Package 'clampSeg'

October 12, 2022

Title Idealisation of Patch Clamp Recordings

Version 1.1-1

Depends R (>= 3.3.0)

Imports stepR (>= 2.1.0), lowpassFilter, stats, methods

Suggests testthat, R.cache (>= 0.10.0), R.rsp

Description
Implements the model-free multiscale idealisation approaches: Jump-Segmentation by MUltiResolution Filter (JSMURF) <doi:10.1109/TNB.2013.2284063>, JUmp Local dEconvolution Segmentation filter (JULES) <doi:10.1109/TNB.2018.2845126> and Heterogeneous Idealization by Local testing and DEconvolution (HILDE) <arXiv:2008.02658>. Further details on how to use them are given in the accompanying vignette.

VignetteBuilder R.rsp

License GPL-3

Encoding UTF-8

LazyData true

NeedsCompilation no

Author Pein Florian [aut, cre],
Timo Aspelmeier [ctb]

Maintainer Pein Florian <f.pein@lancaster.ac.uk>

Repository CRAN

Date/Publication 2022-01-27 23:10:06 UTC

R topics documented:

`clampSeg-package` *Idealisation of Patch Clamp Recordings*

Description

Implements the model-free multiscale idealisation approaches: **J**ump-**S**egmentation by **MU**ltiResolution **F**ilter (JSMURF) (*Hotz et al.*, 2013), **JU**mp **L**ocal d**E**convolution **S**egmentation (JULES) filter (*Pein et al.*, 2018) and **H**eterogeneous **I**dealization by **L**ocal testing and **DE**convolution (HILDE) (*Pein et al.*, 2020). These methods combine multiscale testing with deconvolution to idealise patch clamp recordings. They allow to deal with subconductance states and flickering. Further details are given in the accompanying vignette.

Details

The main functions are `jsmurf`, `jules` and `hilde` which implement JSMURF, JULES and HILDE, respectively. JSMURF is the most simplest and fastest approach. If short events (flickering) occurs, JULES or HILDE should be used instead. All three methods can assume homogeneous noise, but JSMURF and HILDE have options to allow for heterogeneous noise. Further details on when which method is suitable and on how to use them are given in Section II in the vignette.
The signal underlying the data in a patch clamp recording is assumed to be a step (piecewise constant) function, e.g. constant conductance levels are assumed. The signal is perturbed by (Gaussian) white noise and convolved with a lowpass filter, resulting in a smooth signal perturbed by correlated noise with known correlation structure. The white noise is scaled by a constant if homogeneous noise is assumed and by an unknown piecewise constant function if heterogeneous noise is assumed. Heterogeneous noise can for instance by caused by open channel noise. The recorded data points are modelled as sampled (digitised) recordings of this process. For more details on this model see Section III in the vignette. 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. `createLocalList` allows pre-calculations for `improveSmallScales` and `hilde`. Doing so reduces the running time if `improveSmallScales` and `hilde` are called more than once. The multiscale step in all three approaches requires critical values. Be default, those values are automatically computed. Alternatively, they can be computed separately by `getCritVal`. Their computation relies on Monte-Carlo simulations.
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. Hence, multiple possibilities for saving and loading the simulations are offered and used by default. 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

simulations 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. The refinement steps (second and third step) of HILDE, which improve a fit on small temporal scales by testing for additional events and applying local deconvolution, can be accessed by the function improveSmallScales.

References

Pein, F., Bartsch, A., Steinem, C., Munk, A. (2020) Heterogeneous Idealization of Ion Channel Recordings - Open Channel Noise. *arXiv*:2008.02658.

Pein, F., Tecuapetla-Gómez, I., Schütte, O., Steinem, C., Munk, A. (2018) Fully-automatic multiresolution idealization for filtered ion channel recordings: flickering event detection. *IEEE Transactions on NanoBioscience* **17**(3), 300–320.

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.

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

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.

See Also

jsmurf, jules, hilde, getCritVal, lowpassFilter, createLocalList, gramA, deconvolveLocally, stepDetection, improveSmallScales

Examples

```
## idealisation of the gramicidin A recording 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., 2018, Figure 1 lower panel)
plot(time, gramA, pch = ".", col = "grey30", ylim = c(20, 50),
     ylab = "Conductance in pS", xlab = "Time in s")


# idealisations require Monte-Carlo simulations
# and therefore might last a few minutes,
# progress of the Monte-Carlo simulations is reported
# JSMURF assuming homogeneous noise
fit1 <- jsmurf(gramA, filter = filter, family = "jsmurfPS",
               startTime = 9, messages = 100)
# JSMURF allowing heterogeneous noise
fit2 <- jsmurf(gramA, filter = filter, family = "hjsmurf",
               startTime = 9, messages = 100)
```

```
# JULES assuming homogeneous noise
fit3 <- jules(gramA, filter = filter, startTime = 9, messages = 100)
# HILDE assuming homogeneous noise
fit4 <- hilde(gramA, filter = filter, family = "jsmurfPS", method = "LR",
              startTime = 9, messages = 10)
# HILDE allowing heterogeneous noise
fit5 <- hilde(gramA, filter = filter, family = "hjsmurf", method = "2Param",
              startTime = 9, messages = 10, r = 100)
# r = 100 is used to reduce its run time,
# this is okay for illustration purposes, but for precise results
# a larger number of Monte-Carlo simulations is recommend

# gramA contains short peaks and the noise is homogeneous
# hence jules seems to be most appropriate
idealisation <- fit3

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

## in the following we use jules to illustrate various points,
## similar points are valid for jsmurf and hilde, too,
## see also their individual documentation for more details
# any second call should be much faster
# as the previous Monte-Carlo simulation will be loaded
jules(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 11.36972
# 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 run 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., 2018, 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")

# idealisation
lines(idealisation, col = "red", lwd = 3)

# idealisation convolved with the filter
ind <- seq(10.408, 10.411, 1e-6)
convolvedSignal <- lowpassFilter::getConvolution(ind, idealisation, filter)
lines(ind, convolvedSignal, col = "blue", lwd = 3)

# fit prior to the deconvolution step
# does not fit the recorded data points appropriately
# fit
lines(fit, col = "orange", lwd = 3)

# fit convolved with the filter
ind <- seq(10.408, 10.411, 1e-6)
convolvedSignal <- lowpassFilter::getConvolution(ind, fit, filter)
lines(ind, convolvedSignal, col = "darkgreen", lwd = 3)


## 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")

# idealisation
lines(idealisation, col = "red", lwd = 3)

# idealisation convolved with the filter
ind <- seq(9.647, 9.65, 1e-6)
convolvedSignal <- lowpassFilter::getConvolution(ind, idealisation, filter)
lines(ind, convolvedSignal, col = "blue", 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)

# idealisation
lines(idealisationWrong, col = "orange", lwd = 3)

# idealisation convolved with the filter
```

```
ind <- seq(9.647, 9.65, 1e-6)
convolvedSignal <- lowpassFilter::getConvolution(ind, idealisationWrong, filter)
lines(ind, convolvedSignal, col = "darkgreen", 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 = filter, 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 be saved 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)
```

