, messages = 100,
      options = list(load = list(workspace = c("vector", "vectorIncreased")),
                     save = list(workspace = c("vector", "vectorIncreased"))))

# explicit Monte-Carlo simulations, not recommended
stat <- stepR::monteCarloSimulation(n = length(gramA), , family = "mDependentPS",
                                    filter = filter, output = "maximum",
                                    r = 1e3, messages = 100)
jules(gramA, filter = filter, startTime = 9, stat = stat)

# with given standard deviation
sd <- stepR::sdrobnorm(gramA, lag = filter$len + 1)
identical(jules(gramA, filter = filter, startTime = 9, sd = sd), idealisation)

# with less regularisation of the correlation matrix
jules(gramA, filter = filter, startTime = 9, regularization = 0.5)

# with estimation of the level of long segments by the mean
# but requiring 30 observations for it
jules(gramA, filter = filter, startTime = 9,
      localEstimate = mean, thresholdLongSegment = 30)

# with one refinement step less, but with a larger grid
# progress of the deconvolution is reported
# potential warning for no deconvolution is suppressed
jules(gramA, filter = filter, startTime = 9,
      gridSize = c(1 / filter$sr, 1 / 10 / filter$sr),
      windowFactorRefinement = 2, report = TRUE,
      suppressWarningNoDeconvolution = TRUE)
```

deconvolveLocally *Local deconvolution*

Description

Implements the estimation step of JULES (*Pein et al.*, 2017, section III B) in which an initial fit (reconstruction), e.g. computed by `stepDetection`, is refined by local deconvolution.

Usage

```
deconvolveLocally(fit, data, filter, startTime = 0, regularization = 1,
                  thresholdLongSegment = 10L, localEstimate = stats::median,
                gridSize = c(1 / filter$sr, 1 / 10 / filter$sr, 1 / 100 / filter$sr),
                  windowFactorRefinement = 1,
                  output = c("onlyIdealization", "everyGrid"), report = FALSE,
                  suppressWarningNoDeconvolution = FALSE)
```

Arguments

`fit`	an `stepblock` object or a list containing an entry `fit` with a `stepblock` object giving the initial fit (reconstruction), e.g. computed by `stepDetection`
`data`	a numeric vector containing the recorded data points
`filter`	an object of class `lowpassFilter` giving the used analogue lowpass filter
`startTime`	a single numeric giving the time at which recording (sampling) of `data` started, sampling time points will be assumed to be `startTime + seq(along = data) / filter$sr`
`regularization`	a single positive numeric or a numeric vector with positive entries or a `list` of length `length(gridSize)`, with each entry a single positive numeric or a numeric vector with positive entries, giving the regularisation added to the correlation matrix, see *details*. For a `list` the i-th entry will be used in the i-th refinement
`thresholdLongSegment`	a single integer giving the threshold determing how many observations are necessary to estimate a level (without deconvolution)
`localEstimate`	a function for estimating the levels of all long segments, see *details*, will be called with `localEstimate(data[i:j])` with `i` and `j` two integers in `1:length(data)` and `j - i >= thresholdLongSegment`
`gridSize`	a numeric vector giving the size of the grids in the iterative grid search, see *details*
`windowFactorRefinement`	a single numeric or a numeric vector of length `length(gridSize) - 1` giving factors for the refinment of the grid, see *details*. If a single numeric is given its value is used in all refinement steps
`output`	a string specifing the return type, see *Value*
`report`	a single `logical`, if `TRUE` the progress will be reported by `messages`

`suppressWarningNoDeconvolution`
: a single `logical`, if `FALSE` a `warning` will be given if at least one segment exists for which no deconvolution can be performed, since two short segments follow each other immediately

Details

The local deconvolution consists of two parts.
In the first part, all segments of the initial fit will be divided into long and short ones. The first and lasts `filter$len` data points of each segment will be ignored and if the remaining data points `data[i:j]` are at least `thresholdLongSegment`, i.e. `j - i + 1 >= thresholdLongSegment`, the level (value) of this segment will be determined by `localEstimate(data[i:j])`.
The long segments allow in the second part to perform the deconvolution locally by maximizing the likelihood function by an iterative grid search. Three scenarios might occur: Two long segments can follow each other, in this case the change, but no level, has to be estimated by maximizing the likelihood function of only few observations in this single parameter. A single short segment can be in between of two long segments, in this case two changes and one level have to be estimated by maximizing the likelihood function of only few observations in these three parameters. Finally, two short segments can follow each other, in this case no deconvolution is performed and the initial parameters are returned for these segments together with entries in the `"noDeconvolution"` `attribute`. More precisely, let `i:j` be the short segments, then `i:j` will be added to the `"noDeconvolution"` `attribute` and for the idealisation (if `output == "everyGrid"` this applies for each entry) the entries `value[i:j]`, `leftEnd[i:(j + 1)]` and `rightEnd[(i - 1):j]` are kept from the initial fit without refinement by deconvolution. If `suppressWarningNoDeconvolution == FALSE`, additionally, a `warning` will be given at first occurrence.
Maximisation of the likelihood is performed by minimizing (*Pein et al.*, 2017, (9)), a term of the form $x^T \Sigma x$, where Σ is the regularised correlation matrix and x a numeric vector of the same dimension. More precisely, the (unregularised) correlations are `filter$acf`, to this the regularisation `regularization` is added. In detail, if `regularization` is a numeric, the regularised correlation is

```
cor <- filter$acf
cor[seq(along = regularization)] <- cor[seq(along = regularization)] + regularization
```

and if `regularization` is a list the same, but `regularization` is in the `i-th` refinement replaced by `regularization[[i]]`. Then, Σ is a symmetric Toeplitz matrix with entries `cor`, i.e. a matrix with `cor[1]` on the main diagonal, `cor[2]` on the second diagonal, etc. and `0` for all entries outside of the first `length(cor)` diagonals.
The minimisations are performed by an iterative grid search: In a first step potential changes will be allowed to be at the grid / time points `seq(cp - filter$len / filter$sr, cp, gridSize[1])`, with `cp` the considered change of the initial fit. For each grid point in case of a single change and for each combination of grid points in case of two changes the term in (9) is computed and the change(s) for which the minimum is attained is / are chosen. Afterwards, refinements are done with the grids

```
seq(cp - windowFactorRefinement[j - 1] * gridSize[j - 1],
    cp + windowFactorRefinement[j - 1] * gridSize[j - 1],
    gridSize[j]),
```

with `cp` the change of the iteration before, as long as entries in `gridSize` are given.

Value

The idealisation (fit, regression) obtained by local deconvolution procedure of the estimation step of JULES. If `output == "onlyIdealization"` an object of class `stepblock` containing the final idealisation obtained by local deconvolution. If `output == "everyGrid"` a `list` of length `length(gridSize)` containing the idealisation after each refining step. Additionally, in both cases, an `attribute` `"noDeconvolution"`, an integer vector, gives the segments for which no deconvolution could be performed, since two short segments followed each other, see *details*.

References

Pein, F., Tecuapetla-Gómez, I., Schütte, O., Steinem, C., Munk, A. (2017) Fully-automatic multiresolution idealization for filtered ion channel recordings: flickering event detection. *arXiv*:1706.03671.

See Also

`jules`, `stepDetection`, `lowpassFilter`

Examples

