Package ‘MsCoreUtils’

October 17, 2020

Title Core Utils for Mass Spectrometry Data

Version 1.0.0

Description MsCoreUtils defines low-level functions for mass spectrometry data and is independent of any high-level data structures. These functions include mass spectra processing functions (noise estimation, smoothing, binning), quantitative aggregation functions (median polish, robust summarisation, ...), missing data imputation, data normalisation (quantiles, vsn, ...) as well as misc helper functions, that are used across high-level data structure within the R for Mass Spectrometry packages.

Depends R (>= 3.6.0)

Imports methods, S4Vectors, MASS, stats

Suggests testthat, knitr, BiocStyle, rmarkdown, roxygen2, imputeLCMD, impute, norm, pcaMethods, vsn, preprocessCore

License Artistic-2.0

Encoding UTF-8

VignetteBuilder knitr

LinkingTo Rcpp

BugReports https://github.com/RforMassSpectrometry/MsCoreUtils/issues

URL https://github.com/RforMassSpectrometry/MsCoreUtils

biocViews Infrastructure, Proteomics, MassSpectrometry, Metabolomics

Roxygen list(markdown=TRUE)

RoxygenNote 7.0.2

git_url https://git.bioconductor.org/packages/MsCoreUtils

git_branch RELEASE_3_11

git_last_commit d447cd6

git_last_commit_date 2020-04-27

Date/Publication 2020-10-16

Author RforMassSpectrometry Package Maintainer [cre],
Laurent Gatto [aut] (<https://orcid.org/0000-0002-1520-2268>),
Johannes Rainer [aut] (<https://orcid.org/0000-0002-6977-7147>),
Sebastian Gibb [aut] (<https://orcid.org/0000-0001-7406-4443>),
Adriaan Sticker [ctb],
Sigurdur Smarason [ctb],
Thomas Naake [ctb]

Maintainer

RforMassSpectrometry Package Maintainer <maintainer@rformassspectrometry.org>

R topics documented:

aggregate_by_vector . . . 2
between . . . 3
bin . . . 4
closest . . . 5
coerce . . . 8
dotproduct . . . 9
group . . . 10
i2index . . . 11
impute_matrix . . . 12
isPeaksMatrix . . . 15
localMaxima . . . 16
medianPolish . . . 17
noise . . . 17
normalizeMethods . . . 18
ppm . . . 20
rbindFill . . . 20
refineCentroids . . . 21
rla . . . 22
robustSummary . . . 24
smooth . . . 24
validPeaksMatrix . . . 26
valleys . . . 27
vapply1c . . . 28

Index **29**

`aggregate_by_vector` *Aggreagate quantitative features.*

Description

This function takes a matrix of quantitative features `x` and a factor (of length equal to `nrow(x)`) defining subsets, and applies a user-defined function to aggregate each subset into a vector of quantitative values.

User-defined functions must thus return a vector of length equal to `ncol(x)`. Examples thereof are

- `medianPolish()` to fits an additive model (two way decomposition) using Tukey's median polish_ procedure using `stats::medpolish()`;
- `robustSummary()` to calculate a robust aggregation using `MASS::rlm()`;
- `base::colMeans()` to use the mean of each column;
- `base::colSums()` to use the sum of each column;
- `matrixStats::colMedians()` to use the median of each column.

Usage

```
aggregate_by_vector(x, INDEX, FUN, ...)
```

Arguments

x	A matrix of mode numeric.
INDEX	A factor of length nrow(x).
FUN	A function to be applied to the subsets of x.
...	Additional arguments passed to FUN.

Value

A new matrix of dimensions ncol(x) and length(INDEX) with dimnames equal to colnames(x) and INDEX.

Author(s)

Laurent Gatto

See Also

Other Quantitative feature aggregation: medianPolish(), robustSummary()

Examples

```
x <- structure(c(10.3961935744407, 17.1663715212693, 14.1027587989326,
                 12.850349037785, 10.6379251053134, 7.52885076885599,
                 3.91816118984218, 11.1339832690524, 16.5321471730746,
                 14.1787908569268, 11.9422579479634, 11.5154097311056,
                 7.69906817878979, 3.97092153807337, 11.9394664781386,
                 15.3791100898935, 14.2409281956285, 11.2106867261254,
                 12.2958526883634, 9.00858488668671, 3.83120129974963,
                 12.9033445520186, 14.375814954807, 14.1617803596661,
                 10.1237981632645, 13.3390344671153, 9.75719265786117,
                 3.81046169359919),
               .Dim = c(7L, 4L),
               .Dimnames = list(c("X1", "X27", "X41", "X47", "X52",
                                  "X53", "X55"),
                                c("iTRAQ4.114", "iTRAQ4.115",
                                  "iTRAQ4.116", "iTRAQ4.117")))
x

k <- factor(c("B", "E", "X", "E", "B", "B", "E"))

aggregate_by_vector(x, k, colMeans)
aggregate_by_vector(x, k, robustSummary)
aggregate_by_vector(x, k, medianPolish)
```

---

between — *Range helper functions*

Description

These functions help to work with numeric ranges.

Usage

```
between(x, range)

x %between% range
```

Arguments

x	numeric, input values.
range	numeric(2), range to compare against.

Value

logical vector of length length(x).

Author(s)

Sebastian Gibb

See Also

Other helper functions for developers: isPeaksMatrix(), rbindFill(), validPeaksMatrix(), vapply1c()

Examples

```
between(1:4, 2:3)
1:4 %between% 2:3
```

---

bin *Binning*

Description

Aggregate values in x for bins defined on y: all values in x for values in y falling into a bin (defined on y) are aggregated with the provided function FUN.