`createLocalList` *Precomputes quantities for hilde and improveSmallScales*

Description

Allows to precompute quantities that can be passed to `hilde` and `improveSmallScales`, all arguments have to coincide with the corresponding ones in those function calls. Doing so saves run time if those functions are called multiple times (with the same arguments) as then the quantities have to be computed only once.

Usage

```
createLocalList(filter, method = c("2Param", "LR"),
                lengths = if (method == "LR") 1:20 else 1:65)
```

Arguments

filter	an object of class lowpassFilter giving the used analogue lowpass filter
method	the testing method for short events, "2Param" allows for heterogeneous noise, "LR" assumes homogeneous noise
lengths	a vector of integers giving the lengths on which tests will be performed to detect short events, should be chosen such that events on larger scales are already detected by the previous jsmurf step

Value

An object of class "localList", contains computed quantities used by hilde and improveSmallScales

See Also

hilde, improveSmallScales, lowpassFilter

Examples

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


localList <- createLocalList(filter = filter, method = "LR")

# idealisation by HILDE using localList
# this call requires a Monte-Carlo simulation
# and therefore might last a few minutes,
# progress of the Monte-Carlo simulation is reported
idealisation <- hilde(gramA, filter = filter, family = "jsmurfPS", method = "LR",
                      startTime = 9, messages = 10, localList = localList)

identical(hilde(gramA, filter = filter, family = "jsmurfPS", method = "LR",
                startTime = 9), idealisation)

# HILDE allowing heterogeneous noise, for only few lengths
localList <- createLocalList(filter = filter, method = "2Param", lengths = c(3, 10, 25))

# localList has to be computed with the same filter, method and lengths
hilde(gramA, filter = filter, family = "hjsmurf", method = "2Param",
      startTime = 9, messages = 10, r = 100,
      lengths = c(3, 10, 25), localList = localList)
# r = 100 is used to reduce its run time,
# this is okay for illustration purposes, but for precise results
# a larger number of Monte-Carlo simulations is recommend
```

`deconvolveLocally` *Local deconvolution*

Description

Implements the estimation step of JULES (*Pein et al.*, 2018, 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, 1 / 10, 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 determining 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 refinement of the grid, see *details*. If a single numeric is given its value is used in all refinement steps
`output`	a string specifying 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.*, 2018, (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. (2018) Fully-automatic multiresolution idealization for filtered ion channel recordings: flickering event detection. *IEEE Transactions on NanoBioscience* **17**(3), 300–320.

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)

## zoom into a single event, (Pein et al., 2018, Figure 2 lower left panel)
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")

# deconvolution
```

```
lines(deconvolution, col = "red", lwd = 3)

# deconvolution convolved with the filter
ind <- seq(10.408, 10.411, 1e-6)
convolvedSignal <- lowpassFilter::getConvolution(ind, deconvolution, filter)
lines(ind, convolvedSignal, col = "blue", lwd = 3)

# for comparison, fit prior to the deconvolution step
# does not fit the recorded data points appropriately
# fit
lines(fit, col = "orange", lwd = 3)

# fit convolved with the filter
ind <- seq(10.408, 10.411, 1e-6)
convolvedSignal <- lowpassFilter::getConvolution(ind, fit, filter)
lines(ind, convolvedSignal, col = "darkgreen", lwd = 3)


## 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")

# deconvolution
lines(deconvolution, col = "red", lwd = 3)

# deconvolution convolved with the filter
ind <- seq(9.647, 9.65, 1e-6)
convolvedSignal <- lowpassFilter::getConvolution(ind, deconvolution, filter)
lines(ind, convolvedSignal, col = "blue", 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)

# deconvolution
lines(deconvolutionWrong, col = "orange", lwd = 3)

# ideconvolution convolved with the filter
ind <- seq(9.647, 9.65, 1e-6)
convolvedSignal <- lowpassFilter::getConvolution(ind, deconvolutionWrong, filter)
lines(ind, convolvedSignal, col = "darkgreen", 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 values*

Description

Computes critical values for the functions `jsmurf`, `jules`, `hilde`, `stepDetection` and `improveSmallScales`. `getCritVal` is usually automatically called, but can be called explicitly, for instance outside of a for loop to save run time. Computation requires 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, 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, filter, family = c("jules", "jsmurf", "jsmurfPS", "jsmurfLR",
                           "hjsmurf", "hjsmurfSPS", "hjsmurfLR", "LR", "2Param"),
           alpha = 0.05, r = NULL, nq = n, options = NULL,
           stat = NULL, messages = NULL, ...)
```

Arguments

`n`	a positive integer giving the number of observations
`filter`	an object of class `lowpassFilter` giving the used analogue lowpass filter
`family`	the parametric family for which critical values should be computed, select `"jules"` for a critical value that will be passed to `jules` or `stepDetection`, the families `"jsmurf"`, `"jsmurfPS"`, `"jsmurfLR"`, `"hjsmurf"`, `"hjsmurfSPS"` and `"hjsmurfLR"` according to the argument `family` in `jsmurf` and `hilde`, and `"LR"` and `"2Param"` according to the argument `method` in `hilde` and `improveSmallScales`
`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 conductance changes and

detecting additional artefacts. For more details on this choice see the accompanying vignette or (Frick et al., 2014, Section 4) and (Pein et al., 2017, Section 3.4)

`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, a larger number improves accuracy but simulations last longer; by default `1e4` is used except for families `"LR"` and `"2Param"`, where `1e3` is used since their simulations are rather slow