```
## refinement of an initial fit of the gramicidin A recordings given by gramA
# the used filter
filter <- lowpassFilter(type = "bessel", param = list(pole = 4L, cutoff = 1e3 / 1e4),
                        sr = 1e4)
# initial fit
# with given q to save computation time
# this q is specific to length of the data and the filter
fit <- stepDetection(gramA, q = 1.370737, filter = filter, startTime = 9)

deconvolution <- deconvolveLocally(fit, data = gramA, filter = filter, startTime = 9)

# return fit after each refinement
every <- deconvolveLocally(fit, data = gramA, filter = filter, startTime = 9,
                           output = "every")

deconvolutionEvery <- every[[3]]
attr(deconvolutionEvery, "noDeconvolution") <- attr(every, "noDeconvolution")
identical(deconvolution, deconvolutionEvery)

# identical to a direct idealisation by jules
identical(jules(gramA, q = 1.370737, filter = filter, startTime = 9),
          deconvolution)

# plot of the data as in (Pein et al., 2017, figure 2 middle panel)
time <- 9 + seq(along = gramA) / filter$sr # time points
plot(time, gramA, pch = ".", col = "grey30", ylim = c(20, 50),
     ylab = "Conductance in pS", xlab = "Time in s")
lines(deconvolution, col = "#FF0000", lwd = 3)

## zoom into a single event, (Pein et al., 2017, figure 2 lower left panel)
plot(time, gramA, pch = 16, col = "grey30", ylim = c(20, 50),
     xlim = c(10.40835, 10.4103), ylab = "Conductance in pS", xlab = "Time in s")
```

```
# relevant part of the deconvolution
cps <- deconvolution$leftEnd[8:9]
levels <- deconvolution$value[7:9]
t <- seq(cps[1] - 0.0009, cps[2] + 0.0023, 1e-6)

# deconvolution
lines(t, ifelse(t < cps[1], rep(levels[1], length(t)),
                ifelse(t < cps[2], rep(levels[2], length(t)),
                       rep(levels[3], length(t)))),
      col = "#FF0000", lwd = 3)

# deconvolution convolved with the filter
lines(t, levels[1] * (1 - filter$truncatedStepfun(t - cps[1])) +
        levels[2] * (filter$truncatedStepfun(t - cps[1]) -
                       filter$truncatedStepfun(t - cps[2])) +
        levels[3] * filter$truncatedStepfun(t - cps[2]),
      col = "#770000", lwd = 3)

# fit prior to the deconvolution
# does not fit the recorded data points appropriately
cps <- fit$leftEnd[8:9]
levels <- fit$value[7:9]
t <- seq(cps[1] - 0.0009, cps[2] + 0.0023, 1e-6)

# fit
lines(t, ifelse(t < cps[1], rep(levels[1], length(t)),
                ifelse(t < cps[2], rep(levels[2], length(t)),
                       rep(levels[3], length(t)))),
      col = "blue", lwd = 3)

# fit convolved with the filter
lines(t, levels[1] * (1 - filter$truncatedStepfun(t - cps[1])) +
        levels[2] * (filter$truncatedStepfun(t - cps[1]) -
                       filter$truncatedStepfun(t - cps[2])) +
        levels[3] * filter$truncatedStepfun(t - cps[2]),
      col = "darkblue", lwd = 3)


## zoom into a single jump
plot(time, gramA, pch = 16, col = "grey30", ylim = c(20, 50),
     xlim = c(9.6476, 9.6496), ylab = "Conductance in pS", xlab = "Time in s")

# relevant part of the deconvolution
cp <- deconvolution$leftEnd[2]
levels <- deconvolution$value[1:2]
t <- seq(cp - 0.0009, cp + 0.0023, 1e-6)

# deconvolution
lines(t, ifelse(t < cp, rep(levels[1], length(t)), rep(levels[2], length(t))),
      col = "#FF0000", lwd = 3)

# deconvolution convolved with the filter
```

```
lines(t, levels[1] * (1 - filter$stepfun(t - cp)) + levels[2] * filter$stepfun(t - cp),
      col = "#770000", lwd = 3)

# deconvolution with a wrong filter
# does not fit the recorded data points appropriately
wrongFilter <- lowpassFilter(type = "bessel",
                             param = list(pole = 6L, cutoff = 0.2),
                             sr = 1e4)
deconvolutionWrong <- deconvolveLocally(fit, data = gramA, filter = wrongFilter,
                                        startTime = 9)

# relevant part of the deconvolution
cp <- deconvolutionWrong$leftEnd[2]
levels <- deconvolutionWrong$value[1:2]
t <- seq(cp - 0.0012, cp + 0.0023, 1e-6)

# deconvolution
lines(t, ifelse(t < cp, rep(levels[1], length(t)), rep(levels[2], length(t))),
      col = "blue", lwd = 3)

# deconvolution convolved with the filter
lines(t, levels[1] * (1 - filter$stepfun(t - cp)) + levels[2] * filter$stepfun(t - cp),
      col = "darkblue", lwd = 3)


# with less regularisation of the correlation matrix
deconvolveLocally(fit, data = gramA, filter = filter, startTime = 9,
                  regularization = 0.5)

# with estimation of the level of long segments by the mean
# but requiring 30 observations for it
deconvolveLocally(fit, data = gramA, filter = filter, startTime = 9,
                  localEstimate = mean, thresholdLongSegment = 30)

# with one refinement step less, but with a larger grid
# progress of the deconvolution is reported
# potential warning for no deconvolution is suppressed
deconvolveLocally(fit, data = gramA, filter = filter, startTime = 9,
                  gridSize = c(1 / filter$sr, 1 / 10 / filter$sr),
                  windowFactorRefinement = 2, report = TRUE,
                  suppressWarningNoDeconvolution = TRUE)
```

`getCritVal` *Critical value*

Description

Computes the critical value `q` in (*Pein et al.*, 2017, (7)) based on a Monte-Carlo simulation such that (7) is a level `alpha` test. Since a Monte-Carlo simulation lasts potentially much longer (up to several hours or days if the number of observations is in the millions) than the main calculations,

the simulations are by default saved in the workspace and on the file system such that a second call that require the same Monte-Carlo simulation will be much faster. For more details, in particular to which arguments the Monte-Carlo simulations are specific, see Section *Simulating, saving and loading of Monte-Carlo simulations* below. Progress of a Monte-Carlo simulation can be reported by the argument `messages` and the saving can be controlled by the argument `option`.

Usage