Usage

```
bin(
  x,
  y,
  size = 1,
  breaks = seq(floor(min(y)), ceiling(max(y)), by = size),
  FUN = max
)
```

Arguments

x	numeric with the values that should be aggregated/binned.
y	numeric with same length than x with values to be used for the binning.
size	numeric(1) with the size of a bin.
breaks	numeric defining the breaks (bins).
FUN	function to be used to aggregate values of x falling into the bins defined by breaks.

Value

list with elements x (aggregated values of x) and mids (the bin mid points).

Author(s)

Johannes Rainer, Sebastian Gibb

See Also

Other grouping/matching functions: closest()

Examples

```
## Define example intensities and m/z values
ints <- abs(rnorm(20, mean = 40))
mz <- seq(1:length(ints)) + rnorm(length(ints), sd = 0.001)

## Bin intensities by m/z bins with a bin size of 2
bin(ints, mz, size = 2)

## Repeat but summing up intensities instead of taking the max
bin(ints, mz, size = 2, FUN = sum)
```

---

closest *Relaxed Value Matching*

Description

These functions offer relaxed matching of one vector in another. In contrast to the similar match() and %in% functions they just accept numeric arguments but have an additional tolerance argument that allows relaxed matching.

Usage

```
closest(
  x,
  table,
  tolerance = Inf,
  ppm = 0,
  duplicates = c("keep", "closest", "remove"),
  nomatch = NA_integer_
```

```
)

common(
  x,
  table,
  tolerance = Inf,
  ppm = 0,
  duplicates = c("keep", "closest", "remove")
)

join(x, y, tolerance = 0, ppm = 0, type = c("outer", "left", "right", "inner"))
```

Arguments

x	numeric, the values to be matched.
table	numeric, the values to be matched against. In contrast to match() table has to be sorted in increasing order.
tolerance	numeric, accepted tolerance. Could be of length one or the same length as table.
ppm	numeric(1) representing a relative, value-specific parts-per-million (PPM) tolerance that is added to tolerance.
duplicates	character(1), how to handle duplicated matches.
nomatch	numeric(1), if the difference between the value in x and table is larger than tolerance nomatch is returned.
y	numeric, the values to be joined. Should be sorted.
type	character(1), defines how x and y should be joined. See details for join.

Details

For closest/common the tolerance argument could be set to 0 to get the same results as for match()/%in%. If it is set to Inf (default) the index of the closest values is returned without any restriction.

It is not guaranteed that there is a one-to-one matching for neither the x to table nor the table to x matching.

If multiple elements in x match a single element in table all their corresponding indices are returned if duplicates="keep" is set (default). This behaviour is identical to match(). For duplicates="closest" just the closest element in x gets the corresponding index in table and for duplicates="remove" all elements in x that match to the same element in table are set to nomatch.

If a single element in x matches multiple elements in table the *closest* is returned for duplicates="keep" or duplicates="duplicates" (*keeping* multiple matches isn't possible in this case because the implementation relies on findInterval). If the differences between x and the corresponding matches in table are identical the lower index (the smaller element in table) is returned. For duplicates="remove" all multiple matches are returned as nomatch as above.

join: joins two numeric vectors by mapping values in x with values in y and *vice versa* if they are similar enough (provided the tolerance and ppm specified). The function returns a matrix with the indices of mapped values in x and y. Parameter type allows to define how the vectors will be joined: type = "left": values in x will be mapped to values in y, elements in y not matching any value in x will be discarded. type = "right": same as type = "left" but for y. type = "outer": return matches for all values in x and in y. type = "inner": report only indices of values that could be mapped.

Value

`closest` returns an `integer` vector of the same length as `x` giving the closest position in `table` of the first match or `nomatch` if there is no match.

`common` returns a `logical` vector of length `x` that is `TRUE` if the element in `x` was found in `table`. It is similar to `%in%`.

`join` returns a `matrix` with two columns, namely `x` and `y`, representing the index of the values in `x` matching the corresponding value in `y` (or `NA` if the value does not match).

Note

`closest` will replace all `NA` values in `x` by `nomatch` (that is identical to the behaviour of `match`).

`join` is based on `closest(x,y,tolerance,duplicates = "closest")`. That means for multiple matches just the closest one is reported.

Author(s)

Sebastian Gibb

See Also

`match()`

`%in%`

Other grouping/matching functions: `bin()`

Examples

```
## Define two vectors to match
x <- c(1, 3, 5)
y <- 1:10

## Compare match and closest
match(x, y)
closest(x, y)

## If there is no exact match
x <- x + 0.1
match(x, y) # no match
closest(x, y)

## Some new values
x <- c(1.11, 45.02, 556.45)
y <- c(3.01, 34.12, 45.021, 46.1, 556.449)

## Using a single tolerance value
closest(x, y, tolerance = 0.01)

## Using a value-specific tolerance accepting differences of 20 ppm
closest(x, y, tolerance = ppm(y, 20))

## Same using 50 ppm
closest(x, y, tolerance = ppm(y, 50))

## Sometimes multiple elements in `x` match to `table`
x <- c(1.6, 1.75, 1.8)
```

```
y <- 1:2
closest(x, y, tolerance = 0.5)
closest(x, y, tolerance = 0.5, duplicates = "closest")
closest(x, y, tolerance = 0.5, duplicates = "remove")

## Are there any common values?
x <- c(1.6, 1.75, 1.8)
y <- 1:2
common(x, y, tolerance = 0.5)
common(x, y, tolerance = 0.5, duplicates = "closest")
common(x, y, tolerance = 0.5, duplicates = "remove")

## Join two vectors
x <- c(1, 2, 3, 6)
y <- c(3, 4, 5, 6, 7)

jo <- join(x, y, type = "outer")
jo
x[jo$x]
y[jo$y]

jl <- join(x, y, type = "left")
jl
x[jl$x]
y[jl$y]

jr <- join(x, y, type = "right")
jr
x[jr$x]
y[jr$y]

ji <- join(x, y, type = "inner")
ji
x[ji$x]
y[ji$y]
```

`coerce` *Coerce functions*

Description

- `asInteger`: convert `x` to an `integer` and throw an error if `x` is not a `numeric`.