`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 automatically 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

`...` additional arguments of the parametric families `"LR"` and `"2Param"`, i.e. `thresholdLongSegment`, `localValue`, `localVar`, `regularization`, `suppressWarningNoDeconvolution`, `localList`, please see their documentation in `improveSmallScales` to understand their meaning, parameters have to coincide in the call of `getCritVal` and `hilde` or `improveSmallScales`, argument `localVar` is only allowed for family `"2Param"`, for other families additional arguments are ignored with a `warning`

Value

For families `"jules"`, `"jsmurf"`, `"jsmurfPS"`, `"jsmurfLR"` a single numeric giving the critical value and for families `"hjsmurf"`, `"hjsmurfSPS"`, `"hjsmurfLR"`, `"LR"` and `"2Param"` a numeric vector giving scale dependent critical values. Additionally, an `attribute` `n` with a single integer giving the number of data points for which the values are computed.

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. By default, simulations are stored and loaded with suitable default values and no user choices are required. If desired, 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 parametric `family`, their parameters in case of families `"LR"` or `"2Param"`, 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

Computation of scale depend critical values, i.e. calucatlions for the families "hjsmurf", "hjsmurfSPS", "hjsmurfLR", "LR" and "2Param" require an object of class "MCSimulationVector". Otherwise, 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" for families "jules", "jsmurf", "jsmurfPS", "jsmurfLR" and "matrix" for families "hjsmurf", "hjsmurfSPS", "hjsmurfLR", "LR" and "2Param" 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. For families "jules", "jsmurf", "jsmurfPS", "jsmurfLR" one should also consider the following: 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., Bartsch, A., Steinem, C., Munk, A. (2020) Heterogeneous Idealization of Ion Channel Recordings - Open Channel Noise. *arXiv*:2008.02658.

Pein, F., Tecuapetla-Gómez, I., Schütte, O., Steinem, C., Munk, A. (2018) Fully-automatic multiresolution idealization for filtered ion channel recordings: flickering event detection. *IEEE Transactions on NanoBioscience* **17**(3), 300–320.

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., 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

`jsmurf`, `jules`, `hilde`, `lowpassFilter`, `stepDetection`, `improveSmallScales`

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)

# critical value for jsmurf,
# Monte-Carlo simulations are specific to the parametric family,
# hence a new Monte-Carlo simulation is required
getCritVal(length(gramA), family = "jsmurfPS", filter = filter, messages = 100)

# scale dependent critical value for jsmurf allowing for heterogeneous noise,
# return value is a vector
getCritVal(length(gramA), family = "hjsmurf", filter = filter, messages = 100)

# scale dependent critical value for "LR" as used by improveSmallScales and hilde,
# return value is a vector
getCritVal(length(gramA), family = "LR", filter = filter, messages = 100)

# families "LR" and "2Param" allows to specify additional parameters in ...
# Monte-Carlo simulations are also specific to those values
getCritVal(length(gramA), family = "LR", filter = filter, messages = 100,
           localValue = mean, thresholdLongSegment = 15L)

# 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 seconds 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 conductance 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.*, 2018, Section V A) and a plot in the examples or in (*Pein et al.*, 2018, 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. (2018) Fully-automatic multiresolution idealization for filtered ion channel recordings: flickering event detection. *IEEE Transactions on NanoBioscience* **17**(3), 300–320.

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., 2018, Figure 1 lower panel)
plot(time, gramA, pch = ".", col = "grey30", ylim = c(20, 50),
     ylab = "Conductance in pS", xlab = "Time in s")
```

`hilde` *HILDE*

Description

Implements the **H**eterogeneous **I**dealization by **L**ocal testing and **DE**convolution (HILDE) filter (*Pein et al.*, 2020). This non-parametric (model-free) segmentation method combines statistical multiresolution techniques with local deconvolution for idealising patch clamp (ion channel) recordings. It is able to idealize short events (flickering) and allows for heterogeneous noise, but is rather slow. Hence, we recommend to use `jsmurf` or `jules` instead if they are suitable as well. Please see the arguments `family` and `method` as well as the *examples* for how to access the function correctly depending on whether homogeneous is assumed or heterogeneous noise is allowed. `hilde` is a combination of `jsmurf` (with `locationCorrection == "none"`) and `improveSmallScales`. Further details about how to decide whether the noise is homogeneous or heterogeneous and whether events are short, and hence which method is suitable, are given in the accompanying vignette.
If `q1 == NULL` or `q2 == NULL` a Monte-Carlo simulation is required for computing the critical values. 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

```
hilde(data, filter, family = c("hjsmurf", "hjsmurfSPS", "hjsmurfLR",
                               "jsmurf", "jsmurfPS", "jsmurfLR"),
    method = c("2Param", "LR"), q1 = NULL, alpha1 = 0.01, q2 = NULL, alpha2 = 0.04,
      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
family	the parametric family used in the jsmurf step; "jsmurf", "jsmurfPS" and "jsmurfLR" assume homogeneous noise and "hjsmurf", "hjsmurfSPS" and "hjsmurfLR" allow for heterogeneous noise. By default, we recommend to use "jsmurfPS" when homogeneous noise is assumed and "hjsmurf" when heterogeneous noise is allowed, see *examples*. "jsmurf" is the standard statistic from (*Hotz et al.*, 2013), "jsmurfPS" is a slightly more powerful partial sum statistic, "jsmurfLR" is a likelihood-ratio statistic, which is even more powerful but slow. "hjsmurf" is the standard statistic for heterogeneous noise which estimates the variance locally, "hjsmurfSPS" is a studentized partial sum statistic and "hjsmurfLR" is a likelihood ratio statistic, which is more powerful, but very slow
method	the testing method for short events in the improveSmallScales step; "2Param" allows for heterogeneous noise, "LR" assumes homogeneous noise
q1	will be passed to the argument q in jsmurf; by default chosen automatically by getCritVal, for families "jsmurf", "jsmurfPS" and "jsmurfLR" a single numeric, for families "hjsmurf", "hjsmurfSPS" and "hjsmurfLR" a numeric vector giving scale dependent critical values
alpha1	will be passed to the argument alpha in jsmurf; a probability, i.e. a single numeric between 0 and 1, giving the significance level to compute q1 (if q1 == 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 conductance changes and detecting additional artefacts
q2	will be passed to the argument q in improveSmallScales; a numeric vector of the same length as lengths giving critical value for the tests for short events, by default chosen automatically by getCritVal
alpha2	will be passed to the argument alpha in improveSmallScales; a probability, i.e. a single numeric between 0 and 1, giving the significance level to compute the critical value (if q2 == NULL), see getCritVal. Its choice balances the risks of missing short events and detecting additional artefacts
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. For families "hjsmurf", "hjsmurfSPS" and "hjsmurfLR" this argument is ignored with a warning

