Package ‘MsCoreUtils’

April 13, 2024

Title Core Utils for Mass Spectrometry Data
Version 1.15.6
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, baseline estimation), quantitative aggregation functions (median polish, robust summarisation, ...), missing data imputation, data normalisation (quantiles, vsn, ...), 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, clue
Suggests testthat, knitr, BiocStyle, rmarkdown, roxygen2, imputeLCMD, impute, norm, pcaMethods, vsn, Matrix, preprocessCore, missForest
Enhances HDF5Array
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 7.3.1
git_url https://git.bioconductor.org/packages/MsCoreUtils
git_branch devel
git_last_commit d167c3e
git_last_commit_date 2024-04-02
Repository  Bioconductor 3.19
Date/Publication  2024-04-12

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>),
        Philippine Louail [aut] (<https://orcid.org/0009-0007-5429-6846>),
        Adriaan Sticker [ctb],
        Sigurdur Smarason [ctb],
        Thomas Naake [ctb],
        Josep Maria Badia Aparicio [ctb]
        (<https://orcid.org/0000-0002-5704-1124>),
        Michael Witting [ctb] (<https://orcid.org/0000-0002-1462-4426>),
        Samuel Wieczorek [ctb],
        Roger Gine Bertomeu [ctb] (<https://orcid.org/0000-0003-0288-9619>),
        Mar Garcia-Aloy [ctb] (<https://orcid.org/0000-0002-1330-6610>)

Maintainer  RforMassSpectrometry Package Maintainer <maintainer@rformassspectrometry.org>

Contents

.peakRegionMask .......................................................... 3
aggregate ................................................................. 4
between ...................................................................... 6
bin ............................................................................ 7
breaks_ppm ................................................................. 9
closest ..................................................................... 10
coerce ..................................................................... 14
colCounts ................................................................. 14
distance ................................................................. 15
entropy ................................................................. 17
estimateBaseline ....................................................... 18
force_sorted ............................................................ 22
gnps .................................................................... 23
group ..................................................................... 25
i2index ................................................................. 26
impute_matrix .......................................................... 27
isPeaksMatrix .......................................................... 32
localMaxima ............................................................ 33
maxi ..................................................................... 34
medianPolish ........................................................... 35
noise ................................................................. 35
normalizeMethods ....................................................... 36
ppm ................................................................. 38
rbindFill .............................................................. 38
refineCentroids ....................................................... 39
rla ................................................................. 40
.peakRegionMask

Description

This function finds the mz region spanning by a peak. It creates an 0/1 matrix used for multiplications in other functions.

Usage

.peakRegionMask(x, p, k = 30L)

Arguments

- **x**: numeric, e.g. intensity values.
- **p**: integer, indices of identified peaks/local maxima.
- **k**: integer(1): maximum number of values left and right of the peak that should be looked for valleys.

Value

A matrix with a column for each peak in p and 2 * k + 1 rows where the middle row k + 1 is the peak centroid. If the values is 1 the index belongs to the peak region.

Author(s)

Sebastian Gibb

See Also

Other extreme value functions: localMaxima(), refineCentroids(), valleys()

Examples

```r
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 = 3L))

m <- MsCoreUtils:::peakRegionMask(ints, peaks, k = 5L)
```
aggregate quantitative features

Description

These functions take a matrix of quantitative features \( x \) and aggregate the features (rows) according to either a vector (or factor) \( \text{INDEX} \) or an adjacency matrix \( \text{MAT} \). The aggregation method is defined by function \( \text{FUN} \).

Adjacency matrices are an elegant way to explicitly encode for shared peptides (see example below) during aggregation.

Usage

\[
\begin{align*}
colMeansMat & (x, \text{MAT}, \text{na.rm} = \text{FALSE}) \\
colSumsMat & (x, \text{MAT}, \text{na.rm} = \text{FALSE}) \\
aggregate\_by\_matrix & (x, \text{MAT}, \text{FUN}, \ldots) \\
aggregate\_by\_vector & (x, \text{INDEX}, \text{FUN}, \ldots)
\end{align*}
\]

Arguments

- \( x \): A matrix of mode numeric or an \( \text{HDF5Matrix} \) object of type numeric.
- \( \text{MAT} \): An adjacency matrix that defines peptide-protein relations with \( \text{nrow(MAT)} == \text{nrow(x)} \): a non-missing/non-null value at position \( (i,j) \) indicates that peptide \( i \) belong to protein \( j \). This matrix is typically binary but can also contain weighted relations.
- \( \text{na.rm} \): A logical(1) indicating whether the missing values (including \( \text{NaN} \)) should be omitted from the calculations or not. Defaults to \( \text{FALSE} \).
- \( \text{FUN} \): A function to be applied to the subsets of \( x \).
- \( \ldots \): Additional arguments passed to \( \text{FUN} \).
- \( \text{INDEX} \): A vector or factor of length \( \text{nrow(x)} \).

Value

- \( \text{aggregate\_by\_matrix()} \) returns a matrix (or \text{Matrix}) of dimensions \( \text{ncol(MAT)} \) and \( \text{ncol(x)} \), with dimensions equal to \( \text{colnames(x)}\) and \( \text{rownames(MAT)} \).
- \( \text{aggregate\_by\_vector()} \) returns a new matrix (if \( x \) is a matrix) or \( \text{HDF5Matrix} \) (if \( x \) is an \( \text{HDF5Matrix} \)) of dimensions \( \text{length(INDEX)} \) and \( \text{ncol(x)} \), with dimensions equal to \( \text{colnames(x)}\) and \( \text{INDEX} \).
Vector-based aggregation functions

When aggregating with a vector/factor, user-defined functions must return a vector of length equal to ncol(x) for each level in INDEX. Examples thereof are:

- `medianPolish()` to fit 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.