```
getCritVal(n, alpha = 0.05, filter, r = 1e4, nq = n, options = NULL, stat = NULL,
           messages = NULL)
```

Arguments

`n`	a positive integer giving the number of observations
`alpha`	a probability, i.e. a single numeric between 0 and 1, giving the significance level. Its choice is a trade-off between data fit and parsimony of the estimator. In other words, this argument balances the risks of missing changes and detecting additional artefacts. For more details on this choice see (Frick et al., 2014, section 4) and (Pein et al., 2017, section 3.4)
`filter`	an object of class `lowpassFilter` giving the used analogue lowpass filter
`r`	a positive integer giving the required number of Monte-Carlo simulations if they will be simulated or loaded from the workspace or the file system
`nq`	a positive integer larger than or equal to `n` giving the (increased) number of observations for the Monte-Carlo simulation. See Section *Simulating, saving and loading of Monte-Carlo simulations* for more details
`options`	a `list` specifying how Monte-Carlo simulations will be simulated, saved and loaded. For more details see Section *Simulating, saving and loading of Monte-Carlo simulations*
`stat`	an object of class `"MCSimulationVector"` or `"MCSimulationMaximum"`, usually computed by `monteCarloSimulation`. Has to be simulated for at least the given number of observations `n` and for the given `filter`. If missing it will be loaded and if not found simulated accordingly to the given `options`. For more details see Section *Simulating, saving and loading of Monte-Carlo simulations*
`messages`	a positive integer or `NULL`, in each `messages` iteration a message will be given in order to show the progress of the simulation, if `NULL` no message will be given

Value

A single numeric giving the critical value q in (*Pein et al.*, 2017, (7)).

Simulating, saving and loading of Monte-Carlo simulations

Since a Monte-Carlo simulation lasts potentially much longer (up to several hours or days if the number of observations is in the millions) than the main calculations, this function offers multiple possibilities to save and load the simulations. The simulation, saving and loading can be controlled by the argument `option`. This argument has to be a `list` or `NULL`. For the `list` the following named entries are allowed: `"simulation"`, `"save"`, `"load"`, `"envir"` and `"dirs"`. All missing

entries will be set to their default option.
Each Monte-Carlo simulation is specific to the number of observations and the used filter. Monte-Carlo simulations can also be performed for a (slightly) larger number of observations n_q given in the argument nq, which avoids extensive resimulations for only a little bit varying number of observations at price of a (slightly) smaller detection power. We recommend to not use a nq more than two times larger than the number of observations n.
Objects of the following types can be simulated, saved and loaded:

- "vector": an object of class "MCSimulationMaximum" for n observations, i.e. a numeric vector of length r
- "vectorIncreased": an object of class "MCSimulationMaximum" for nq observations, i.e. a numeric vector of length r
- "matrix": an object of class "MCSimulationVector" for n observations, i.e. a matrix of dimensions as.integer(log2(n)) + 1L and r
- "matrixIncreased": an object of class "MCSimulationVector" for nq observations, i.e. a matrix of as.integer(log2(n)) + 1L and r

Objects of class "MCSimulationVector" and objects of class "MCSimulationMaximum" lead to the same result (if the number of observations is the same), but an object of class "MCSimulationVector" requires much more storage space and has slightly larger saving and loading times. However, simulations of type "vectorIncreased", i.e. objects of class "MCSimulationMaximum" with nq observations, have to be resimulated if as.integer(log2(n1)) != as.integer(log2(n2)) when the saved simulation was computed with n == n1 and the simulation now is required for n == n2 and nq >= n1 and nq >= n2. All in all, if all data sets in the analysis have the same number of observations simulations of type "vector" are recommended. If they have a slightly different number of observations it is recommend to set nq to the largest number and to use simulations for an increased number of observations: If as.integer(log2(n)) is the same for all data sets type "vectorIncreased" is recommend, if they differ type "matrixIncreased" avoids a resimulation at the price of a larger object to be stored and loaded.
The simulations can either be saved in the workspace in the variable critValStepRTab or persistently on the file system for which the package R.cache is used. Loading from the workspace is faster, but either the user has to save the workspace manually or in a new session simulations have to be performed again. Moreover, storing in and loading from variables and RDS files is supported.

options$envir and options$dirs: For loading from / saving in the workspace the variable critValStepRTab in the environment options$envir will be looked for and if missing in case of saving also created there. Moreover, the variable(s) specified in options$save$variable (explained in the Subsection *Saving: options$save*) will be assigned to this environment. By default the global environment .GlobalEnv is used, i.e. options$envir == .GlobalEnv.
For loading from / saving on the file system loadCache(key = keyList, dirs = options$dirs) and saveCache(stat, key = attr(stat, "keyList"), dirs = options$dirs) are called, respectively. In other words, options$dirs has to be a character vector constituting the path to the cache subdirectory relative to the cache root directory as returned by getCacheRootPath(). If options$dirs == "", the path will be the cache root path. By default the subdirectory "stepR" is used, i.e. options$dirs == "stepR". Missing directories will be created.

Simulation: options$simulation: Whenever Monte-Carlo simulations have to be performed, i.e. when stat == NULL and the required Monte-Carlo simulation could not be loaded, the type specified in options$simulation will be simulated by monteCarloSimulation. In other words,

options$simulation must be a single string of the following: "vector", "vectorIncreased", "matrix" or "matrixIncreased". By default (options$simulation == NULL), an object of class "MCSimulationVector" for nq observations will be simulated, i.e. options$simulation == "matrixIncreased". For this choice please recall the explanations regarding computation time and flexibility at the beginning of this section.

Loading: options$load: Loading of the simulations can be controlled by the entry options$load which itself has to be a list with possible entries: "RDSfile", "workspace", "package" and "fileSystem". Missing entries disable the loading from this option. Whenever a Monte-Carlo simulation is required, i.e. when the variable q is not given, it will be searched for at the following places in the given order until found:

1. in the variable stat,
2. in options$load$RDSfile as an RDS file, i.e. the simulation will be loaded by
   readRDS(options$load$RDSfile).
   In other words, options$load$RDSfile has to be a connection or the name of the file where the R object is read from,
3. in the workspace or on the file system in the following order: "vector", "matrix", "vectorIncreased" and finally of "matrixIncreased". For each option it will first be looked in the workspace and then on the file system. All searches can be disabled by not specifying the corresponding string in options$load$workspace and options$load$fileSystem. In other words, options$load$workspace and options$load$fileSystem have to be vectors of strings containing none, some or all of "vector", "matrix", "vectorIncreased" and "matrixIncreased",
4. if all other options fail a Monte-Carlo simulation will be performed.

By default (if options$load is missing / NULL) no RDS file is specified and all other options are enabled, i.e.

```
options$load <- list(workspace = c("vector", "vectorIncreased",
                                   "matrix", "matrixIncreased"),
                     fileSystem = c("vector", "vectorIncreased",
                                    "matrix", "matrixIncreased"),
                     RDSfile = NULL).
```

Saving: options$save: Saving of the simulations can be controlled by the entry options$save which itself has to be a list with possible entries: "workspace", "fileSystem", "RDSfile" and "variable". Missing entries disable the saving in this option.
All available simulations, no matter whether they are given by stat, loaded, simulated or in case of "vector" and "vectorIncreased" computed from "matrix" and "matrixIncreased", respectively, will be saved in all options for which the corresponding type is specified. Here we say a simulation is of type "vectorIncreased" or "matrixIncreased" if the simulation is not performed for n observations. More specifically, a simulation will be saved:

1. in the workspace or on the file system if the corresponding string is contained in options$save$workspace and options$save$fileSystem, respectively. In other words, options$save$workspace and options$save$fileSystem have to be vectors of strings containing none, some or all of "vector", "matrix", "vectorIncreased" and "matrixIncreased",
2. in a variable named by options$save$variable in the environment options$envir. Hence, options$save$variable has to be a vector of one or two containing variable names (character vectors). If options$save$variable is of length two a simulation of type "vector" or "vectorIncreased" (only one can occur at one function call) will be saved in options$save$variable[1]

and "matrix" or "matrixIncreased" (only one can occur at one function call) will be saved in options$save$variable[2]. If options$save$variable is of length one both will be saved in options$save$variable which means if both occur at the same call only "vector" or "vectorIncreased" will be saved. Each saving can be disabled by not specifying options$save$variable or by passing "" to the corresponding entry of options$save$variable.

By default (if options$save is missing) "vector" and "vectorIncreased" will be saved in the workspace and "matrixIncreased" on the file system, i.e.