`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 specifying the return type, see *Value*

`...` additional parameters to be passed to `getCritVal` or `improveSmallScales`:

1. `getCritVal` will be called automatically (if `q1 == NULL` or `q2 == NULL`), the number of data points `n = length(data)` will be set, the argument `family` will be assigned 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. Note that the same arguments will be passed twice if `q1` and `q2` have to be computed. If this is not suitable, `getCritVal` can be called instead
2. `improveSmallScales` will be called automatically, the by `jsmurf` computed fit will be passed to `fit` and `data`, `filter`, `method`, `q = q2`, `alpha = alpha2`, `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 HILDE. 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 by `jsmurf`, `q1` and `q2` with the given / computed critical values, `filter` with the given filter and for families `"jsmurf"`, `"jsmurfPS"` and `"jsmurfLR"` `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 by `jsmurf`, `q1` and `q2` with the given / computed critical values, `filter` with the given filter and for families `"jsmurf"`, `"jsmurfPS"` and `"jsmurfLR"` `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 `improveSmallScales`.

Storing of Monte-Carlo simulations

If `q1 == NULL` or `q2 == NULL` a Monte-Carlo simulation is required to compute the critical values. 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 parametric `family` / specified testing `method`, the number of observations and the used filter. Simulations related to computing `q2` are also specific to the arguments `thresholdLongSegment`, `localValue` and `localVar`. Currently, storing such a Monte-Carlo simulation is only possible for their default values. 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" (only possible for q1 and families "jsmurf", "jsmurfPS" and "jsmurfLR") or "matrixIncreased", i.e. objects of classes "MCSimulationMaximum" and "MCSimulationVector" 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., Bartsch, A., Steinem, C., Munk, A. (2020) Heterogeneous Idealization of Ion Channel Recordings - Open Channel Noise. *arXiv*:2008.02658.

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.

See Also

getCritVal, jsmurf, jules, lowpassFilter, improveSmallScales, createLocalList

Examples

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

# idealisation by HILDE assuming homogeneous noise
# this call requires a Monte-Carlo simulation
# and therefore might last a few minutes,
# progress of the Monte-Carlo simulation is reported
idealisation <- hilde(gramA, filter = filter, family = "jsmurfPS", method = "LR",
                      startTime = 9, messages = 10)

# any second call should be much faster
# as the previous Monte-Carlo simulation will be loaded
hilde(gramA, filter = filter, family = "jsmurfPS", method = "LR", startTime = 9)

# HILDE allowing heterogeneous noise
hilde(gramA, filter = filter, family = "hjsmurf", method = "2Param",
      startTime = 9, messages = 10, r = 100)
# r = 100 is used to reduce its run time,
# this is okay for illustration purposes, but for precise results
# a larger number of Monte-Carlo simulations is recommend

# much larger significance level alpha1 for a larger detection power
# in the refinement step on small temporal scales,
```

```
# but also with the risk of detecting additional artefacts
hilde(gramA, filter = filter, family = "jsmurfPS", method = "LR",
      alpha1 = 0.9, alpha2 = 0.9, startTime = 9)

# getCritVal was called in hilde, can be called explicitly
# for instance outside of a for loop to save run time
q2 <- getCritVal(length(gramA), filter = filter, family = "LR")
identical(hilde(gramA, filter = filter, family = "jsmurfPS",
                method = "LR", startTime = 9, q2 = q2), idealisation)

# both steps of HILDE can be called separately
fit <- jsmurf(gramA, filter = filter, family = "jsmurfPS", alpha = 0.01,
              startTime = 9, locationCorrection = "none")
deconvolution <- improveSmallScales(fit, data = gramA, method = "LR", filter = filter,
                                    startTime = 9, messages = 100)
attr(deconvolution, "q") <- NULL
identical(deconvolution, idealisation)

# more detailed output
each <- hilde(gramA, filter = filter, family = "jsmurfPS", method = "LR",
              startTime = 9, output = "each")

every <- hilde(gramA, filter = filter, family = "jsmurfPS", method = "LR",
               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
## similar to (Pein et al., 2018, 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")

# idealisation
lines(idealisation, col = "red", lwd = 3)

# idealisation convolved with the filter
ind <- seq(10.408, 10.411, 1e-6)
convolvedSignal <- lowpassFilter::getConvolution(ind, idealisation, filter)
lines(ind, convolvedSignal, col = "blue", lwd = 3)

# for comparison, fit prior to the improvement step
# does not contain the event and hence fits the recorded data points badly
# fit
lines(fit, col = "orange", lwd = 3)
```

```
# fit convolved with the filter
ind <- seq(10.408, 10.411, 1e-6)
convolvedSignal <- lowpassFilter::getConvolution(ind, fit, filter)
lines(ind, convolvedSignal, col = "darkgreen", lwd = 3)


## 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")

# idealisation
lines(idealisation, col = "red", lwd = 3)

# idealisation convolved with the filter
ind <- seq(9.647, 9.65, 1e-6)
convolvedSignal <- lowpassFilter::getConvolution(ind, idealisation, filter)
lines(ind, convolvedSignal, col = "blue", 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 <- hilde(gramA, filter = wrongFilter, family = "jsmurfPS",
                           method = "LR", startTime = 9, messages = 10)