Matrix-based aggregation functions

When aggregating with an adjacency matrix, user-defined functions must return a new matrix. Examples thereof are:

- `colSumsMat(x, MAT)` aggregates by the summing the peptide intensities for each protein. Shared peptides are re-used multiple times.
- `colMeansMat(x, MAT)` aggregation by the calculating the mean of peptide intensities. Shared peptides are re-used multiple times.

Handling missing values

By default, missing values in the quantitative data will propagate to the aggregated data. You can provide `na.rm = TRUE` to most functions listed above to ignore missing values, except for `robustSummary()` where you should supply `na.action = na.omit` (see `?MASS::rlm`).

Author(s)

Laurent Gatto and Samuel Wieczorek (aggregation from an adjacency matrix).

See Also

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

Examples

```r
x <- matrix(c(10.39, 17.16, 14.10, 12.85, 10.63, 7.52, 3.91,
              11.13, 16.53, 14.17, 11.94, 11.51, 7.69, 3.97,
              11.93, 15.37, 14.24, 11.21, 12.29, 9.00, 3.83,
              12.90, 14.37, 14.16, 10.12, 13.33, 9.75, 3.81),
             nrow = 7,
             dimnames = list(paste0("Pep", 1:7), paste0("Sample", 1:4)))
x

# -------------------------
# Aggregation by vector
# -------------------------
```
(k <- paste0("Prot", 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)

## -------------------------
## Aggregation by matrix
## -------------------------

adj <- matrix(c(1, 0, 0, 1, 1, 1, 0, 0,
                 1, 0, 1, 0, 0, 1, 0, 0,
                 1, 0, 0, 0, 1),
               nrow = 7,
               dimnames = list(paste0("Pep", 1:7),
                               paste0("Prot", c("B", "E", "X"))))

adj

## Peptide 4 is shared by 2 proteins (has a rowSums of 2),
## namely proteins B and E
rowSums(adj)

aggregate_by_matrix(x, adj, colSumsMat)
aggregate_by_matrix(x, adj, colMeansMat)

## ---------------
## Missing values
## ---------------

x <- matrix(c(NA, 2:6), ncol = 2,
            dimnames = list(paste0("Pep", 1:3),
                            c("S1", "S2")))

x

## simply use na.rm = TRUE to ignore missing values
## during the aggregation

(k <- LETTERS[c(1, 1, 2)])
aggregate_by_vector(x, k, colSums)
aggregate_by_vector(x, k, colSums, na.rm = TRUE)

(adj <- matrix(c(1, 0, 0, 0, 0, 1), ncol = 2,
               dimnames = list(paste0("Pep", 1:3),
                               c("A", "B"))))
aggregate_by_matrix(x, adj, colSumsMat, na.rm = FALSE)
aggregate_by_matrix(x, adj, colSumsMat, na.rm = TRUE)
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(), which.first()

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

```r
bin(
   x,
   y,
   size = 1,
   breaks = seq(floor(min(y)), ceiling(max(y)), by = size),
   FUN = max,
   returnMids = TRUE,
   .check = TRUE
)
```

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. `y` **must** be increasingly sorted, or else an error will be thrown.
- **size**: numeric(1) with the size of a bin.
- **breaks**: numeric defining the breaks (bins). See `breaks_ppm()` to define breaks with increasing size (depending on ppm).
- **FUN**: function to be used to aggregate values of `x` falling into the bins defined by `breaks`. `FUN` is expected to return a numeric(1).
- **returnMids**: logical(1) whether the midpoints for the breaks should be returned in addition to the binned (aggregated) values of `x`. Setting `returnMids = FALSE` might be useful if the breaks are defined before hand and binning needs to be performed on a large set of values (i.e. within a loop for multiple pairs of `x` and `y` values).
- **.check**: logical(1) whether to check that `y` is an ordered vector. Setting `.check = FALSE` will improve performance, provided you are sure that `y` is always ordered.

Value

Depending on the value of `returnMids`:

- **returnMids = TRUE** (the default): returns a list with elements `x` (aggregated values of `x`) and `mids` (the bin mid points).
- **returnMids = FALSE**: returns a numeric with just the binned values for `x`.

Author(s)

Johannes Rainer, Sebastian Gibb

See Also

Other grouping/matching functions: `closest()`, `gnps()`
## Define example intensities and m/z values

```r
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

```r
bin(ints, mz, size = 2)
```

## Repeat but summing up intensities instead of taking the max

```r
bin(ints, mz, size = 2, FUN = sum)
```

## Get only the binned values without the bin mid points.