Usage

```
asInteger(x)
```

Arguments

`x`	input argument.

Author(s)

Johannes Rainer

Examples

```
## Convert numeric to integer
asInteger(3.4)

asInteger(3)
```

`dotproduct` *Calculate the normalized dot product*

Description

Calculate the normalized dot product (NDP).`dotproduct` returns a numeric value ranging between 0 and 1, where 0 indicates no similarity between the two MS/MS features, while 1 indicates that the MS/MS features are identical.

Usage

```
dotproduct(x, y, m = 0.5, n = 0)
```

Arguments

`x`	`matrix` with two column where one contains m/z values (column `"mz"`) and the second corresponding intensity values (column `"intensity"`)
`y`	`matrix` with two column where one contains m/z values (column `"mz"`) and the second corresponding intensity values (column `"intensity"`)
`m`	`numeric(1)`, exponent for peak intensity-based weights
`n`	`numeric(1)`, exponent for m/z-based weights

Details

Each row in `x` corresponds to the respective row in `y`, i.e. the peaks (entries `"mz"`) per spectrum have to match.

`m` and `n` are weights given on the peak intensity and the m/z values respectively. As default (`m = 0.5`), the square root of the intensity values are taken to calculate weights. With increasing values for `m`, high intensity values become more important for the similarity calculation, i.e. the differences between intensities will be aggravated. With increasing values for `n`, high m/z values will be taken more into account for similarity calculation. Especially when working with small molecules, a value `n > 0` can be set, to give a weight on the m/z values to accommodate that shared fragments with higher m/z are less likely and will mean that molecules might be more similar. If `n != 0`, a warning will be raised if the corresponding m/z values are not identical, since small differences in m/z values will distort the similarity values with increasing `n`. If `m=0` or `n=0`, intensity values or m/z values, respectively, are not taken into account.

The normalized dot product is calculated according to:

$$NDP = \frac{\sum(W_{S1,i} \cdot W_{S2,i})^2}{\sum(W_{S1,i}^2) \cdot \sum(W_{S2,i}^2)}$$

, with $W = [peak \quad intensity]^m \cdot [m/z]^n$. For further information on normalized dot product see for example Li et al. (2015). Prior to calculating W_{S1} or W_{S2}, all intensity values are divided by the maximum intensity value and multiplied by 100.

Value

`numeric(1)`, `dotproduct` returns a numeric similarity coefficient between 0 and 1.

Author(s)

Thomas Naake, <thomasnaake@googlemail.com>

References

Li et al. (2015): Navigating natural variation in herbivory-induced secondary metabolism in coyote tobacco populations using MS/MS structural analysis. PNAS, E4147–E4155, doi: 10.1073/pnas.1503106112.

Examples

```
x <- matrix(c(c(100.001, 100.002, NA, 300.01, 300.02, NA),
        c(2, 1.5, 0, 1.2, 0.9, 0)), ncol = 2,)
y <- matrix(c(c(100.0, NA, 200.0, 300.002, 300.025, 300.0255),
        c(2, 0, 3, 1, 4, 0.4)), ncol = 2)
colnames(x) <- colnames(y) <- c("mz", "intensity")
dotproduct(x, y, m = 0.5, n = 0)
```

---

group — *Grouping of numeric values by similarity*

Description

The `group` function groups numeric values by first ordering and then putting all values into the same group if their difference is smaller defined by parameters `tolerance` (a constant value) and `ppm` (a value-specific relative value expressed in parts-per-million).

Usage

```
group(x, tolerance = 0, ppm = 0)
```

Arguments

`x`	increasingly ordered `numeric` with the values to be grouped.
`tolerance`	`numeric(1)` with the maximal accepted difference between values in `x` to be grouped into the same entity.
`ppm`	`numeric(1)` defining a value-dependent maximal accepted difference between values in `x` expressed in parts-per-million.

Value

`integer` of length equal to `x` with the groups.

Note

Since grouping is performed on pairwise differences between consecutive values (after ordering `x`), the difference between the smallest and largest value in a group can be larger than `tolerance` and `ppm`.

Author(s)

Johannes Rainer, Sebastin Gibb

Examples

```
## Define a (sorted) numeric vector
x <- c(34, 35, 35, 35 + ppm(35, 10), 56, 56.05, 56.1)

## With `ppm = 0` and `tolerance = 0` only identical values are grouped
group(x)

## With `tolerance = 0.05`
group(x, tolerance = 0.05)

## Also values 56, 56.05 and 56.1 were grouped into a single group,
## although the difference between the smallest 56 and largest value in
## this group (56.1) is 0.1. The (pairwise) difference between the ordered
## values is however 0.05.

## With ppm
group(x, ppm = 10)

## Same on an unsorted vector
x <- c(65, 34, 65.1, 35, 66, 65.2)
group(x, tolerance = 0.1)

## Values 65, 65.1 and 65.2 have been grouped into the same group.
```

---

i2index — *Input parameter check for subsetting operations*

Description

`i2index` is a simple helper function to be used in subsetting functions. It checks and converts the parameter `i`, which can be of type `integer`, `logical` or `character` to integer vector that can be used as index for subsetting.

Usage

```
i2index(i, length = length(i), names = NULL)
```

Arguments

`i`	`character` `logical` or `integer` used in [i] for subsetting.
`length`	`integer` representing the `length` of the object to be subsetted.
`names`	`character` with the names (rownames or similar) of the object. This is only required if `i` is of type `character`.