# idealisation
lines(idealisationWrong, col = "orange", lwd = 3)

# idealisation convolved with the filter
ind <- seq(9.647, 9.65, 1e-6)
convolvedSignal <- lowpassFilter::getConvolution(ind, idealisationWrong, filter)
lines(ind, convolvedSignal, col = "darkgreen", 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)
hilde(gramA[1:2.99e4], filter = filter, family = "jsmurfPS", method = "LR",
      startTime = 9, nq = 3e4)
# note that arguments to compute critical values are used to compute q1 and q2
# if this is not wanted, getCritVal can be called separately
q1 <- getCritVal(length(gramA[1:2.99e4]), filter = filter, family = "jsmurfPS",
                 messages = 100, r = 1e3)
hilde(gramA[1:2.99e4], filter = filter, family = "jsmurfPS", method = "LR",
      q1 = q1, startTime = 9, nq = 3e4) # nq = 3e4 is only used to compute q2

# 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
hilde(gramA[1:1e4], filter = filter, family = "jsmurfPS", method = "LR",
      startTime = 9, nq = 3e4)

# the above calls saved and (attempted to) load Monte-Carlo simulations
# in the following call the simulations will neither be saved nor loaded
# Monte-Carlo simulations are required for q1 and for q2
hilde(gramA, filter = filter, family = "jsmurfPS", method = "LR",
      startTime = 9, messages = 10, r = 100,
      options = list(load = list(), save = list()))

# with given standard deviation
sd <- stepR::sdrobnorm(gramA, lag = filter$len + 1)
identical(hilde(gramA, filter = filter, family = "jsmurfPS", method = "LR",
                startTime = 9, sd = sd), idealisation)

# with less regularisation of the correlation matrix
hilde(gramA, filter = filter, family = "jsmurfPS", method = "LR",
      startTime = 9, regularization = 0.5)

# with estimation of the level of long segments by the mean
# but requiring 30 observations for it
hilde(gramA, filter = filter, family = "jsmurfPS", method = "LR",
      startTime = 9, localValue = 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
hilde(gramA, filter = filter, family = "jsmurfPS", method = "LR",
      startTime = 9, messages = 100,
      lengths = c(3:5, 8, 11, 16, 20),
      gridSize = c(1 / filter$sr, 1 / 10 / filter$sr),
      windowFactorRefinement = 2, report = TRUE,
      suppressWarningNoDeconvolution = TRUE)

# pre-computation of certain quantities using createLocalList
# this saves run time if hilde or (improveSmallScales) is called more than once
# localList is passed via ... to improveSmallScales
localList <- createLocalList(filter = filter, method = "LR")
identical(hilde(gramA, filter = filter, family = "jsmurfPS", method = "LR",
                startTime = 9, localList = localList), idealisation)
```

`improveSmallScales` *Improves small scales*

Description

Implements the second and third step of HILDE (*Pein et al.*, 2020). It refines an initial fit, e.g. obtained by `jsmurf` on small temporal scales by testing for events and local deconvolution. Refinment

can be done assuming homogeneous noise, but also allow heterogeneous noise. Please see the argument `method` and the *examples* for how to access the function correctly depending on whether homogeneous noise is assumed or heterogeneous noise is allowed. Further details about how to decide whether the noise is homogeneous or heterogeneous and whether events are short are given in the accompanying vignette.
If `q == NULL` a Monte-Carlo simulation is required for computing critical values. 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 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