```r
bin(ints, mz, size = 2, returnMids = FALSE)
```

---

**breaks_ppm**  
Sequence with increasing difference between elements

### Description

`breaks_ppm` creates a sequence of numbers with increasing differences between them. Parameter `ppm` defines the amount by which the difference between values increases. The value for an element `i+1` is calculated by adding `size` to the value of element `i` and in addition also the `ppm(a, ppm)`, where `a` is the value of the element `i` plus `size`. This iterative calculation is stopped once the value of an element is larger than `to`. The last value in the result vector will thus not be equal to `to` (which is in contrast to the base `seq()` function) but slightly higher.

A typical use case of this function would be to calculate breaks for the binning of m/z values of mass spectra. This function allows to create m/z-relative bin sizes which better represents measurement errors observed on certain mass spectrometry instruments.

### Usage

```r
breaks_ppm(from = 1, to = 1, by = 1, ppm = 0)
```

### Arguments

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>from</td>
<td>numeric(1) with the value from which the sequence should start.</td>
</tr>
<tr>
<td>to</td>
<td>numeric(1) defining the upper bound of the sequence. Note that the last value of the result will not be equal to <code>to</code> but equal to the first number in the sequence which is larger than this value.</td>
</tr>
<tr>
<td>by</td>
<td>numeric(1) defining the constant part of the difference by which numbers should increase.</td>
</tr>
<tr>
<td>ppm</td>
<td>numeric(1) defining the variable part of the difference by which numbers should increase (expressed in parts-per-million of the values).</td>
</tr>
</tbody>
</table>

### Value

numeric with the sequence of values with increasing differences. The returned values include `from` and `to`. 

---
**Author(s)**

Johannes Rainer

**Examples**

```r
res <- breaks_ppm(20, 50, by = 1, ppm = 50)
res
## difference between the values increases (by ppm)
diff(res)
```

---

### 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**

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

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

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

x numeric, the values to be matched. In contrast to \texttt{match()} \(x\) has to be sorted in increasing order and must not contain any NA.

table numeric, the values to be matched against. In contrast to \texttt{match()} \(\text{table}\) has to be sorted in increasing order and must not contain any NA.

tolerance numeric, accepted tolerance. Could be of length one or the same length as \(x\).

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. Has to be one of \texttt{c("keep", "closest", "remove")}. No abbreviations allowed.

nomatch integer(1), if the difference between the value in \(x\) and \(\text{table}\) is larger than tolerance \(\text{nomatch}\) is returned.

.check logical(1) turn off checks for increasingly sorted \(x\) and \(y\). This should just be done if it is ensured by other methods that \(x\) and \(y\) are sorted, see also \texttt{closest()}.

y numeric, the values to be joined. Should be sorted.

type character(1), defines how \(x\) and \(y\) should be joined. See details for \texttt{join}.

... ignored.

Details

For \texttt{closest/common} the tolerance argument could be set to 0 to get the same results as for \texttt{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 \(\text{table}\) nor the \(\text{table}\) to \(x\) matching.

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

If a single element in \(x\) matches multiple elements in \(\text{table}\) the \texttt{closest} is returned for \texttt{duplicates="keep"} or \texttt{duplicates="closest"} (keeping multiple matches isn’t possible in this case because the return value should be of the same length as \(x\)). If the differences between \(x\) and the corresponding matches in \(\text{table}\) are identical the lower index (the smaller element in \(\text{table}\)) is returned. There is one exception: if the lower index is already returned for another \(x\) with a smaller difference to this index the higher one is returned for \texttt{duplicates = "closer"} (but only if there is no other \(x\) that is closer to the higher one). For \texttt{duplicates="remove"} all multiple matches are returned as \texttt{nomatch} as above.
.checks = TRUE tests among other input validation checks for increasingly sorted \( x \) and table arguments that are mandatory assumptions for the closest algorithm. These checks require to loop through both vectors and compare each element against its precursor. Depending on the length and distribution of \( x \) and table these checks take equal/more time than the whole closest algorithm. If it is ensured by other methods that both arguments \( x \) and table are sorted the tests could be skipped by \( .\check{\text{e}}\text{ck} = \text{FALSE} \). In the case that \( .\check{\text{e}}\text{ck} = \text{FALSE} \) is used and one of \( x \) and table is not sorted (or decreasingly sorted) the output would be incorrect in the best case and result in infinity loop in the average and worst case.

\textbf{join: }joins two numeric vectors by mapping values in \( x \) with values in \( y \) and \textit{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: \texttt{type = "left"} : values in \( x \) will be mapped to values in \( y \), elements in \( y \) not matching any value in \( x \) will be discarded. \texttt{type = "right"} : same as \texttt{type = "left"} but for \( y \). \texttt{type = "outer"} : return matches for all values in \( x \) and in \( y \). \texttt{type = "inner"} : report only indices of values that could be mapped.

\textbf{Value}

\texttt{closest} returns an \texttt{integer} vector of the same length as \( x \) giving the closest position in table of the first match or \texttt{nomatch} if there is no match.

\texttt{common} returns a \texttt{logical} vector of length \( x \) that is \texttt{TRUE} if the element in \( x \) was found in \( table \). It is similar to \texttt{\%in\%}.

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

\textbf{Note}

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

\textbf{Author(s)}

Sebastian Gibb, Johannes Rainer

\textbf{See Also}

\texttt{match()}
\texttt{\%in\%}

Other grouping/matching functions: \texttt{\textit{bin()}}, \texttt{\textit{gnps()}}

\textbf{Examples}

\begin{verbatim}
## Define two vectors to match
x <- c(1, 3, 5)
y <- 1:10

## Compare match and closest
match(x, y)
closest(x, y)
\end{verbatim}
## 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, ppm = 20)

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

Coeerce functions

Description

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

Usage

```r
asInteger(x)
```

Arguments

- `x` input argument.

Author(s)

Johannes Rainer

Examples

```r
## Convert numeric to integer
asInteger(3.4)
asInteger(3)
```

colCounts

Counts the number of features

Description

Returns the number of non-NA features in a features by sample matrix.

Usage