Value

`integer` with the indices

Author(s)

Johannes Rainer

Examples

```
## With `i` being an `integer`
i2index(c(4, 1, 3), length = 10)

## With `i` being a `logical`
i2index(c(TRUE, FALSE, FALSE, TRUE, FALSE), length = 5)

## With `i` being a `character`
i2index(c("b", "a", "d"), length = 5, names = c("a", "b", "c", "d", "e"))
```

---

impute_matrix *Quantitative mass spectrometry data imputation*

Description

The `impute_matrix` function performs data imputation on `matrix` objects instance using a variety of methods (see below).

Users should proceed with care when imputing data and take precautions to assure that the imputation produce valid results, in particular with naive imputations such as replacing missing values with 0.

Usage

```
impute_matrix(x, method, ...)

imputeMethods()

impute_neighbour_average(x, k = min(x, na.rm = TRUE))

impute_knn(x, ...)

impute_mle(x, ...)

impute_bpca(x, ...)

impute_mixed(x, randna, mar, mnar, ...)

impute_min(x)
```

Arguments

`x`	A matrix with missing values to be imputed.
`method`	`character(1)` defining the imputation method. See `imputeMethods()` for available ones.
`...`	Additional parameters passed to the inner imputation function.

k	numeric(1) providing the imputation value used for the first and last samples if they contain an NA. The default is to use the smallest value in the data.
randna	logical of length equal to nrow(object) defining which rows are missing at random. The other ones are considered missing not at random. Only relevant when methods is mixed.
mar	Imputation method for values missing at random. See method above.
mnar	Imputation method for values missing not at random. See method above.

Details

There are two types of mechanisms resulting in missing values in LC/MSMS experiments.

- Missing values resulting from absence of detection of a feature, despite ions being present at detectable concentrations. For example in the case of ion suppression or as a result from the stochastic, data-dependent nature of the MS acquisition method. These missing value are expected to be randomly distributed in the data and are defined as missing at random (MAR) or missing completely at random (MCAR).
- Biologically relevant missing values resulting from the absence of the low abundance of ions (below the limit of detection of the instrument). These missing values are not expected to be randomly distributed in the data and are defined as missing not at random (MNAR).

MNAR features should ideally be imputed with a left-censor method, such as QRILC below. Conversely, it is recommended to use host deck methods such nearest neighbours, Bayesian missing value imputation or maximum likelihood methods when values are missing at random.

Currently, the following imputation methods are available.

- *MLE*: Maximum likelihood-based imputation method using the EM algorithm. Implemented in the norm::imp.norm(). function. See norm::imp.norm() for details and additional parameters. Note that here, ... are passed to the [norm::em.norm()function, rather to the actual imputation functionimp.norm‘.
- *bpca*: Bayesian missing value imputation are available, as implemented in the pcaMethods::pca() function. See pcaMethods::pca() for details and additional parameters.
- *knn*: Nearest neighbour averaging, as implemented in the impute::impute.knn function. See impute::impute.knn()] for details and additional parameters.
- *QRILC*: A missing data imputation method that performs the imputation of left-censored missing data using random draws from a truncated distribution with parameters estimated using quantile regression. Implemented in the imputeLCMD::impute.QRILC function. imputeLCMD::impute.QRILC() for details and additional parameters.
- *MinDet*: Performs the imputation of left-censored missing data using a deterministic minimal value approach. Considering a expression data with n samples and p features, for each sample, the missing entries are replaced with a minimal value observed in that sample. The minimal value observed is estimated as being the q-th quantile (default q = 0.01) of the observed values in that sample. Implemented in the imputeLCMD::impute.MinDet function. See imputeLCMD::impute.MinDet() for details and additional parameters.
- *MinProb*: Performs the imputation of left-censored missing data by random draws from a Gaussian distribution centred to a minimal value. Considering an expression data matrix with n samples and p features, for each sample, the mean value of the Gaussian distribution is set to a minimal observed value in that sample. The minimal value observed is estimated as being the q-th quantile (default q = 0.01) of the observed values in that sample. The standard deviation is estimated as the median of the feature standard deviations. Note that when estimating the standard deviation of the Gaussian distribution, only the peptides/proteins which present more

than 50\ are considered. Implemented in the `imputeLCMD::impute.MinProb` function. See `imputeLCMD::impute.MinProb()` for details and additional parameters.

- *min*: Replaces the missing values with the smallest non-missing value in the data.
- *zero*: Replaces the missing values with 0.
- *mixed*: A mixed imputation applying two methods (to be defined by the user as `mar` for values missing at random and `mnar` for values missing not at random, see example) on two MCAR/MNAR subsets of the data (as defined by the user by a `randna` logical, of length equal to nrow(object)).
- *nbavg*: Average neighbour imputation for fractions collected along a fractionation/separation gradient, such as sub-cellular fractions. The method assumes that the fraction are ordered along the gradient and is invalid otherwise.

  Continuous sets `NA` value at the beginning and the end of the quantitation vectors are set to the lowest observed value in the data or to a user defined value passed as argument `k`. Then, when a missing value is flanked by two non-missing neighbouring values, it is imputed by the mean of its direct neighbours.
- *with*: Replaces all missing values with a user-provided value.
- *none*: No imputation is performed and the missing values are left untouched. Implemented in case one wants to only impute value missing at random or not at random with the *mixed* method.

The `imputeMethods()` function returns a vector with valid imputation method arguments.

Author(s)

Laurent Gatto

References

Olga Troyanskaya, Michael Cantor, Gavin Sherlock, Pat Brown, Trevor Hastie, Robert Tibshirani, David Botstein and Russ B. Altman, Missing value estimation methods for DNA microarrays Bioinformatics (2001) 17 (6): 520-525.