```
options$save <- list(workspace = c("vector", "vectorIncreased"),
                     fileSystem = c("matrix", "matrixIncreased"),
                     RDSfile = NULL, variable = NULL).
```

Simulations can be removed from the workspace by removing the variable critValStepRTab, i.e. by calling remove(critValStepRTab, envir = envir), with envir the used environment, and from the file system by deleting the corresponding subfolder, i.e. by calling

```
unlink(file.path(R.cache::getCacheRootPath(), dirs), recursive = TRUE),
```

with dirs the corresponding subdirectory.

References

Pein, F., Tecuapetla-Gómez, I., Schütte, O., Steinem, C., Munk, A. (2017) Fully-automatic multiresolution idealization for filtered ion channel recordings: flickering event detection. *arXiv*:1706.03671.

Frick, K., Munk, A., Sieling, H. (2014) Multiscale change-point inference. With discussion and rejoinder by the authors. *Journal of the Royal Statistical Society, Series B* **76**(3), 495–580.

Pein, F., Sieling, H., Munk, A. (2017) Heterogeneous change point inference. *Journal of the Royal Statistical Society, Series B*, **79**(4), 1207–1227.

See Also

jules, lowpassFilter, stepDetection

Examples

```
# the for the recording of the gramA data set used filter
filter <- lowpassFilter(type = "bessel", param = list(pole = 4L, cutoff = 1e3 / 1e4),
                        sr = 1e4)

# critical value for jules or stepDetection
# this call requires a Monte-Carlo simulation at the first time
# and therefore might last a few minutes,
# progress of the Monte-Carlo simulation is reported
q <- getCritVal(length(gramA), filter = filter, messages = 100)

# this second call should be much faster
# as the previous Monte-Carlo simulation will be loaded
getCritVal(length(gramA), filter = filter)

# much larger significance level alpha for a larger detection power,
# but also with the risk of detecting additional artefacts
getCritVal(length(gramA), filter = filter, alpha = 0.9)
```

```
# medium significance level alpha for a tradeoff between detection power
# and the risk to detect additional artefacts
getCritVal(length(gramA), filter = filter, alpha = 0.5)

# critical values depend on the number of observations and on the filter
# also a new Monte-Carlo simulation is required
getCritVal(100, filter = filter, messages = 500)

otherFilter <- lowpassFilter(type = "bessel",
                             param = list(pole = 6L, cutoff = 0.2),
                             sr = 1e4)
getCritVal(100, filter = otherFilter, messages = 500)

# simulation for a larger number of oberservations can be used (nq = 100)
# does not require a new simulation as the simulation from above will be used
# (if the previous call was executed first)
getCritVal(90, filter = filter, nq = 100)

# simulation of type "vectorIncreased" for n1 observations can only be reused
# for n2 observations if as.integer(log2(n1)) == as.integer(log2(n2))
# no simulation is required, since a simulation of type "matrixIncreased"
# will be loaded from the fileSystem
# this call also saved a simulation of type "vectorIncreased" in the workspace
getCritVal(30, filter = filter, nq = 100)
# here a new simulation is required
# (if no appropriate simulation is saved from a call outside of this file)
getCritVal(10, filter = filter, nq = 100, messages = 500,
           options = list(load = list(workspace = c("vector", "vectorIncreased"))))

# the above calls saved and (attempted to) load Monte-Carlo simulations
# in the following call the simulations will neither be saved nor loaded
# to save some time the number of iterations is reduced to r = 1e3
# hence the critical value is computed with less precision
# In general, r = 1e3 is enough for a first impression
# for a detailed analysis r = 1e4 is suggested
getCritVal(100, filter = filter, messages = 100, r = 1e3,
           options = list(load = list(), save = list()))

# simulations will only be saved in and loaded from the workspace,
# but not on the file system
getCritVal(100, filter = filter, messages = 100, r = 1e3,
           options = list(load = list(workspace = c("vector", "vectorIncreased")),
                          save = list(workspace = c("vector", "vectorIncreased"))))

# explicit Monte-Carlo simulations, not recommended
stat <- stepR::monteCarloSimulation(n = 100, , family = "mDependentPS",
                                    filter = filter, output = "maximum",
                                    r = 1e3, messages = 100)
getCritVal(100, filter = filter, stat = stat)
```

gramA *Patch clamp recording of gramicidin A*

Description

3 second part of a patch clamp recording of gramicidin A with solvent-free lipid bilayers using the Port-a-Patch measured in the Steinem lab (Institute of Organic and Biomolecular Chemistry, University of Goettingen). All rights reserved by them. The recorded data points are a conducatance trace in pico Siemens and were recorded at a sampling rate of 10 kHz using a 1 kHz 4-pole Bessel filter. More details of the recording can be found in (*Pein et al.*, 2017, section V A) and a plot in the examples or in (*Pein et al.*, 2017, figure 1 lower panel).

Usage

```
gramA
```

Format

A numeric vector containing 30,000 values.

References

Pein, F., Tecuapetla-Gómez, I., Schütte, O., Steinem, C., Munk, A. (2017) Fully-automatic multiresolution idealization for filtered ion channel recordings: flickering event detection. *arXiv*:1706.03671.

Examples

```
# the recorded data points
gramA

# the used filter
filter <- lowpassFilter(type = "bessel", param = list(pole = 4L, cutoff = 1e3 / 1e4),
                        sr = 1e4)

# the corresponding time points
time <- 9 + seq(along = gramA) / filter$sr

# plot of the data as in (Pein et al., 2017, figure 1 lower panel)
plot(time, gramA, pch = ".", col = "grey30", ylim = c(20, 50),
     ylab = "Conductance in pS", xlab = "Time in s")
```

jules *JULES*

Description

Implements the **JU**mp **L**ocal d**E**convolution **S**egmentation (JULES) filter (*Pein et al.*, 2017). This non-parametric (model-free) segmentation method combines statistical multiresolution techniques with local deconvolution for idealising patch clamp (ion channel) recordings. In particular, also flickering (events on small time scales) can be detected and idealised which is not possible with common thresholding methods.
If `q == NULL` a Monte-Carlo simulation is required for computing the critical value. Since a Monte-Carlo simulation lasts potentially much longer (up to several hours or days if the number of observations is in the millions) than the main calculations, this package saves them by default in the workspace and on the file system such that a second call requiring the same Monte-Carlo simulation will be much faster. For more details, in particular to which arguments the Monte-Carlo simulations are specific, see Section *Storing of Monte-Carlo simulations* below. Progress of a Monte-Carlo simulation can be reported by the argument `messages` and the saving can be controlled by the argument `option`, both can be specified in `...` and are explained in `getCritVal`.

Usage