```
improveSmallScales(fit, data, filter, method = c("2Param", "LR"),
                   lengths = NULL, q = NULL, alpha = 0.04, r = 1e3, startTime = 0,
                     thresholdLongSegment = if (method == "LR") 10L else 25L,
                     localValue = stats::median,
                     localVar = function(data) stepR::sdrobnorm(data,
                                                 lag = filter$len + 1L)^2,
                     regularization = 1,
                     gridSize = c(1, 1 / 10, 1 / 100) / filter$sr,
                     windowFactorRefinement = 1,
                     output = c("onlyIdealization", "everyGrid"), report = FALSE,
                     suppressWarningNoDeconvolution = FALSE,
                     localList = NULL, ...)
```

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 `jsmurf` with `locationCorrection == "none"`
`data`	a numeric vector containing the recorded data points
`filter`	an object of class `lowpassFilter` giving the used analogue lowpass filter
`method`	the testing method for short events, `"2Param"` allows for heterogeneous noise, `"LR"` assumes homogeneous noise
`lengths`	a vector of integers giving the lengths on which tests will be performed to detect short events, should be chosen such that events on larger scales are already contained in `fit`
`q`	a numeric vector of the same length as `lengths` giving scale dependent critical values for the tests for short events, 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 values (if `q == NULL`), see `getCritVal`. Its choice balances the risks of missing short events and detecting additional artefacts

`r` an integer giving the number of Monte-Carlo simulations (if `q == NULL`), a larger value increases accuracy, but also the run time

`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`

`thresholdLongSegment` a single integer giving the threshold determining how many observations are necessary to estimate parameters, conductance value and its variance, without deconvolution; has to be chosen such that `localValue` and `localVar` can be applied to a vector of this and larger length

`localValue` a function for estimating the conductance levels on long segments, see *details*, will be called with `localValue(data[i:j])` with `i` and `j` two integers in `1:length(data)` and `j - i >= thresholdLongSegment`, has to return a single numeric

`localVar` a function for estimating the variance on long segments, see *details*, will be called with `localVar(data[i:j])` with `i` and `j` two integers in `1:length(data)` and `j - i >= thresholdLongSegment`, has to return a single, positive numeric

`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

`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 refinement of the grid, see *details*. If a single numeric is given its value is used in all refinement steps

`output` a string specifying 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

`localList` an object of class `"localList"`, usually computed by `createLocalList`, doing such a pre-computation saves run time if `improveSmallScales` or `hilde` is called multiple times with the same arguments

`...` 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, argument `method` will be assigned to `family` 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

Details

First of al, tests for additional short events are performed. Those tests take into account the lowpass filter explicitly. Afterwards all conductance levels (of the newly found and of the already existing

event) and locations of the conductance changes are determined by local deconvolution. The local deconvolution consists of two parts.
In the first part, all segments of the initial fit, potentially interrupted by newly detected events, 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 parameters, conductance level and variance, of this segment will be determined by `localValue(data[i:j])` and `localVar(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.*, 2018, (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 testing for short events and local deconvolution. 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*. Moreover, the (computed) q is returned as an attribute.

Storing of Monte-Carlo simulations

If q == NULL a Monte-Carlo simulation is required to compute the critical values. 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 test method, the number of observations and the used filter. Monte-Carlo simulations are also specific to the arguments thresholdLongSegment, localValue and localVar. Currently, storing a Monte-Carlo simulation is only possible for their default values. 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 "matrixIncreased", i.e. objects of class "MCSimulationVector" 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., Bartsch, A., Steinem, C., Munk, A. (2020) Heterogeneous Idealization of Ion Channel Recordings - Open Channel Noise. *arXiv*:2008.02658.

Pein, F., Tecuapetla-Gómez, I., Schütte, O., Steinem, C., Munk, A. (2018) Fully-automatic multiresolution idealization for filtered ion channel recordings: flickering event detection. *IEEE Transactions on NanoBioscience* **17**(3), 300–320.

See Also

hilde, lowpassFilter, createLocalList

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, good on larger temporal scales, but misses short events
# with given q to save computation time
```

```
# this q is specific to length of the data and the filter
fit <- jsmurf(gramA, filter = filter, family = "jsmurfPS", q = 1.775696, startTime = 9,
              locationCorrection = "none")

# improvement on small temporal scales by testing for short events and local deconvolution
# this call requires a Monte-Carlo simulation
# and therefore might last a few minutes,
# progress of the Monte-Carlo simulation is reported
deconvolution <- improveSmallScales(fit, data = gramA, method = "LR", filter = filter,
                                    startTime = 9, messages = 100)

# any second call should be much faster
# as the previous Monte-Carlo simulation will be loaded
# return fit after each refinement
every <- improveSmallScales(fit, data = gramA, method = "LR", filter = filter,
                            startTime = 9, output = "every")
deconvolutionEvery <- every[[3]]
attr(deconvolutionEvery, "noDeconvolution") <- attr(every, "noDeconvolution")
attr(deconvolutionEvery, "q") <- attr(every, "q")
identical(deconvolution, deconvolutionEvery)

# identical to a direct idealisation by hilde
compare <- deconvolution
attr(compare, "q") <- NULL
identical(hilde(gramA, filter = filter, family = "jsmurfPS", method = "LR",
                startTime = 9), compare)

# allowing heterogeneous noise
fitH <- jsmurf(gramA, filter = filter, family = "hjsmurf", r = 100, startTime = 9,
               locationCorrection = "none")
improveSmallScales(fitH, data = gramA, method = "2Param", filter = filter,
                   startTime = 9, messages = 10, r = 100)
# r = 100 is used to reduce its run time,
# this is okay for illustration purposes, but for precise results
# a larger number of Monte-Carlo simulations is recommend


## zoom into a single event,
## similar to (Pein et al., 2018, Figure 2 lower left panel)
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")

# deconvolution
lines(deconvolution, col = "red", lwd = 3)

# deconvolution convolved with the filter
ind <- seq(10.408, 10.411, 1e-6)
convolvedSignal <- lowpassFilter::getConvolution(ind, deconvolution, filter)
lines(ind, convolvedSignal, col = "blue", lwd = 3)

# for comparison, fit prior to the improvement step
# does not contain the event and hence fits the recorded data points badly
```

```
# fit
lines(fit, col = "orange", lwd = 3)

# fit convolved with the filter
ind <- seq(10.408, 10.411, 1e-6)
convolvedSignal <- lowpassFilter::getConvolution(ind, fit, filter)
lines(ind, convolvedSignal, col = "darkgreen", lwd = 3)


# with less regularisation of the correlation matrix
improveSmallScales(fit, data = gramA, method = "LR", filter = filter,
                   startTime = 9, messages = 100, regularization = 0.5)

# with estimation of the level of long segments by the mean
# but requiring 30 observations for it
improveSmallScales(fit, data = gramA, method = "LR", filter = filter,
                   startTime = 9, messages = 100,
                   localValue = mean, thresholdLongSegment = 30)

# with one refinement step less, but with a larger grid
# test are performed on less lengths
# progress of the deconvolution is reported
# potential warning for no deconvolution is suppressed
improveSmallScales(fit, data = gramA, method = "LR", filter = filter,
                   startTime = 9, messages = 100,
                   lengths = c(3:5, 8, 11, 16, 20),
                   gridSize = c(1 / filter$sr, 1 / 10 / filter$sr),
                   windowFactorRefinement = 2, report = TRUE,
                   suppressWarningNoDeconvolution = TRUE)

# pre-computation of quantities using createLocalList
# this saves run time if improveSmallScales (or hilde) is called more than once
localList <- createLocalList(filter = filter, method = "LR")
identical(improveSmallScales(fit, data = gramA, method = "LR", filter = filter,
                             startTime = 9, localList = localList), deconvolution)
```

`jsmurf` *JSMURF*

Description

Implements the **J**ump-**S**egmentation by **MU**ltiResolution **F**ilter (JSMURF) (*Hotz et al.*, 2013). This model-free multiscale idealization approach works well on medium and large temporal scales. If short events (flickering) occurs, `jules` or `hilde` should be used instead. The original work (*Hotz et al.*, 2013) assumed homogeneous noise, but an extension to heterogeneous noise was provided in (*Pein et al.*, 2020). Please see the argument `family` and the *examples* for how to access the function correctly depending on whether homogeneous noise is assumed or heterogeneous noise is allowed. Further details about how to decide whether the noise is homogeneous or heterogeneous and whether events are short are given in the accompanying vignette.