Oba et al., A Bayesian missing value estimation method for gene expression profile data, Bioinformatics (2003) 19 (16): 2088-2096.

Cosmin Lazar (2015). imputeLCMD: A collection of methods for left-censored missing data imputation. R package version 2.0. http://CRAN.R-project.org/package=imputeLCMD.

Lazar C, Gatto L, Ferro M, Bruley C, Burger T. Accounting for the Multiple Natures of Missing Values in Label-Free Quantitative Proteomics Data Sets to Compare Imputation Strategies. J Proteome Res. 2016 Apr 1;15(4):1116-25. doi: 10.1021/acs.jproteome.5b00981. PubMed PMID:26906401.

Examples

```
## test data
set.seed(42)
m <- matrix(rlnorm(60), 10)
dimnames(m) <- list(letters[1:10], LETTERS[1:6])
m[sample(60, 10)] <- NA

## available methods
imputeMethods()
```

```
impute_matrix(m, method = "zero")

impute_matrix(m, method = "min")

impute_matrix(m, method = "knn")

## same as impute_zero
impute_matrix(m, method = "with", val = 0)

## impute with half of the smalles value
impute_matrix(m, method = "with",
              val = min(m, na.rm = TRUE) * 0.5)

## all but third and fourth features' missing values
## are the result of random missing values
randna <- rep(TRUE, 10)
randna[c(3, 9)] <- FALSE

impute_matrix(m, method = "mixed",
              randna = randna,
              mar = "knn",
              mnar = "min")
```

`isPeaksMatrix` *Check functions*

Description

These functions are used to check input arguments.

Usage

```
isPeaksMatrix(x)
```

Arguments

x	object to test.

Details

`isPeaksMatrix`: test for a `numeric` matrix with two columns named "mz" and "intensity". The "mz" column has to be sorted increasingly.

Value

`logical(1)`, `TRUE` if checks are successful otherwise `FALSE`.

Author(s)

Sebastian Gibb

See Also

Other helper functions for developers: `between()`, `rbindFill()`, `validPeaksMatrix()`, `vapply1c()`

Examples

```
isPeaksMatrix(1:2)
isPeaksMatrix(cbind(mz = 2:1, intensity = 1:2))
isPeaksMatrix(cbind(mz = 1:2, intensity = 1:2))
```

`localMaxima` *Local Maxima*

Description

This function finds local maxima in a numeric vector. A local maximum is defined as maximum in a window of the current index +/- `hws`.

Usage

```
localMaxima(x, hws = 1L)
```

Arguments

`x`	`numeric`, vector that should be searched for local maxima.
`hws`	`integer(1)`, half window size, the resulting window reaches from `(i -hws):(i + hws)`.

Value

A `logical` of the same length as `x` that is `TRUE` for each local maxima.

Author(s)

Sebastian Gibb

See Also

Other extreme value functions: `.peakRegionMask()`, `refineCentroids()`, `valleys()`

Examples

```
x <- c(1:5, 4:1, 1:10, 9:1, 1:5, 4:1)
localMaxima(x)
localMaxima(x, hws = 10)
```

medianPolish — *Return the Median Polish (Robust Twoway Decomposition) of a matrix*

Description

Fits an additive model (two way decomposition) using Tukey's median polish procedure using `stats::medpolish()`.

Usage

```
medianPolish(x, verbose = FALSE, ...)
```

Arguments

`x`	A `matrix` of mode `numeric`.
`verbose`	Default is `FALSE`.
`...`	Additional arguments passed to `stats::medpolish()`.

Value

A `numeric` vector of length identical to `ncol(x)`.

Author(s)

Laurent Gatto

See Also

Other Quantitative feature aggregation: `aggregate_by_vector()`, `robustSummary()`

Examples

```
x <- matrix(rnorm(30), nrow = 3)
medianPolish(x)
```

noise — *Noise Estimation*

Description

This functions estimate the noise in the data.

Usage

```
noise(x, y, method = c("MAD", "SuperSmoother"), ...)
```

Arguments

x	numeric, x values for noise estimation (e.g. *mz*)
y	numeric, y values for noise estimation (e.g. intensity)
method	character(1) used method. Currently MAD (median absolute deviation) and Friedman's SuperSmoother are supported.
...	further arguments passed to method.

Value

A numeric of the same length as x with the estimated noise.

Author(s)

Sebastian Gibb

See Also

stats::mad(), stats::supsmu()

Other noise estimation and smoothing functions: smooth()

Examples

```
x <- 1:20
y <- c(1:10, 10:1)
noise(x, y)
noise(x, y, method = "SuperSmoother", span = 1 / 3)
```

---

normalizeMethods *Quantitative data normalisation*

Description

Function to normalise a matrix of quantitative omics data. The nature of the normalisation is controlled by the method argument, described below.

Usage

```
normalizeMethods()

normalize_matrix(x, method, ...)
```

Arguments

x	A matrix to be normalised.
method	character(1) defining the normalisation method. See normalizeMethods() for available ones.
...	Additional parameters passed to the inner normalisation function.

Details

The method parameter can be one of "sum", "max", "center.mean", "center.median", "div.mean", "div.median", "diff.meda", "quantiles", "quantiles.robust" or "vsn". The normalizeMethods() function returns a vector of available normalisation methods.

- For "sum" and "max", each feature's intensity is divided by the maximum or the sum of the feature respectively. These two methods are applied along the features (rows).
- "center.mean" and "center.median" center the respective sample (column) intensities by subtracting the respective column means or medians. "div.mean" and "div.median" divide by the column means or medians.
- "diff.median" centers all samples (columns) so that they all match the grand median by subtracting the respective columns medians differences to the grand median.
- Using "quantiles" or "quantiles.robust" applies (robust) quantile normalisation, as implemented in preprocessCore::normalize.quantiles() and preprocessCore::normalize.quantiles.robu "vsn" uses the vsn::vsn2() function. Note that the latter also glog-transforms the intensities. See respective manuals for more details and function arguments.