```r
colCounts(x, ...)
```

Arguments

- `x` A matrix of mode numeric.
- `...` Currently ignored.
Value
A numeric vector of length identical to ncol(x).

Author(s)
Laurent Gatto

See Also
Other Quantitative feature aggregation: aggregate(), medianPolish(), robustSummary()

Examples
m <- matrix(c(1, NA, 2, 3, NA, NA, 4, 5, 6),
nrow = 3)
colCounts(m)
m <- matrix(rnorm(30), nrow = 3)
colCounts(m)

distance Spectra Distance/Similarity Measurements

Description
These functions provide different normalized similarity/distance measurements.

Usage
ndotproduct(x, y, m = 0L, n = 0.5, na.rm = TRUE, ...)
dotproduct(x, y, m = 0L, n = 0.5, na.rm = TRUE, ...)
neuclidean(x, y, m = 0L, n = 0.5, na.rm = TRUE, ...)
navdist(x, y, m = 0L, n = 0.5, na.rm = TRUE, ...)
spectraangle(x, y, m = 0L, n = 0.5, na.rm = TRUE, ...)

Arguments
x matrix, two-columns e.g. m/z, intensity
y matrix, two-columns e.g. m/z, intensity
m numeric, weighting for the first column of x and y (e.g. "mz"), default: 0 means
don’t weight by the first column. For more details see the ndotproduct details
section.
distance

n numeric, weighting for the second column of x and y (e.g. "intensity"), default: 0.5 means effectively using sqrt(x[,2]) and sqrt(y[,2]). For more details see the ndotproduct details section.

na.rm logical(1), should NA be removed prior to calculation (default TRUE).

... ignored.

Details

All functions that calculate normalized similarity/distance measurements are prefixed with a n.

ndotproduct: the normalized dot product is described in Stein and Scott 1994 as: \(NDP = \frac{\sum(W_i \cdot W_2^2)}{\sum(W_i^2 \cdot W_2^2)}\); where \(W_i = x^m \cdot y^n\), where x and y are the m/z and intensity values, respectively. Please note also that \(NDP = NCos^2\); where NCos is the cosine value (i.e. the orthodox normalized dot product) of the intensity vectors as described in Yilmaz et al. 2017. Stein and Scott 1994 empirically determined the optimal exponents as \(m = 3\) and \(n = 0.6\) by analyzing ca. 12000 EI-MS data of 8000 organic compounds in the NIST Mass Spectral Library. MassBank (Horai et al. 2010) uses \(m = 2\) and \(n = 0.5\) for small compounds. In general with increasing values for \(m\), high m/z values will be taken more into account for similarity calculation. Especially when working with small molecules, a value \(m > 0\) 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. Increasing \(n\) will result in a higher importance of the intensity values. Most commonly \(m = 0\) and \(n = 0.5\) are used.

neuclidean: the normalized euclidean distance is described in Stein and Scott 1994 as: \(NED = (1 + \frac{\sum(|W_i - W_2|^2)}{\sum(|W_2|^2)})^{-1}\); where \(W_i = x^m \cdot y^n\), where x and y are the m/z and intensity values, respectively. See the details section about ndotproduct for an explanation how to set \(m\) and \(n\).

navdist: the normalized absolute values distance is described in Stein and Scott 1994 as: \(NED = (1 + \frac{\sum(|W_i - W_2|)}{\sum(|W_2|)})^{-1}\); where \(W_i = x^m \cdot y^n\), where x and y are the m/z and intensity values, respectively. See the details section about ndotproduct for an explanation how to set \(m\) and \(n\).

nspectraangle: the normalized spectra angle is described in Toprak et al 2014 as: \(NSA = 1 - \frac{2 \cdot \cos^{-1}(W_1 \cdot W_2)}{\pi}\); where \(W_i = x^m \cdot y^n\), where x and y are the m/z and intensity values, respectively. The weighting was not originally proposed by Toprak et al. 2014. See the details section about ndotproduct for an explanation how to set \(m\) and \(n\).

Value
double(1) value between 0:1, where 0 is completely different and 1 identically.

Note

These methods are implemented as described in Stein and Scott 1994 (navdist, ndotproduct, neuclidean) and Toprak et al. 2014 (nspectraangle) but because there is no reference implementation available we are unable to guarantee that the results are identical. Note that the Stein and Scott 1994 normalized dot product method (and by extension ndotproduct) corresponds to the square of the orthodox normalized dot product (or cosine distance) used also commonly as spectrum similarity measure (Yilmaz et al. 2017). Please see also the corresponding discussion at the github pull request linked below. If you find any problems or reference implementation please open an issue at https://github.com/rformassspectrometry/MsCoreUtils/issues.
Author(s)

navdist, neoclidean, nspectraangle: Sebastian Gibb
ndotproduct: Sebastian Gibb and Thomas Naake, <thomasnaake@googlemail.com>

References


Pull Request for these distance/similarity measurements: https://github.com/rformassspectrometry/MsCoreUtils/pull/33

See Also

Other distance/similarity functions: gnps()

Examples