If q == NULL a Monte-Carlo simulation is required for computing critical values. 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 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

```
jsmurf(data, filter, family = c("jsmurf", "jsmurfPS", "jsmurfLR",
                                "hjsmurf", "hjsmurfSPS", "hjsmurfLR"),
       q = NULL, alpha = 0.05, sd = NULL, startTime = 0,
       locationCorrection = c("deconvolution", "constant", "none"),
       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
family	the parametric family for the multiscale test in JSMURF. "jsmurf", "jsmurfPS" and "jsmurfLR" assume homogeneous noise and "hjsmurf", "hjsmurfSPS" and "hjsmurfLR" allow for heterogeneous noise. By default, we recommend to use "jsmurfPS" when homogeneous noise is assumed and "hjsmurf" when heterogeneous noise is allowed, see *examples*. "jsmurf" is the standard statistic from (*Hotz et al.*, 2013), "jsmurfPS" is a slightly more powerful partial sum statistic, "jsmurfLR" is a likelihood-ratio statistic, which is even more powerful but slow. "hjsmurf" is the standard statistic for heterogeneous noise which estimates the variance locally, "hjsmurfSPS" is a studentized partial sum statistic and "hjsmurfLR" is a likelihood ratio statistic, which is more powerful, but very slow
q	by default chosen automatically by getCritVal, for families "jsmurf", "jsmurfPS" and "jsmurfLR" a single numeric giving the critical value, for families "hjsmurf", "hjsmurfSPS" and "hjsmurfLR" a numeric vector giving scale dependent critical values
alpha	a probability, i.e. a single numeric between 0 and 1, giving the significance level to compute 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 conductance changes and detecting additional artefacts
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$len + 1L. For families "hjsmurf", "hjsmurfSPS" and "hjsmurfLR" this argument is ignored with a warning
startTime	a single numeric giving the time at which recording (sampling) of data started, sampling time points will be assumed to be startTime + seq(along = data) / filter$sr

`locationCorrection`
: indicating how the locations of conductance changes are corrected for smoothing effects due to lowpass filtering, if `"deconvolution"` the local deconvolution procedure in `deconvolveLocally` from (*Pein et al.*, 2018) is called, if `"constant"` all changes are moved to the left by the constant `filter$jump / filter$sr`, this was the approach in (*Hotz et al.*, 2013), if `"none"` no correction is applied

`output`
: a string specifying 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 `family`, `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 if `locationCorrection == "deconvolution"`, the obtained 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 JSMURF. 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 before locations of conductance changes were corrected for filtering, `q` with the given / computed critical values, `filter` with the given filter and for families `"jsmurf"`, `"jsmurfPS"` and `"jsmurfLR"` `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 before locations of conductance changes were corrected for filtering, `q` with the given / computed critical values, `filter` with the given filter and for families `"jsmurf"`, `"jsmurfPS"` and `"jsmurfLR"` `sd` with the given / estimated standard deviation. Additionally, if `locationCorrection == "deconvolution"`, 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 values. 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 parametric `family`, 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" or "matrixIncreased", i.e. objects of classes "MCSimulationMaximum" and "MCSimulationVector" 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

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.

Pein, F., Bartsch, A., Steinem, C., Munk, A. (2020) Heterogeneous Idealization of Ion Channel Recordings - Open Channel Noise. *arXiv*:2008.02658.

Pein, F., Tecuapetla-Gómez, I., Schütte, O., Steinem, C., Munk, A. (2018) Fully-automatic multiresolution idealization for filtered ion channel recordings: flickering event detection. *IEEE Transactions on NanoBioscience* **17**(3), 300–320.

See Also

getCritVal, lowpassFilter, deconvolveLocally, jules, hilde

Examples

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

# idealisation by JSMURF assuming homogeneous noise
# this call requires a Monte-Carlo simulation
# and therefore might last a few minutes,
# progress of the Monte-Carlo simulation is reported
idealisation <- jsmurf(gramA, filter = filter, family = "jsmurfPS",
                       startTime = 9, messages = 100)
# detects conductance changes, but misses short events (flickering)
# if they are not of interest the above idealisation is suitable
# otherwise JULES should be used instead

# JSMURF allowing heterogeneous noise
# for illustration, but less appropriate for this dataset
jsmurf(gramA, filter = filter, family = "hjsmurf",
       startTime = 9, messages = 100)

# any second call should be much faster
```

```
# as the previous Monte-Carlo simulation will be loaded
jsmurf(gramA, filter = filter, family = "jsmurfPS",
       startTime = 9, messages = 100)

# 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
jsmurf(gramA, filter = filter, family = "jsmurfPS",
       alpha = 0.9, startTime = 9)


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

# more detailed output
each <-  jsmurf(gramA, filter = filter, family = "jsmurfPS",
                startTime = 9, output = "each")
every <-  jsmurf(gramA, filter = filter, family = "jsmurfPS",
                 startTime = 9, output = "every")

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

## 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")

# idealisation
lines(idealisation, col = "red", lwd = 3)

# idealisation convolved with the filter
ind <- seq(9.647, 9.65, 1e-6)
convolvedSignal <- lowpassFilter::getConvolution(ind, idealisation, filter)
lines(ind, convolvedSignal, col = "blue", 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 simulations are specific the number of observations and the filter
# hence a new simulation is required (if called for the first time)
idealisationWrong <- jsmurf(gramA, filter = wrongFilter, family = "jsmurfPS",
                            startTime = 9, messages = 100)
```

```
# idealisation
lines(idealisationWrong, col = "orange", lwd = 3)

# idealisation convolved with the filter
ind <- seq(9.647, 9.65, 1e-6)
convolvedSignal <- lowpassFilter::getConvolution(ind, idealisationWrong, filter)
lines(ind, convolvedSignal, col = "darkgreen", lwd = 3)


# location correction by a constant, almost the same as the local deconvolution
idealisationConst <- jsmurf(gramA, filter = filter, family = "jsmurfPS",
                          locationCorrection = "constant", startTime = 9, messages = 100)

# idealisation
lines(idealisationConst, col = "brown", lwd = 3)

# idealisation convolved with the filter
ind <- seq(9.647, 9.65, 1e-6)
convolvedSignal <- lowpassFilter::getConvolution(ind, idealisationConst, filter)
lines(ind, convolvedSignal, col = "purple", lwd = 3)

# no correction of locations for filter effects, jump location is shifted to the left
idealisationNone <- jsmurf(gramA, filter = filter, family = "jsmurfPS",
                           locationCorrection = "none", startTime = 9, messages = 100)

# idealisation
lines(idealisationNone, col = "black", lwd = 3)

# idealisation convolved with the filter
ind <- seq(9.647, 9.65, 1e-6)
convolvedSignal <- lowpassFilter::getConvolution(ind, idealisationNone, filter)
lines(ind, convolvedSignal, col = "green", lwd = 3)

# local deconvolution can be called separately
identical(deconvolveLocally(idealisationNone, data = gramA, filter = filter, startTime = 9),
          idealisation)


# 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)
jsmurf(gramA[1:2.99e4], filter = filter, family = "jsmurfPS", 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
jsmurf(gramA[1:1e4], filter = filter, family = "jsmurfPS", startTime = 9,
       nq = 3e4, messages = 100, r = 1e3)
# here a new simulation is required
# (if no appropriate simulation is saved from a previous call)
```

```
jsmurf(gramA[1:1e3], filter = filter, family = "jsmurfPS", 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
jsmurf(gramA, filter = filter, family = "jsmurfPS", startTime = 9,
       messages = 100, r = 1e3, options = list(load = list(), save = list()))

# only simulations of type "vector" and "vectorInceased" will be saved and
# loaded from the workspace, but no simulations of type "matrix" and
# "matrixIncreased" on the file system
jsmurf(gramA[1:1e4], filter = filter, family = "jsmurfPS",
       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 = "jsmurfPS",
                                    filter = filter, output = "maximum",
                                    r = 1e3, messages = 100)
jsmurf(gramA, filter = filter, family = "jsmurfPS", startTime = 9, stat = stat)

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

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

`jules` *JULES*

Description

Implements the **JU**mp **L**ocal d**E**convolution **S**egmentation (JULES) filter (*Pein et al.*, 2018). 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. JULES requires that the underlying noise is homogeneous. If the noise is heterogeneous, `hilde` should be used instead. `hilde` might be also more powerful (but slower) when events are very short. If all events are very long, multiple times the filter length, the simpler approach `jsmurf` is suitable. Further details about how to decide whether the noise is homogeneous or heterogeneous and whether events are short are given in the accompanying vignette.

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, the 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.*, 2018, (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
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 specifying 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, the family = "jules" 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. (2018) Fully-automatic multiresolution idealization for filtered ion channel recordings: flickering event detection. *IEEE Transactions on NanoBioscience* **17**(3), 300–320.

See Also

`getCritVal`, `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)

# any second call should be much faster
# as the previous Monte-Carlo simulation will be loaded
jules(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 11.36972
# 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 run 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., 2018, 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")

# idealisation
lines(idealisation, col = "red", lwd = 3)

# idealisation convolved with the filter
```

```
ind <- seq(10.408, 10.411, 1e-6)
convolvedSignal <- lowpassFilter::getConvolution(ind, idealisation, filter)
lines(ind, convolvedSignal, col = "blue", lwd = 3)

# fit prior to the deconvolution step
# does not fit the recorded data points appropriately
# fit
lines(fit, col = "orange", lwd = 3)

# fit convolved with the filter
ind <- seq(10.408, 10.411, 1e-6)
convolvedSignal <- lowpassFilter::getConvolution(ind, fit, filter)
lines(ind, convolvedSignal, col = "darkgreen", lwd = 3)


## 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")

# idealisation
lines(idealisation, col = "red", lwd = 3)

# idealisation convolved with the filter
ind <- seq(9.647, 9.65, 1e-6)
convolvedSignal <- lowpassFilter::getConvolution(ind, idealisation, filter)
lines(ind, convolvedSignal, col = "blue", 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)

# idealisation
lines(idealisationWrong, col = "orange", lwd = 3)

# idealisation convolved with the filter
ind <- seq(9.647, 9.65, 1e-6)
convolvedSignal <- lowpassFilter::getConvolution(ind, idealisationWrong, filter)
lines(ind, convolvedSignal, col = "darkgreen", 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 = filter, 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 be saved 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

Reexported from `lowpassFilter`. Creates lowpass filters.

Usage