Value

A normalised matrix of dimensions dim(x).

Author(s)

Laurent Gatto

See Also

The scale() function that centers (like center.mean above) and scales.

Examples

```
normalizeMethods()

## test data
set.seed(42)
m <- matrix(rlnorm(60), 10)

normalize_matrix(m, method = "sum")

normalize_matrix(m, method = "max")

normalize_matrix(m, method = "quantiles")

normalize_matrix(m, method = "center.mean")
```

ppm — *PPM - Parts per Million*

Description

ppm is a small helper function to determine the parts-per-million for a user-provided value and ppm.

Usage

```
ppm(x, ppm)
```

Arguments

x	numeric, value(s) used for ppm calculation, e.g. mz value(s).
ppm	numeric, parts-per-million (ppm) value(s).

Value

numeric: parts-per-million of x

Author(s)

Sebastian Gibb

Examples

```
ppm(c(1000, 2000), 5)
```

rbindFill — *Combine R Objects by Row*

Description

This function combines instances of matrix, data.frame or DataFrame objects into a single instance adding eventually missing columns (filling them with NAs).

Usage

```
rbindFill(...)
```

Arguments

...	2 or more: matrix, data.frame or DataFrame.

Value

Depending on the input a single matrix, data.frame or DataFrame.

Note

rbindFill might not work if one of the columns contains S4 classes.

Author(s)

Johannes Rainer, Sebastian Gibb

See Also

Other helper functions for developers: `between()`, `isPeaksMatrix()`, `validPeaksMatrix()`, `vapply1c()`

Examples

```
## Combine matrices
a <- matrix(1:9, nrow = 3, ncol = 3)
colnames(a) <- c("a", "b", "c")
b <- matrix(1:12, nrow = 3, ncol = 4)
colnames(b) <- c("b", "a", "d", "e")
rbindFill(a, b)
rbindFill(b, a, b)
```

`refineCentroids` *Refine Peak Centroids*

Description

This function refines the centroided values of a peak by weighting the y values in the neighbourhood that belong most likely to the same peak.

Usage

```
refineCentroids(x, y, p, k = 2L, threshold = 0.33, descending = FALSE)
```

Arguments

`x`	`numeric`, i.e. m/z values.
`y`	`numeric`, i.e. intensity values.
`p`	`integer`, indices of identified peaks/local maxima.
`k`	`integer(1)`, number of values left and right of the peak that should be considered in the weighted mean calculation.
`threshold`	`double(1)`, proportion of the maximal peak intensity. Just values above are used for the weighted mean calclulation.
`descending`	`logical`, if `TRUE` just values between the nearest valleys around the peak centroids are used.

Details

For `descending = FALSE` the function looks for the `k` nearest neighbouring data points and use their `x` for weighted mean with their corresponding `y` values as weights for calculation of the new peak centroid. If `k` are chosen too large it could result in skewed peak centroids, see example below. If `descending = TRUE` is used the `k` should be general larger because it is trimmed automatically to the nearest valleys on both sides of the peak so the problem with skewed centroids is rare.

Author(s)

Sebastian Gibb, Johannes Rainer

See Also

Other extreme value functions: `.peakRegionMask()`, `localMaxima()`, `valleys()`

Examples

```
ints <- c(5, 8, 12, 7, 4, 9, 15, 16, 11, 8, 3, 2, 3, 9, 12, 14, 13, 8, 3)
mzs <- seq_along(ints)

plot(mzs, ints, type = "h")

pidx <- as.integer(c(3, 8, 16))
points(mzs[pidx], ints[pidx], pch = 16)

## Use the weighted average considering the adjacent mz
mzs1 <- refineCentroids(mzs, ints, pidx,
                        k = 2L, descending = FALSE, threshold = 0)
mzs2 <- refineCentroids(mzs, ints, pidx,
                        k = 5L, descending = FALSE, threshold = 0)
mzs3 <- refineCentroids(mzs, ints, pidx,
                        k = 5L, descending = TRUE, threshold = 0)
points(mzs1, ints[pidx], col = "red", type = "h")
## please recognize the artificial moved centroids of the first peak caused
## by a too large k, here
points(mzs2, ints[pidx], col = "blue", type = "h")
points(mzs3, ints[pidx], col = "green", type = "h")
legend("topright",
       legend = paste0("k = ", c(2, 5, 5),
                       ", descending =", c("FALSE", "FALSE", "TRUE")),
       col = c("red", "blue", "green"), lwd = 1)
```

---

rla — *Calculate relative log abundances* `rla` *calculates the relative log abundances (RLA, see reference) on a* `numeric` *vector.* `rowRla` *performs row-wise RLA calculations on a numeric* `matrix`.

Description

Calculate relative log abundances

`rla` calculates the relative log abundances (RLA, see reference) on a `numeric` vector. `rowRla` performs row-wise RLA calculations on a numeric `matrix`.

Usage

```
rla(
  x,
  f = rep_len(1, length(x)),
  transform = c("log2", "log10", "identity"),
  na.rm = TRUE
```

```
)

rowRla(x, f, transform = c("log2", "log10", "identity"))
```

Arguments

x	numeric (for rla) or matrix (for rowRla) with the abundances (in natural scale) on which the RLA should be calculated.
f	factor, numeric or character with the same length than x (or, for rowRla equal to the number of columns of x) allowing to define the grouping of values in x. If omitted all values are considered to be from the same group.
transform	character(1) defining the function to transform x. Defaults to transform = "log2" which log2 transforms x prior to calculation. If x is already in log scale use transform = "identity" to avoid transformation of the values.
na.rm	logical(1) whether NA values should be removed prior to calculation of the group-wise medians.