```r
x <- matrix(c(1:5, 1:5), ncol = 2, dimnames = list(c(), c("mz", "intensity")))
y <- matrix(c(1:5, 5:1), ncol = 2, dimnames = list(c(), c("mz", "intensity")))

ndotproduct(x, y)
ndotproduct(x, y, m = 2, n = 0.5)
ndotproduct(x, y, m = 3, n = 0.6)

neuclidean(x, y)
navdist(x, y)
nspectraangle(x, y)
```

<table>
<thead>
<tr>
<th>entropy</th>
<th>Spectral entropy</th>
</tr>
</thead>
</table>
estimateBaseline

Description

These functions allow to calculate entropy measurements of an MS/MS spectrum based on the metrics suggested by Li et al. (https://doi.org/10.1038/s41592-021-01331-z). Spectral entropy and normalized entropy are used to measure the complexity of an spectra. MassBank of North America (MoNA) defines spectra entropy as the intensity weighted spectral peak number (https://mona.fiehnlab.ucdavis.edu/documentation/entropy). Additionally it is suggested to consider spectra with a normalized entropy larger than 0.8, or a spectral entropy larger than 3 as low-quality spectra.

Usage

entropy(x)
nentropy(x)

Arguments

x numeric, intensities of the fragment ions.

Value

numeric: (normalized) entropy of x.

Author(s)

Mar Garcia-Aloy

Examples

spectrum <- rbind(c(41.04, 37.16), c(69.07, 66.83), c(86.1, 999.0))
entropy(spectrum[,2])
nentropy(spectrum[,2])

estimateBaseline

Estimates the Baseline of a Mass Spectrum

Description

This function estimates the the baseline of mass spectrometry data, represented by numeric vectors of masses and intensities of identical lengths.
Usage

estimateBaseline(
  x,
  y,
  method = c("SNIP", "TopHat", "ConvexHull", "median"),
  ...
)

estimateBaselineConvexHull(x, y)

estimateBaselineMedian(x, y, halfWindowSize = 100L)

estimateBaselineSnip(x, y, iterations = 100L, decreasing = TRUE)

estimateBaselineTopHat(x, y, halfWindowSize = 100L)

Arguments

x numeric() vector of masses.
y numeric() vector of intensities.
method character(1) specifying the estimation method. One of "SNIP" (default), "TopHat", "ConvexHull" or "median". See details below.
... Additional parameters passed to the respective functions.
halfWindowSize integer() defining the half window size. Default is 100L. The resulting window reaches from \(x[\text{cur_index} - \text{halfWindowSize}]\) to \(x[\text{cur_index} + \text{halfWindowSize}]\).
iterations integer() controlling the window size \(k\), similar to halfWindowSize in "TopHat", "median") of the algorithm. The resulting window reaches from \(x[\text{cur_index} - \text{iterations}]\) to \(x[\text{cur_index} + \text{iterations}]\).
decreasing logical(1) whether the clipping window should be decreasing, as defined in Morhac (2009). A decreasing clipping window is suggested to get a smoother baseline. For TRUE (FALSE) \(k=\text{iterations}\) is decreased (increased) by one until zero (iterations) is reached. The default setting is decreasing = TRUE.

Details

- **SNIP**: This baseline estimation is based on the Statistics-sensitive Non-linear Iterative Peak-clipping algorithm (SNIP) described in Ryan et al 1988. The algorithm based on the following equation:

  \[ y_i(k) = \min\{y_i, \frac{(y_i-k + y_i+k)}{2}\} \]

  It has two additional arguments namely an integer iterations and a logical decreasing.

- **TopHat**: This algorithm applies a moving minimum (erosion filter) and subsequently a moving maximum (dilation filter) filter on the intensity values. The implementation is based on van Herk (1996). It has an additional halfWindowSize argument determining the half size of the moving window for the TopHat filter.
• ConvexHull: The baseline estimation is based on a convex hull constructed below the spectrum.
• Median: This baseline estimation uses a moving median. It is based on \texttt{stats::runmed()}. The additional argument \texttt{halfWindowSize} corresponds to the \texttt{k} argument in \texttt{stats::runmed()} (\(k = 2 \times \text{halfWindowSize} + 1\)) and controls the half size of the moving window.

\textbf{Value}

\texttt{numeric()} with estimated baseline intensities.

\textbf{Author(s)}

Sebastian Gibb

\textbf{References}

These functions have been ported from the MALDIqaunt package.

\textbf{SNIP:}


\textbf{TopHat:}


\textbf{ConvexHull:}


\textbf{Examples}

```r
## ----------------------------
## Simulation example data
nmz <- 5000
mz <- seq(1000, length.out = nmz)
## create peaks
center <- seq(50, nmz, by = 500)
peaks <- lapply(center, function(cc)1000 * dpois(0:100, (1000 + cc) / 75))
## create baseline
intensity <- 100 * exp(-seq_len(nmz)/2000)
## add peaks to baseline
```
```r
for (i in seq(along = center)) {
  intensity[center[i]:((center[i] + 100))] <-
  intensity[center[i]:((center[i] + 100))] + peaks[i]
}
## add noise
intensity <- intensity + rnorm(nmz, mean = 0, sd = 1)

plot(mz, intensity, type = "l")

## SNIP baseline
base_SNIP <- estimateBaseline(mz, intensity, 
method = "SNIP", 
iterations = 20L)
## same as estimateBaselineSnip(mz, intensity, iterations = 20L)
lines(mz, base_SNIP, col = "red")

## TopHat baseline
base_TH25 <- estimateBaseline(mz, intensity, 
method = "TopHat", 
halfWindowSize = 25L)
## same as estimateBaselineTopHat(mz, intensity, halfWindowSize = 25L)
lines(mz, base_TH25, col = "blue")

base_TH15 <- estimateBaseline(mz, intensity, 
method = "TopHat", 
halfWindowSize = 15L)
lines(mz, base_TH15, col = "steelblue")