```
jules(data, filter, q = NULL, alpha = 0.05, sd = NULL, startTime = 0,
      output = c("onlyIdealization", "eachStep", "everything"), ...)
```

Arguments

`data`	a numeric vector containing the recorded data points
`filter`	an object of class `lowpassFilter` giving the used analogue lowpass filter
`q`	a single numeric giving the critical value q in (*Pein et al.*, 17, (7)), by default chosen automatically by `getCritVal`
`alpha`	a probability, i.e. a single numeric between 0 and 1, giving the significance level to compute the critical value `q` (if `q == NULL`), see `getCritVal`. Its choice is a trade-off between data fit and parsimony of the estimator. In other words, this argument balances the risks of missing changes and detecting additional artefacts. For more details on this choice see (Frick et al., 2014, section 4) and (Pein et al., 2016, section 3.4)
`sd`	a single positive numeric giving the standard deviation (noise level) σ_0 of the data points before filtering, by default (NULL) estimated by `sdrobnorm` with `lag = filter$len + 1L`
`startTime`	a single numeric giving the time at which recording (sampling) of `data` started, sampling time points will be assumed to be `startTime + seq(along = data) / filter$sr`
`output`	a string specifing the return type, see *Value*
`...`	additional parameters to be passed to `getCritVal` or `deconvolveLocally`:

1. getCritVal will be called automatically (if q == NULL), the number of data points n = length(data) will be set and alpha and filter will be passed. For these parameter no user interaction is required and possible, all other parameters of getCritVal can be passed additionally
2. deconvolveLocally will be called automatically, the by stepDetection computed reconstruction / fit will be passed to fit and data, filter, startTime will be passed and output will be set accordingly to the output argument. For these parameter no user interaction is required and possible, all other parameters of deconvolveLocally can be passed additionally

Value

The idealisation (estimation, regression) obtained by JULES. If output == "onlyIdealization" an object object of class stepblock containing the idealisation. If output == "eachStep" a list containing the entries idealization with the idealisation, fit with the fit obtained by the detection step only, q with the given / computed critical value, filter with the given filter and sd with the given / estimated standard deviation. If output == "everything" a list containing the entries idealization with a list containing the idealisation after each refining step in the local deconvolution, fit with the fit obtained by the detection step only, stepfit with the fit obtained by the detection step before postfiltering, q with the given / computed critical value, filter with the given filter and sd with the given / estimated standard deviation. Additionally, in all cases, the idealisation has an attribute "noDeconvolution", an integer vector, that gives the segments for which no deconvolution could be performed, since two short segments followed each other, see also *details* in deconvolveLocally.

Storing of Monte-Carlo simulations

If q == NULL a Monte-Carlo simulation is required to compute the critical value. Since a Monte-Carlo simulation lasts potentially much longer (up to several hours or days if the number of observations is in the millions) than the main calculations, multiple possibilities for saving and loading the simulations are offered. Progress of a simulation can be reported by the argument messages which can be specified in ... and is explained in the documentation of getCritVal. Each Monte-Carlo simulation is specific to the number of observations and the used filter. But note that also Monte-Carlo simulations for a (slightly) larger number of observations n_q, given in the argument nq in ... and explained in the documentation of getCritVal, can be used, which avoids extensive resimulations for only a little bit varying number of observations, but results in a (small) loss of power. However, simulations of type "vectorIncreased", i.e. objects of class "MCSimulationMaximum" with nq observations, have to be resimulated if as.integer(log2(n1)) != as.integer(log2(n2)) when the saved simulation was computed with n == n1 and the simulation now is required for n == n2 and nq >= n1 and nq >= n2. Simulations can either be saved in the workspace in the variable critValStepRTab or persistently on the file system for which the package R.cache is used. Moreover, storing in and loading from variables and RDS files is supported. The simulation, saving and loading can be controlled by the argument option which can be specified in ... and is explained in the documentation of getCritVal. By default simulations will be saved in the workspace and on the file system. For more details and for how simulation can be removed see Section *Simulating, saving and loading of Monte-Carlo simulations* in getCritVal.

References

Pein, F., Tecuapetla-Gómez, I., Schütte, O., Steinem, C., Munk, A. (2017) Fully-automatic multiresolution idealization for filtered ion channel recordings: flickering event detection. *arXiv*:1706.03671.

See Also

critVal, lowpassFilter, deconvolveLocally, stepDetection

Examples

```
## idealisation of the gramicidin A recordings given by gramA with jules
# the used filter
filter <- lowpassFilter(type = "bessel", param = list(pole = 4L, cutoff = 1e3 / 1e4),
                        sr = 1e4)

# idealisation by JULES
# this call requires a Monte-Carlo simulation
# and therefore might last a few minutes,
# progress of the Monte-Carlo simulation is reported
idealisation <- jules(gramA, filter = filter, startTime = 9, messages = 100)

# this second call should be much faster
# as the previous Monte-Carlo simulation will be loaded
jules(gramA, filter = filter, startTime = 9)

# plot of the data as in (Pein et al., 2017, figure 2 middle panel)
time <- 9 + seq(along = gramA) / filter$sr # time points
plot(time, gramA, pch = ".", col = "grey30", ylim = c(20, 50),
     ylab = "Conductance in pS", xlab = "Time in s")
lines(idealisation, col = "#FF0000", lwd = 3)

# much larger significance level alpha for a larger detection power,
# but also with the risk of detecting additional artefacts
# in this example much more changes are detected,
# most of them are probably artefacts, but for instance the event at 11.36972
# might be an additional small event that was missed before
jules(gramA, filter = filter, alpha = 0.9, startTime = 9)

# getCritVal was called in jules, can be called explicitly
# for instance outside of a for loop to save computation time
q <- getCritVal(length(gramA), filter = filter)
identical(jules(gramA, q = q, filter = filter, startTime = 9), idealisation)

# more detailed output with information about the single steps
each <- jules(gramA, filter = filter, startTime = 9, output = "each")
every <- jules(gramA, filter = filter, startTime = 9, output = "every")

identical(idealisation, each$idealization)
idealisationEvery <- every$idealization[[3]]
attr(idealisationEvery, "noDeconvolution") <- attr(every$idealization,
                                                   "noDeconvolution")
identical(idealisation, idealisationEvery)
```

```
fit <- stepDetection(gramA, filter = filter, startTime = 9)
identical(each$fit, fit)
identical(every$fit, fit)


## zoom into a single event, (Pein et al., 2017, figure 2 lower left panel)
plot(time, gramA, pch = 16, col = "grey30", ylim = c(20, 50),
     xlim = c(10.40835, 10.4103), ylab = "Conductance in pS", xlab = "Time in s")

# relevant part of the idealisation
cps <- idealisation$leftEnd[8:9]
levels <- idealisation$value[7:9]
t <- seq(cps[1] - 0.0009, cps[2] + 0.0023, 1e-6)

# idealisation
lines(t, ifelse(t < cps[1], rep(levels[1], length(t)),
                ifelse(t < cps[2], rep(levels[2], length(t)),
                       rep(levels[3], length(t)))),
      col = "#FF0000", lwd = 3)

# idealisation convolved with the filter
lines(t, levels[1] * (1 - filter$truncatedStepfun(t - cps[1])) +
        levels[2] * (filter$truncatedStepfun(t - cps[1]) -
                       filter$truncatedStepfun(t - cps[2])) +
        levels[3] * filter$truncatedStepfun(t - cps[2]),
      col = "#770000", lwd = 3)

# fit prior to the deconvolution step
# does not fit the recorded data points appropriately
cps <- fit$leftEnd[8:9]
levels <- fit$value[7:9]
t <- seq(cps[1] - 0.0009, cps[2] + 0.0023, 1e-6)