Details

The RLA is defined as the (log2) abundance of an analyte relative to the median across all abundances of that analyte in samples of the same group. The grouping of values can be defined with parameter f.

Value

numeric with the relative log abundances (in log2 scale) with the same length than x (for rla) or matrix with the same dimensions than x (for rowRla).

Author(s)

Johannes Rainer

References

De Livera AM, Dias DA, De Souza D, Rupasinghe T, Pyke J, Tull D, Roessner U, McConville M, Speed TP. Normalizing and integrating metabolomics data. *Anal Chem* 2012 Dec 18;84(24):10768-76.

Examples

```
x <- c(3, 4, 5, 1, 2, 3, 7, 8, 9)

grp <- c(1, 1, 1, 2, 2, 2, 3, 3, 3)

rla(x, grp)

x <- rbind(c(324, 4542, 3422, 3232, 5432, 6535, 3321, 1121),
     c(12, 3341, 3034, 6540, 34, 4532, 56, 1221))
grp <- c("a", "b", "b", "b", "a", "b", "a", "b")

## row-wise RLA values
rowRla(x, grp)
```

robustSummary — *Return the Robust Expression Summary of a matrix*

Description

This function calculates the robust summarisation for each feature (protein). Note that the function assumes that the intensities in input e are already log-transformed.

Usage

```
robustSummary(x, ...)
```

Arguments

x	A feature by sample `matrix` containing quantitative data with mandatory `colnames` and `rownames`.
...	Additional arguments passed to `MASS::rlm()`.

Value

`numeric()` vector of length `ncol(x)` with robust summarised values.

Author(s)

Adriaan Sticker, Sebastian Gibb and Laurent Gatto

See Also

Other Quantitative feature aggregation: `aggregate_by_vector()`, `medianPolish()`

Examples

```
x <- matrix(rnorm(30), nrow = 3)
colnames(x) <- letters[1:10]
rownames(x) <- LETTERS[1:3]
robustSummary(x)
```

smooth — *Smoothing*

Description

This function smoothes a numeric vector.

Usage

```
smooth(x, cf)

coefMA(hws)

coefWMA(hws)

coefSG(hws, k = 3L)
```

Arguments

x	numeric, i.e. m/z values.
cf	matrix, a coefficient matrix generated by coefMA, coefWMA or coefSG.
hws	integer(1), half window size, the resulting window reaches from (i -hws):(i + hws).
k	integer(1), set the order of the polynomial used to calculate the coefficients.

Details

For the Savitzky-Golay-Filter the hws should be smaller than *FWHM* of the peaks (full width at half maximum; please find details in Bromba and Ziegler 1981).

In general the hws for the (weighted) moving average (coefMA/coefWMA) has to bemuch smaller than for the Savitzky-Golay-Filter to conserve the peak shape.

Value

smooth: A numeric of the same length as x.

coefMA: A matrix with coefficients for a simple moving average.

coefWMA: A matrix with coefficients for a weighted moving average.

coefSG: A matrix with *Savitzky-Golay-Filter* coefficients.

Functions

- coefMA: Simple Moving Average
  This function calculates the coefficients for a simple moving average.
- coefWMA: Weighted Moving Average
  This function calculates the coefficients for a weighted moving average with weights depending on the distance from the center calculated as 1/2^abs(-hws:hws) with the sum of all weigths normalized to 1.
- coefSG: Savitzky-Golay-Filter
  This function calculates the Savitzky-Golay-Coefficients. The additional argument k controls the order of the used polynomial. If k is set to zero it yield a simple moving average.

Note

The hws depends on the used method ((weighted) moving average/Savitzky-Golay).

Author(s)

Sebastian Gibb, Sigurdur Smarason (weighted moving average)

References

A. Savitzky and M. J. Golay. 1964. Smoothing and differentiation of data by simplified least squares procedures. Analytical chemistry, 36(8), 1627-1639.

M. U. Bromba and H. Ziegler. 1981. Application hints for Savitzky-Golay digital smoothing filters. Analytical Chemistry, 53(11), 1583-1586.

Implementation based on: Steinier, J., Termonia, Y., & Deltour, J. (1972). Comments on Smoothing and differentiation of data by simplified least square procedure. Analytical Chemistry, 44(11), 1906-1909.

See Also

Other noise estimation and smoothing functions: noise()

Examples

```
x <- c(1:10, 9:1)
plot(x, type = "b", pch = 20)
cf <- list(MovingAverage = coefMA(2),
           WeightedMovingAverage = coefWMA(2),
           SavitzkyGolay = coefSG(2))
for (i in seq_along(cf)) {
    lines(smooth(x, cf[[i]]), col = i + 1, pch = 20, type = "b")
}
legend("bottom", legend = c("x", names(cf)), pch = 20,
       col = seq_len(length(cf) + 1))
```

---

validPeaksMatrix *Validation functions*

Description

These functions are used to validate input arguments. In general they are just wrapper around their corresponding is* function with an error message.

Usage

```
validPeaksMatrix(x)
```

Arguments

x	object to test.

Details

validPeaksMatrix: see isPeaksMatrix.

Value

logical(1), TRUE if validation are successful otherwise an error is thrown.

Author(s)

Sebastian Gibb

See Also

Other helper functions for developers: between(), isPeaksMatrix(), rbindFill(), vapply1c()

Examples

```
try(validPeaksMatrix(1:2))
validPeaksMatrix(cbind(mz = 1:2, intensity = 1:2))
```

valleys *Find Peak Valleys*

Description

This function finds the valleys around peaks.

Usage

```
valleys(x, p)
```

Arguments

x	numeric, e.g. intensity values.
p	integer, indices of identified peaks/local maxima.