## Convex hull baseline
base_CH <- estimateBaseline(mz, intensity, 
method = "ConvexHull")
## same as estimateBaselineConvexHull(mz, intensity)
lines(mz, base_CH, col = "green")

## Median baseline
base_med <- estimateBaseline(mz, intensity, 
method = "median")
## same as estimateBaselineMedian(mz, intensity)
lines(mz, base_med, col = "orange")
legend("topright", lwd = 1, 
  legend = c("SNIP", "TopHat (hws = 25)", 
     "TopHat (hws = 15)", 
     "ConvexHull", "Median"),
  col = c("red", "blue", "steelblue", 
      "green", "orange"))
```
force_sorted

Forcing a numeric vector into a monotonously increasing sequence.

Description

This function performs interpolation on the non-increasing parts of a numeric input vector to ensure its values are monotonously increasing. If the values are non-increasing at the end of the vector, these values will be replaced by a sequence of numeric values, starting from the last increasing value in the input vector, and increasing by a very small value, which can be defined with parameter by.

Usage

force_sorted(x, by = .Machine$double.eps)

Arguments

- **x** numeric vector.
- **by** numeric(1) value that will determine the monotonous increase in case the values at the end of the vector are non-increasing and therefore interpolation would not be possible. Defaults to by = .Machine$double.eps which is the smallest positive floating-point number x such that 1 + x != 1.

Value

A vector with continuously increasing values.

Note

NA values will not be replaced and be returned as-is.

Examples

```r
x <- c(NA, NA, NA, 1.2, 1.1, 1.14, 1.2, 1.3, NA, 1.04, 1.4, 1.6, NA, NA)
y <- force_sorted(x)
is.unsorted(y, na.rm = TRUE)
```

```r
# Vector non increasing at the end
x <- c(1, 2, 1.5, 2)
y <- force_sorted(x, by = 0.1)
is.unsorted(y, na.rm = TRUE)
```

```
## We can see the values were not interpolated but rather replaced by the
## last increasing value `2`` and increasing by 0.1.

y
```
The join_gnps and gnps functions allow to calculate spectra similarity scores as used in GNPS. The approach matches first peaks between the two spectra directly using a user-defined ppm and/or tolerance as well as using a fixed delta m/z (considering the same ppm and tolerance) that is defined by the difference of the two spectra’ precursor m/z values. For peaks that match multiple peaks in the other spectrum only the matching peak pair with the higher value/similarity is considered in the final similarity score calculation. Note that GNPS similarity scores are calculated only if the two functions are used together.

- **join_gnps**: matches/maps peaks between spectra with the same approach as in GNPS: peaks are considered matching if a) the difference in their m/z values is smaller than defined by tolerance and ppm (this is the same as joinPeaks) and b) the difference of their m/z adjusted for the difference of the spectra’s precursor is smaller than defined by tolerance and ppm. Based on this definition, peaks in x can match up to two peaks in y hence returned peak indices might be duplicated. Note that if one of xPrecursorMz or yPrecursorMz are NA or if both are the same, the results are the same as with join(). The function returns a list of two integer vectors with the indices of the peaks matching peaks in the other spectrum or NA otherwise.

- **gnps**: calculates the GNPS similarity score on peak matrices’ previously aligned (matched) with join_gnps. For multi-mapping peaks the pair with the higher similarity are considered in the final score calculation.

**Usage**

```r
gnps(x, y, ...)

join_gnps(
  x,
  y,
  xPrecursorMz = NA_real_,
  yPrecursorMz = NA_real_,
  tolerance = 0,
  ppm = 0,
  type = "outer",
  ...
)
```

**Arguments**

- **x** for join_gnps: numeric with m/z values from a spectrum. For gnps: matrix with two columns "mz" and "intensity" containing the peaks aligned with peaks in y (with join_gnps).
for join_gnps: numeric with m/z values from a spectrum. For gnps: matrix with two columns "mz" and "intensity" containing the peaks aligned with peaks in x (with join_gnps).

...for join_gnps: optional parameters passed to the join() function. For gnps: ignored.
xPrecursorMz for join_gnps: numeric(1) with the precursor m/z of the spectrum x.
yPrecursorMz for join_gnps: numeric(1) with the precursor m/z of the spectrum y.
tolerance for join_gnps: numeric(1) defining a constant maximal accepted difference between m/z values of peaks from the two spectra to be matched/mapped.
ppm for join_gnps: numeric(1) defining a relative, m/z-dependent, maximal accepted difference between m/z values of peaks from the two spectra to be matched/mapped.
type for join_gnps: character(1) specifying the type of join that should be performed. See join() for details and options. Defaults to type = "outer".

Details
The implementation of gnps bases on the R code from the publication listed in the references.

Value
See function definition in the description section.

Author(s)
Johannes Rainer, Michael Witting, based on the code from Xing et al. (2020).

References

See Also
Other grouping/matching functions: bin(), closest()
Other distance/similarity functions: distance

Examples
## Define spectra
x <- cbind(mz = c(10, 36, 63, 91, 93), intensity = c(14, 15, 999, 650, 1))
y <- cbind(mz = c(10, 12, 50, 63, 105), intensity = c(35, 5, 16, 999, 450))

## The precursor m/z
pmz_x <- 91
pmz_y <- 105

## Plain join identifies only 2 matching peaks
join(x[, 1], y[, 1])
## join_gnps finds 4 matches

```r
join_gnps(x[, 1], y[, 1], pmz_x, pmz_y)
```

## with one of the two precursor m/z being NA, the result are the same as
## with join.