# fit
lines(t, ifelse(t < cps[1], rep(levels[1], length(t)),
                ifelse(t < cps[2], rep(levels[2], length(t)),
                       rep(levels[3], length(t)))),
      col = "blue", lwd = 3)

# fit convolved with the filter
lines(t, levels[1] * (1 - filter$truncatedStepfun(t - cps[1])) +
        levels[2] * (filter$truncatedStepfun(t - cps[1]) -
                       filter$truncatedStepfun(t - cps[2])) +
        levels[3] * filter$truncatedStepfun(t - cps[2]),
      col = "darkblue", lwd = 3)


## zoom into a single jump
plot(time, gramA, pch = 16, col = "grey30", ylim = c(20, 50),
     xlim = c(9.6476, 9.6496), ylab = "Conductance in pS", xlab = "Time in s")

# relevant part of the idealisation
```

```
cp <- idealisation$leftEnd[2]
levels <- idealisation$value[1:2]
t <- seq(cp - 0.0009, cp + 0.0023, 1e-6)

# idealisation
lines(t, ifelse(t < cp, rep(levels[1], length(t)), rep(levels[2], length(t))),
      col = "#FF0000", lwd = 3)

# idealisation convolved with the filter
lines(t, levels[1] * (1 - filter$stepfun(t - cp)) + levels[2] * filter$stepfun(t - cp),
      col = "#770000", lwd = 3)

# idealisation with a wrong filter
# does not fit the recorded data points appropriately
wrongFilter <- lowpassFilter(type = "bessel",
                             param = list(pole = 6L, cutoff = 0.2),
                             sr = 1e4)

# Monte-Carlo simulation depend on the number of observations and on the filter
# hence a simulation is required again (if called for the first time)
# to save some time the number of iterations is reduced to r = 1e3
# hence the critical value is computed with less precision
# In general, r = 1e3 is enough for a first impression
# for a detailed analysis r = 1e4 is suggested
idealisationWrong <- jules(gramA, filter = wrongFilter, startTime = 9,
                           r = 1e3, messages = 100)

# relevant part of the idealisation
cp <- idealisationWrong$leftEnd[2]
levels <- idealisationWrong$value[1:2]
t <- seq(cp - 0.0012, cp + 0.0023, 1e-6)

# idealisation
lines(t, ifelse(t < cp, rep(levels[1], length(t)), rep(levels[2], length(t))),
      col = "blue", lwd = 3)

# idealisation convolved with the filter
lines(t, levels[1] * (1 - filter$stepfun(t - cp)) + levels[2] * filter$stepfun(t - cp),
      col = "darkblue", lwd = 3)

# simulation for a larger number of observations can be used (nq = 3e4)
# does not require a new simulation as the simulation from above will be used
# (if the previous call was executed first)
jules(gramA[1:2.99e4], filter = wrongFilter, startTime = 9,
      nq = 3e4, r = 1e3, messages = 100)

# simulation of type "vectorIncreased" for n1 observations can only be reused
# for n2 observations if as.integer(log2(n1)) == as.integer(log2(n2))
# no simulation is required, since a simulation of type "matrixIncreased"
# will be loaded from the fileSystem
# this call also saves a simulation of type "vectorIncreased" in the workspace
jules(gramA[1:1e4], filter = filter, startTime = 9,
      nq = 3e4, messages = 100, r = 1e3)
```

```
# here a new simulation is required
# (if no appropriate simulation is saved from a call outside of this file)
jules(gramA[1:1e3], filter = filter, startTime = 9,
      nq = 3e4, messages = 100, r = 1e3,
      options = list(load = list(workspace = c("vector", "vectorIncreased"))))

# the above calls saved and (attempted to) load Monte-Carlo simulations
# in the following call the simulations will neither be saved nor loaded
jules(gramA, filter = filter, startTime = 9, messages = 100, r = 1e3,
      options = list(load = list(), save = list()))

# only simulations of type "vector" and "vectorInceased" will only be in and
# loaded from the workspace, but no simulations of type "matrix" and
# "matrixIncreased" on the file system
jules(gramA, filter = filter, startTime = 9, messages = 100,
      options = list(load = list(workspace = c("vector", "vectorIncreased")),
                     save = list(workspace = c("vector", "vectorIncreased"))))

# explicit Monte-Carlo simulations, not recommended
stat <- stepR::monteCarloSimulation(n = length(gramA), , family = "mDependentPS",
                                    filter = filter, output = "maximum",
                                    r = 1e3, messages = 100)
jules(gramA, filter = filter, startTime = 9, stat = stat)

# with given standard deviation
sd <- stepR::sdrobnorm(gramA, lag = filter$len + 1)
identical(jules(gramA, filter = filter, startTime = 9, sd = sd), idealisation)

# with less regularisation of the correlation matrix
jules(gramA, filter = filter, startTime = 9, regularization = 0.5)

# with estimation of the level of long segments by the mean
# but requiring 30 observations for it
jules(gramA, filter = filter, startTime = 9,
      localEstimate = mean, thresholdLongSegment = 30)

# with one refinement step less, but with a larger grid
# progress of the deconvolution is reported
# potential warning for no deconvolution is suppressed
jules(gramA, filter = filter, startTime = 9,
      gridSize = c(1 / filter$sr, 1 / 10 / filter$sr),
      windowFactorRefinement = 2, report = TRUE,
      suppressWarningNoDeconvolution = TRUE)
```

`lowpassFilter` *Lowpass filtering*

Description

Create lowpass filter.

Usage

```
lowpassFilter(type = c("bessel"), param, sr = 1, len = NULL, shift = 0.5)
## S3 method for class 'lowpassFilter'
print(x, ...)
```

Arguments

type	a string specifying the type of the filter, currently only Bessel filters are supported
param	a list specifying the parameters of the filter depending on type. For "bessel" the entries pole and cutoff have to be specified and no other named entries are allowed. pole has to be a single integer giving the number of poles (order). cutoff has to be a single positive numeric not larger than 1 giving the normalized cutoff frequency, i.e. the cutoff frequency (in the temporal domain) of the filter divided by the sampling rate
sr	a single numeric giving the sampling rate
len	a single integer giving the filter length of the truncated and digitised filter, see *Value* for more details. By default (NULL) chosen such that the autocorrelation function is below 1e-3 at len / sr and all lager lags (len + i) / sr, with i a positive integer
shift	a single numeric between 0 and 1 giving a shift for the digitised filter, i.e. kernel and step are obtained at (0:len + shift) / sr from the corresponding functions
x	the object
...	for generic methods only

Value

An object of class lowpassFilter, i.e. a list that contains

"type", "param", "sr", "len" the corresponding arguments

"kernfun" the kernel function of the filter, obtained as the Laplace transform of the corresponding transfer function

"stepfun" the step-response of the filter, i.e. the antiderivative of the filter kernel

"acfun" the autocorrelation function, i.e. the convolution of the filter kernel with itself

"truncatedKernfun" the kernel function of the at len / sr truncated filter, i.e. kernfun truncated and rescaled such that the new kernel still integrates to 1

"truncatedStepfun" the step-response of the at len / sr truncated filter, i.e. the antiderivative of the kernel of the truncated filter

"truncatedAcfun" the autocorrelation function of the at len / sr truncated filter, i.e. the convolution of the kernel of the truncated filter with itself

"kern" the digitised filter kernel normalised to one, i.e. kernfun((0:len + shift) / sr) / sum(kernfun((0:len + shi

"step" the digitised step-response of the filter, i.e. stepfun((0:len + shift) / sr)

"acf" the discrete autocorrelation, i.e. acfun(0:len / sr)

Author(s)

This function is a modified and extended version of the dfilter function in the stepR package written by Thomas Hotz. New code is written by Florian Pein and Inder Tecuapetla-Gómez.

See Also

filter

Examples