Value

A matrix with three columns representing the index of the left valley, the peak centroid, and the right valley.

Note

The detection of the valleys is based on localMaxima. It returns the *first* occurence of a local maximum (in this specific case the minimum). For plateaus, e.g. c(0,0,0,1:3,2:1,0) this results in a wrongly reported left valley index of 1 (instead of 3, see the example section as well). In real data this should not be a real problem. x[x == min(x)] <-Inf could be used before running valleys to circumvent this specific problem but it is not really tested and could cause different problems.

Author(s)

Sebastian Gibb

See Also

Other extreme value functions: .peakRegionMask(), localMaxima(), refineCentroids()

Examples

```
ints <- c(5, 8, 12, 7, 4, 9, 15, 16, 11, 8, 3, 2, 3, 2, 9, 12, 14, 13, 8, 3)
mzs <- seq_along(ints)
peaks <- which(localMaxima(ints, hws = 3))
cols <- seq_along(peaks) + 1

plot(mzs, ints, type = "h", ylim = c(0, 16))
points(mzs[peaks], ints[peaks], col = cols, pch = 20)

v <- valleys(ints, peaks)
segments(mzs[v[, "left"]], 0, mzs[v[, "right"]], col = cols, lwd = 2)

## Known limitations for plateaus
y <- c(0, 0, 0, 0, 0, 1:5, 4:1, 0)
```

```
valleys(y, 10L) # left should be 5 here but is 1

## a possible workaround that may cause other problems
y[min(y) == y] <- Inf
valleys(y, 10L)
```

---

vapply1c *vapply wrappers*

Description

These functions are short wrappers around typical `vapply` calls for easier development.

Usage

```
vapply1c(X, FUN, ..., USE.NAMES = FALSE)

vapply1d(X, FUN, ..., USE.NAMES = FALSE)

vapply1l(X, FUN, ..., USE.NAMES = FALSE)
```

Arguments

`X`	a vector (atomic or `list`).
`FUN`	the `function` to be applied to each element of `X`.
`...`	optional arguments to `FUN`.
`USE.NAMES`	`logical`, should the return value be named.

Value

`vapply1c` returns a vector of `characters` of length `X`.

`vapply1d` returns a vector of `doubles` of length `X`.

`vapply1l` returns a vector of `logicals` of length `X`.

Author(s)

Sebastian Gibb

See Also

Other helper functions for developers: `between()`, `isPeaksMatrix()`, `rbindFill()`, `validPeaksMatrix()`

Examples

```
l <- list(a=1:3, b=4:6)
vapply1d(l, sum)
```

Index

∗ **Quantitative feature aggregation**
aggregate_by_vector, 2
medianPolish, 17
robustSummary, 24
∗ **coerce functions**
coerce, 8
∗ **extreme value functions**
localMaxima, 16
refineCentroids, 21
valleys, 27
∗ **grouping/matching functions**
bin, 4
closest, 5
∗ **helper functions for developers**
between, 3
isPeaksMatrix, 15
rbindFill, 20
validPeaksMatrix, 26
vapply1c, 28
∗ **helper functions for users**
ppm, 20
∗ **noise estimation and smoothing functions**
noise, 17
smooth, 24
∗ **similarity functions**
dotproduct, 9
.peakRegionMask, *16*, *22*, *27*
%between% *(between)*, 3
%in%, *5–7*

aggregate_by_vector, 2, *17*, *24*
asInteger *(coerce)*, 8

base::colMeans(), *2*
base::colSums(), *2*
between, 3, *15*, *21*, *26*, *28*
bin, 4, *7*

C, *14*
closest, *5*, 5
coefMA *(smooth)*, 24
coefSG *(smooth)*, 24
coefWMA *(smooth)*, 24
coerce, 8
common *(closest)*, 5

dotproduct, 9

findInterval, *6*

group, 10

i2index, 11
impute::impute.knn(), *13*
impute_bpca *(impute_matrix)*, 12
impute_knn *(impute_matrix)*, 12
impute_matrix, 12
impute_min *(impute_matrix)*, 12
impute_mixed *(impute_matrix)*, 12
impute_mle *(impute_matrix)*, 12
impute_neighbour_average *(impute_matrix)*, 12
impute_with *(impute_matrix)*, 12
impute_zero *(impute_matrix)*, 12
imputeLCMD::impute.MinDet(), *13*
imputeLCMD::impute.MinProb(), *14*
imputeLCMD::impute.QRILC(), *13*
imputeMethods *(impute_matrix)*, 12
isPeaksMatrix, *4*, 15, *21*, *26*, *28*

join *(closest)*, 5

localMaxima, 16, *22*, *27*

MASS::rlm(), *2*, *24*
match(), *5–7*
matrixStats::colMedians(), *2*
medianPolish, *3*, 17, *24*
medianPolish(), *2*

noise, 17, *26*
norm::imp.norm(), *13*
normalize_matrix *(normalizeMethods)*, 18
normalizeMethods, 18

pcaMethods::pca(), *13*
ppm, 20
preprocessCore::normalize.quantiles(), *19*

preprocessCore::normalize.quantiles.robust(), *19*

rbindFill, *4*, *15*, 20, *26*, *28*
refineCentroids, *16*, 21, *27*
rla, 22
robustSummary, *3*, *17*, 24
robustSummary(), *2*
rowRla *(*rla*)*, 22

scale(), *19*
smooth, *18*, 24
stats::mad(), *18*
stats::medpolish(), *2*, *17*
stats::supsmu(), *18*

validPeaksMatrix, *4*, *15*, *21*, 26, *28*
valleys, *16*, *22*, 27
vapply1c, *4*, *15*, *21*, *26*, 28
vapply1d *(*vapply1c*)*, 28
vapply1l *(*vapply1c*)*, 28
vsn::vsn2(), *19*