```r
join_gnps(x[, 1], y[, 1], pmz_x, yPrecursorMz = NA)
```

## Calculate GNPS similarity score:

```r
map <- join_gnps(x[, 1], y[, 1], pmz_x, pmz_y)
gnps(x[map[[1]], ], y[map[[2]], ])
```

---

### 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

```r
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, Sebastian Gibb
# i2index

## Examples

```r
## 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

```r
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
**impute_matrix**

Quantitative mass spectrometry data imputation

**Author(s)**

Johannes Rainer

**Examples**

```r
## 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"))
```

**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 produces valid results, in particular with naive imputations such as replacing missing values with 0.

**Usage**

```r
impute_matrix(x, method, FUN, ...)
imputeMethods()
impute_neighbour_average(x, k = min(x, na.rm = TRUE), MARGIN = 1L)
impute_knn(x, MARGIN = 1L, ...)
impute_mle(x, MARGIN = 2L, ...)
impute_b pca(x, MARGIN = 1L, ...)
impute_RF(x, MARGIN = 2L, ...)
impute_mixed(x, randna, mar, mmar, MARGIN = 1L, ...)
impute_min(x)
impute_MinDet(x, q = 0.01, MARGIN = 2L)
```
impute_MinProb(x, q = 0.01, sigma = 1, MARGIN = 2L)
impute_QRILC(x, sigma = 1, MARGIN = 2L)
impute_zero(x)
impute_with(x, val)
impute_fun(x, FUN, MARGIN = 1L, ...)
getImputeMargin(fun)

Arguments

- **x**: A matrix or an HDF5Matrix object to be imputed.
- **method**: character(1) defining the imputation method. See `imputeMethods()` for available ones.
- **FUN**: A user-provided function that takes a matrix as input and returns an imputed matrix of identical dimensions.
- **...**: 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.
- **MARGIN**: integer(1) defining the margin along which to apply imputation, with 1L for rows and 2L for columns. The default value will depend on the imputation method. Use `getImputeMargin(fun)` to get the default margin of imputation function `fun`. If the function doesn’t take a margin argument, NA is returned.
- **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.
- **q**: numeric(1) indicating the quantile to be used to estimate the minimum in MinDet and MinProb. Default is 0.01.
- **sigma**: numeric(1) controlling the standard deviation of the MNAR distribution in MinProb and QRILC. Default is 1.
- **val**: numeric(1) used to replace all missing values.
- **fun**: The imputation function to get the default margin from.

Value

A matrix of same class as `x` with dimensions `dim(x)`.
Types of missing values

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 DDA MS acquisition method. These missing value are expected to be randomly distributed in the data and are defined, in statistical terms, as missing at random (MAR) or missing completely at random (MCAR).

- Biologically relevant missing values resulting from the absence or the low abundance of ions (i.e. 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 hot deck methods such nearest neighbours, Bayesian missing value imputation or maximum likelihood methods when values are missing at random.

Imputing by rows or columns

We assume that the input matrix \( x \) contains features along the rows and samples along the columns, as is generally the case in omics data analysis. When performing imputation, the missing values are taken as a feature-specific property: feature \( x \) is missing because it is absent (in a sample or group), or because it was missed during acquisition (not selected during data dependent acquisition) or data processing (not identified or with an identification score below a chosen false discovery threshold). As such, imputation is by default performed at the feature level. In some cases, such as imputation by zero or a global minimum value, it doesn’t matter. In other cases, it does matter very much, such as for example when using the minimum value computed for each margin (i.e. row or column) as in the MinDet method (see below) - do we want to use the minimum of the sample or of that feature? KNN is another such example: do we consider the most similar features to impute a feature with missing values, or the most similar samples to impute all missing in a sample.

The MARGIN argument can be used to change the imputation margin from features/rows (MARGIN = 1) to samples/columns (MARGIN = 2). Different imputations will have different default values, and changing this parameter can have a major impact on imputation results and downstream results.

Imputation methods

Currently, the following imputation methods are available.

- **MLE**: Maximum likelihood-based imputation method using the EM algorithm. The `impute_mle()` function relies on `norm::imp.norm()` function. See `norm::imp.norm()` for details and additional parameters. Note that here, \( \ldots \) are passed to the `norm::em.norm()` function, rather to the actual imputation function `imp.norm`.

- **bpca**: Bayesian missing value imputation are available, as implemented in the `pcaMethods::pca()` function. See `pcaMethods::pca()` for details and additional parameters.

- **RF**: Random Forest imputation, as implemented in the `missForest::missForest()` function. See `missForest::missForest()` 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. The `impute_QRILC()` function calls `imputeLCMD::impute.QRILC()` from the `imputeLCMD` package.

• **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. The implementation is based on the `imputeLCMD::impute.MinDet()` function.

• **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 (or sample) standard deviations. Note that when estimating the standard deviation of the Gaussian distribution, only the peptides/proteins which present more than 50\% values are considered. The `impute_MinProb()` function calls `imputeLCMD::impute.MinProb()` from the `imputeLCMD` package.

• **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 names. Use `getImputeMargin()` to get the default margin for each imputation function.