```
# the filter used for the gramicidin A recordings given by gramA
filter <- lowpassFilter(type = "bessel", param = list(pole = 4L, cutoff = 1e3 / 1e4),
                        sr = 1e4)

# filter kernel, truncated version
plot(filter$kernfun, xlim = c(0, 20 / filter$sr))
t <- seq(0, 20 / filter$sr, 0.01 / filter$sr)
# truncated version looks very similar
lines(t, filter$truncatedKernfun(t), col = "red")

# filter$len (== 11) is chosen such that filter$acf < 1e-3 for it and all larger lags
plot(filter$acfun, xlim = c(0, 20 / filter$sr), ylim = c(-0.003, 0.003))
abline(h = 0.001, lty = "22")
abline(h = -0.001, lty = "22")

abline(v = (filter$len - 1L) / filter$sr, col = "grey")
abline(v = filter$len / filter$sr, col = "red")

## zoom into a single jump of the idealisation
## we suggest to do this for every new measurement setup once
## to control whether the correct filter is assumed
# idealisation by JULES (might take some time if not called somewhere before,
# please see its documentation for more details)
idealisation <- jules(gramA, filter = filter, startTime = 9, messages = 100)

## zoom into a single jump
plot(9 + seq(along = gramA) / filter$sr, gramA, pch = 16, col = "grey30",
     ylim = c(20, 50), xlim = c(9.6476, 9.6496), ylab = "Conductance in pS",
     xlab = "Time in s")

# relevant part of the idealisation
cp <- idealisation$leftEnd[2]
levels <- idealisation$value[1:2]
t <- seq(cp - 0.0009, cp + 0.0023, 1e-6)

# idealisation
lines(t, ifelse(t < cp, rep(levels[1], length(t)), rep(levels[2], length(t))),
      col = "#FF0000", lwd = 3)

# idealisation convolved with the filter
lines(t, levels[1] * (1 - filter$stepfun(t - cp)) + levels[2] * filter$stepfun(t - cp),
      col = "#770000", lwd = 3)
```

```
# idealisation with a wrong filter
# does not fit the recorded data points appropriately
wrongFilter <- lowpassFilter(type = "bessel",
                             param = list(pole = 6L, cutoff = 0.2),
                             sr = 1e4)
# the needed Monte-Carlo simulation depends on the number of observations and the filter
# hence a new simulation is required (if called for the first time)
idealisationWrong <- jules(gramA, filter = wrongFilter, startTime = 9, messages = 100)

# relevant part of the idealisation
cp <- idealisationWrong$leftEnd[2]
levels <- idealisationWrong$value[1:2]
t <- seq(cp - 0.0012, cp + 0.0023, 1e-6)

# idealisation
lines(t, ifelse(t < cp, rep(levels[1], length(t)), rep(levels[2], length(t))),
      col = "blue", lwd = 3)

# idealisation convolved with the filter
lines(t, levels[1] * (1 - filter$stepfun(t - cp)) + levels[2] * filter$stepfun(t - cp),
      col = "darkblue", lwd = 3)


# filter with sr == 1
filter <- lowpassFilter(type = "bessel", param = list(pole = 4L, cutoff = 1e3 / 1e4))

# filter kernel and its truncated version
plot(filter$kernfun, xlim = c(0, 20 / filter$sr))
t <- seq(0, 20 / filter$sr, 0.01 / filter$sr)
# truncated version looks very similar
lines(t, filter$truncatedKernfun(t), col = "red")
# digitised filter
points((0:filter$len + 0.5) / filter$sr, filter$kern, col = "red", pch = 16)

# without a shift
filter <- lowpassFilter(type = "bessel", param = list(pole = 4L, cutoff = 1e3 / 1e4),
                        shift = 0)
# filter$kern starts with zero
points(0:filter$len / filter$sr, filter$kern, col = "blue", pch = 16)

# much shorter filter
filter <- lowpassFilter(type = "bessel", param = list(pole = 4L, cutoff = 1e3 / 1e4),
                        len = 4L)
points((0:filter$len + 0.5) / filter$sr, filter$kern, col = "darkgreen", pch = 16)
```

`stepDetection` *Detection of steps / jumps by a multiresolution criterion*

Description

Implements the detection step of JULES (*Pein et al.*, 2017, section III B) which consists of a fit by a multiresolution criterion computed by a dynamic program and a postfilter step that removes incremental steps. This initial fit (reconstruction) can then be refined by local deconvolution implemented in `deconvolveLocally` to obtain JULES, also implemented in `jules`.
If `q == NULL` a Monte-Carlo simulation is required for computing the critical value. Since a Monte-Carlo simulation lasts potentially much longer (up to several hours or days if the number of observations is in the millions) than the main calculations, this package saves them by default in the workspace and on the file system such that a second call that require the same Monte-Carlo simulation will be much faster. For more details, in particular to which arguments the Monte-Carlo simulations are specific, see Section *Storing of Monte-Carlo simulations* below. Progress of a Monte-Carlo simulation can be reported by the argument `messages` and the saving can be controlled by the argument `option`, both can be specified in `...` and are explained in `getCritVal`.

Usage

```
stepDetection(data, filter, q = NULL, alpha = 0.05, sd = NULL, startTime = 0,
              output = c("onlyFit", "everything"), ...)
```

Arguments

`data`	a numeric vector containing the recorded data points
`filter`	an object of class `lowpassFilter` giving the used analogue lowpass filter
`q`	a single numeric giving the critical value q in (*Pein et al.*, 17, (7)), by default chosen automatically by `getCritVal`
`alpha`	a probability, i.e. a single numeric between 0 and 1, giving the significance level to compute the critical value `q` (if `q == NULL`), see `getCritVal`. Its choice is a trade-off between data fit and parsimony of the estimator. In other words, this argument balances the risks of missing changes and detecting additional artefacts. For more details on this choice see (Frick et al., 2014, section 4) and (Pein et al., 2016, section 3.4)
`sd`	a single positive numeric giving the standard deviation (noise level) σ_0 of the data points before filtering, by default (NULL) estimated by `sdrobnorm` with `lag = filter$len + 1L`
`startTime`	a single numeric giving the time at which recording (sampling) of `data` started, sampling time points will be assumed to be `startTime + seq(along = data) / filter$sr`
`output`	a string specifing the return type, see *Value*
`...`	additional parameters to be passed to `getCritVal`. `getCritVal` will be called automatically (if `q == NULL`), the number of data points `n = length(data)` will be set and `alpha` and `filter` will be passed. For these parameter no user interaction is required and possible, all other parameters of `getCritVal` can be passed additionally

Value

The reconstruction (fit) obtained by the detection step of JULES. If `output == "onlyFit"` an object object of class `stepblock` containing the fit. If `output == "everything"` a `list` containing the

entries `fit` with the fit, `stepfit` with the fit before postfiltering, `q` with the given / computed critical value, `filter` with the given filter and `sd` with the given / estimated standard deviation.

Storing of Monte-Carlo simulations

If `q == NULL` a Monte-Carlo simulation is required to compute the critical value. Since a Monte-Carlo simulation lasts potentially much longer (up to several hours or days if the number of observations is in the millions) than the main calculations, multiple possibilities for saving and loading the simulations are offered. Progress of a simulation can be reported by the argument `messages` which can be specified in `...` and is explained in the documentation of `getCritVal`. Each Monte-Carlo simulation is specific to the number of observations and the used filter. But note that also Monte-Carlo simulations for a (slightly) larger number of observations n_q, given in the argument `nq` in `...` and explained in the documentation of `getCritVal`, can be used, which avoids extensive resimulations for only a little bit varying number of observations, but results in a (small) loss of power. However, simulations of type `"vectorIncreased"`, i.e. objects of class `"MCSimulationMaximum"` with `nq` observations, have to be resimulated if `as.integer(log2(n1)) != as.integer(log2(n2))` when the saved simulation was computed with `n == n1` and the simulation now is required for `n == n2` and `nq >= n1` and `nq >= n2`. Simulations can either be saved in the workspace in the variable `critValStepRTab` or persistently on the file system for which the package `R.cache` is used. Moreover, storing in and loading from variables and RDS files is supported. The simulation, saving and loading can be controlled by the argument `option` which can be specified in `...` and is explained in the documentation of `getCritVal`. By default simulations will be saved in the workspace and on the file system. For more details and for how simulation can be removed see Section *Simulating, saving and loading of Monte-Carlo simulations* in `getCritVal`.

References

Pein, F., Tecuapetla-Gómez, I., Schütte, O., Steinem, C., Munk, A. (2017) Fully-automatic multiresolution idealization for filtered ion channel recordings: flickering event detection. *arXiv*:1706.03671.

See Also

`jules`, `getCritVal`, `lowpassFilter`, `deconvolveLocally`

Examples