```
lowpassFilter(type = c("bessel"), param, sr = 1, len = NULL, shift = 0.5)
```

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`) it is chosen such that the autocorrelation function is below `1e-3` at `len / sr` and at 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 by evaluating the corresponding functions at `(0:len + shift) / sr`

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

`"acAntiderivative"` the antiderivative of the autocorrelation function

`"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

`"truncatedAcAntiderivative"` the antiderivative of the autocorrelation function of the at `len / sr` truncated filter

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

`"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)`

`"jump"` the last index of the left half of the filter, i.e. `min(which(ret$step >= 0.5)) - 1L`, it indicates how much a jump is shifted in time by a convolution of the signal with the digitised kernel of the lowpassfilter; if all values are below `0.5`, `len` is returned with a `warning`

`"number"` for developers; an integer indicating the type of the filter

`"list"` for developers; a list containing precomputed quantities to recreate the filter in C++

Author(s)

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

References

Pein, F., Bartsch, A., Steinem, C., Munk, A. (2020) Heterogeneous Idealization of Ion Channel Recordings - Open Channel Noise. *arXiv*:2008.02658.

Pein, F., Tecuapetla-Gómez, I., Schütte, O., Steinem, C., Munk, A. (2018) Fully-automatic multiresolution idealization for filtered ion channel recordings: flickering event detection. IEEE Trans. Nanobioscience, 17(3):300-320.

Pein, F. (2017) Heterogeneous Multiscale Change-Point Inference and its Application to Ion Channel Recordings. PhD thesis, Georg-August-Universität Göttingen. http://hdl.handle.net/11858/00-1735-0000-002E-E34A-7.

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 Trans. Nanobioscience, 12(4):376-386.

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")

# idealisation
lines(idealisation, col = "red", lwd = 3)

# idealisation convolved with the filter
ind <- seq(9.647, 9.65, 1e-6)
convolvedSignal <- lowpassFilter::getConvolution(ind, idealisation, filter)
lines(ind, convolvedSignal, col = "blue", 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)

# idealisation
lines(idealisationWrong, col = "orange", lwd = 3)

# idealisation convolved with the filter
ind <- seq(9.647, 9.65, 1e-6)
convolvedSignal <- lowpassFilter::getConvolution(ind, idealisationWrong, filter)
lines(ind, convolvedSignal, col = "darkgreen", 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.*, 2018, Section III-A) 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, the 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.*, 2018, (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
`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 specifying 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, the family = "jules" 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. (2018) Fully-automatic multiresolution idealization for filtered ion channel recordings: flickering event detection. *IEEE Transactions on NanoBioscience* **17**(3), 300–320.

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 (if not performed before)
# 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
# fitting the observations requires a deconvolution
# either by calling deconveLocally,
# or as suggested by calling jules instead of stepDetection
# fit
lines(fit, col = "red", lwd = 3)

# fit convolved with the filter
ind <- seq(10.408, 10.411, 1e-6)
```

```
convolvedSignal <- lowpassFilter::getConvolution(ind, fit, filter)
lines(ind, convolvedSignal, col = "blue", lwd = 3)


# 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 save 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