```
## fit of the gramicidin A recordings given by gramA
# the used filter
filter <- lowpassFilter(type = "bessel", param = list(pole = 4L, cutoff = 1e3 / 1e4),
                        sr = 1e4)

# this call requires a Monte-Carlo simulation
# and therefore might last a few minutes,
# progress of the Monte-Carlo simulation is reported
fit <- stepDetection(gramA, filter = filter, startTime = 9, messages = 100)

# this second call should be much faster
# as the previous Monte-Carlo simulation will be loaded
stepDetection(gramA, filter = filter, startTime = 9)

# much larger significance level alpha for a larger detection power,
```

```
# but also with the risk of detecting additional artefacts
# in this example much more changes are detected,
# most of them are probably artefacts, but for instance the event at 11.3699
# might be an additional small event that was missed before
stepDetection(gramA, filter = filter, alpha = 0.9, startTime = 9)

# getCritVal was called in stepDetection, can be called explicitly
# for instance outside of a for loop to save computation time
q <- getCritVal(length(gramA), filter = filter)
identical(stepDetection(gramA, q = q, filter = filter, startTime = 9), fit)

# more detailed output
every <- stepDetection(gramA, filter = filter, startTime = 9, output = "every")
identical(every$fit, fit)
identical(every$q, q)
identical(every$sd, stepR::sdrobnorm(gramA, lag = filter$len + 1L))
identical(every$filter, every$filter)

# for this data set no incremental changes occur
identical(every$stepfit, every$stepfit)

## zoom into a single event
time <- 9 + seq(along = gramA) / filter$sr # time points
plot(time, gramA, pch = 16, col = "grey30", ylim = c(20, 50),
     xlim = c(10.40835, 10.4103), ylab = "Conductance in pS", xlab = "Time in s")

# fit is a piecewise constant approximation of the observations
# hence its convolution does not fit the recorded data points appropriately
# for a fit of the observations a deconvolution is required
# either by calling deconvolveLocally additionally or better immediately jules
cps <- fit$leftEnd[8:9]
levels <- fit$value[7:9]
t <- seq(cps[1] - 0.0009, cps[2] + 0.0023, 1e-6)

# fit
lines(t, ifelse(t < cps[1], rep(levels[1], length(t)),
                ifelse(t < cps[2], rep(levels[2], length(t)),
                       rep(levels[3], length(t)))),
      col = "blue", lwd = 3)

# fit convolved with the filter
lines(t, levels[1] * (1 - filter$truncatedStepfun(t - cps[1])) +
        levels[2] * (filter$truncatedStepfun(t - cps[1]) -
                       filter$truncatedStepfun(t - cps[2])) +
        levels[3] * filter$truncatedStepfun(t - cps[2]),
      col = "darkblue", lwd = 3)


# fit with a wrong filter
wrongFilter <- lowpassFilter(type = "bessel",
                             param = list(pole = 6L, cutoff = 0.2),
                             sr = 1e4)
```

```
# Monte-Carlo simulation depend on the number of observations and on the filter
# hence a simulation is required again (if called for the first time)
# to save some time the number of iterations is reduced to r = 1e3
# hence the critical value is computed with less precision
# In general, r = 1e3 is enough for a first impression
# for a detailed analysis r = 1e4 is suggested
stepDetection(gramA, filter = filter, startTime = 9, messages = 100L, r = 1e3L)

# simulation for a larger number of observations can be used (nq = 3e4)
# does not require a new simulation as the simulation from above will be used
# (if the previous call was executed first)
stepDetection(gramA, filter = filter, startTime = 9,
              messages = 100L, r = 1e3L, nq = 3e4L)

# simulation of type "vectorIncreased" for n1 observations can only be reused
# for n2 observations if as.integer(log2(n1)) == as.integer(log2(n2))
# no simulation is required, since a simulation of type "matrixIncreased"
# will be loaded from the fileSystem
# this call also saves a simulation of type "vectorIncreased" in the workspace
stepDetection(gramA[1:1e4], filter = filter, startTime = 9,
      nq = 3e4, messages = 100, r = 1e3)
# here a new simulation is required
# (if no appropriate simulation is saved from a call outside of this file)
stepDetection(gramA[1:1e3], filter = filter, startTime = 9,
      nq = 3e4, messages = 100, r = 1e3,
      options = list(load = list(workspace = c("vector", "vectorIncreased"))))

# the above calls saved and (attempted to) load Monte-Carlo simulations
# in the following call the simulations will neither be saved nor loaded
stepDetection(gramA, filter = filter, startTime = 9, messages = 100L, r = 1e3L,
              options = list(load = list(), save = list()))

# only simulations of type "vector" and "vectorInceased" will only be in and
# loaded from the workspace, but no simulations of type "matrix" and
# "matrixIncreased" on the file system
stepDetection(gramA, filter = filter, startTime = 9, messages = 100L, r = 1e3L,
              options = list(load = list(workspace = c("vector", "vectorIncreased")),
                             save = list(workspace = c("vector", "vectorIncreased"))))

# explicit Monte-Carlo simulations, not recommended
stat <- stepR::monteCarloSimulation(n = length(gramA), , family = "mDependentPS",
                                    filter = filter, output = "maximum",
                                    r = 1e3, messages = 100)
stepDetection(gramA, filter = filter, startTime = 9, stat = stat)

# with given standard deviation
sd <- stepR::sdrobnorm(gramA, lag = filter$len + 1)
identical(stepDetection(gramA, filter = filter, startTime = 9, sd = sd), fit)
```

Index