**Author(s)**

Laurent Gatto
References


Examples

```r
## 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 smallest 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")

## user provided (random) imputation function
random_imp <- function(x) {
    m <- mean(x, na.rm = TRUE)
isPeaksMatrix

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(), which.first()
localMaxima

Examples

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

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

```r
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

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

Maximum MS Intensity Value

Description

maxi determines the maximum or mass spectrometry intensity values, e.g. from a spectrum or chromatogram. In contrast to the base R `max()` function this function returns `NA_real_` if all intensity values are `NA` or if `length(x)` is 0 (the base R `max` function returns `-Inf` in these cases).

Usage

maxi(x)

Arguments

x numeric with intensity values from which the maximum should be reported. Will be coerced to numeric.

Value

numeric(1) representing the maximum of values in x. Returns always a numeric (double) even if x is an integer.

Author(s)

Johannes Rainer, Sebastian Gibb

See Also

sumi()

Examples

x <- c(3.2, 34.4, 1.3, NA)
maxi(x)

## Compared to base R max:
max(x)
max(x, na.rm = TRUE)
max(numeric(), na.rm = TRUE)
maxi(numeric())

max(c(NA, NA), na.rm = TRUE)
maxi(c(NA, NA))
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 \texttt{stats::medpolish()}. 

Usage

\texttt{medianPolish(x, verbose = FALSE, \ldots)}

Arguments

\begin{itemize}
  \item \texttt{x} A matrix of mode numeric.
  \item \texttt{verbose} Default is FALSE.
  \item \texttt{\ldots} Additional arguments passed to \texttt{stats::medpolish()}. 
\end{itemize}

Value

A numeric vector of length identical to \texttt{ncol(x)}.

Author(s)

Laurent Gatto

See Also

Other Quantitative feature aggregation: \texttt{aggregate()}, \texttt{colCounts()}, \texttt{robustSummary()} 

Examples

\begin{verbatim}
x <- matrix(rnorm(30), nrow = 3)
medianPolish(x)
\end{verbatim}

noise

Noise Estimation

Description

This functions estimate the noise in the data.

Usage

\texttt{noise(x, y, method = c("MAD", "SuperSmoother"), \ldots)}
normalizeMethods

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

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

Description

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

Usage

```r
normalizeMethods()

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

Arguments

- **x** A matrix or an HDF5Matrix object to be normalised.
- **method** character(1) defining the normalisation method. See normalizeMethods() for available ones.
- ... Additional parameters passed to the inner normalisation function.
normalizeMethods

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.robust(). "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 matrix of same class as x with 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

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 (always a positive value).

Author(s)

Sebastian Gibb

Examples

ppm(c(1000, 2000), 5)
ppm(c(-300, 200), 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.
refineCentroids

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(), which.first()

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 calculation.
- **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

```r
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**

Description

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 = rep_len(1, ncol(x)), 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**

robustSummary

Examples

```r
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)
```

```r
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", "a", "b", "a", "b")
```

```r
## 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

```r
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()`, `colCounts()`, `medianPolish()`

Examples

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

These vectorised functions convert retention times from a numeric in seconds to/from a character as "mm:ss". \texttt{rt2character()} performs the numeric to character conversion while \texttt{rt2numeric()} performs the character to numeric conversion. \texttt{formatRt()} does one of the other depending on the input type.

Usage

\begin{verbatim}
rt2numeric(rt)
rt2character(rt)
formatRt(rt)
\end{verbatim}

Arguments

\begin{verbatim}
rt
\end{verbatim}

A vector of retention times of length > 1. Either a \texttt{numeric()} in seconds or a \texttt{character()} as "mm:ss" depending on the function.

Value

A reformatted retention time.

Author(s)

Laurent Gatto

Examples

\begin{verbatim}
## rt2numeric

rt2numeric("25:24")
rt2numeric(c("25:24", "25:25", "25:26"))

## rt2character

rt2character(1524)
rt2character(1)
rt2character(1:10)

## formatRt

formatRt(1524)
formatRt(1)
formatRt(1:10)
\end{verbatim}
smooth

**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 \((i - \text{hws}): (i + \text{hws})\).
- **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 be much 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 $1/2^{\text{abs}(-\text{hws}:\text{hws})}$ with the sum of all weights 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


See Also

Other noise estimation and smoothing functions: `noise()`

Examples

```r
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))
```
**sumi**

**Summing MS Intensity Values**

**Description**

`sumi` sums mass spectrometry intensity values, e.g. from a spectrum or chromatogram. In contrast to the base R `sum()` function this function returns `NA_real_` if all intensity values are NA or if `length(x)` is 0.

**Usage**

`sumi(x)`

**Arguments**

- `x` numeric with intensity values to be summed up. Will be coerced to numeric using `as.double`.

**Value**

numeric(1) representing the sum of values in `x`. Always returns a numeric (double) even if `x` is an integer.

**Author(s)**

Johannes Rainer

**See Also**

`maxi()`

**Examples**

```r
x <- c(3.2, 34.4, 1.3, NA)
sumi(x)

## Compared to base R sum:
sum(x)
sum(x, na.rm = TRUE)
sum(numeric(), na.rm = TRUE)
sumi(numeric())
sum(c(NA, NA), na.rm = TRUE)
sumi(c(NA, NA))
```
validPeaksMatrix

---

**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

```r
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()`, `which.first()`

### Examples

```r
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**

```r
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 occurrence 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**

```r
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)

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(), which.first()
which.first

Examples

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

---

which.first  \(\text{Which is the first/last TRUE value.}\)

Description

Determines the location, i.e., index of the first or last TRUE value in a logical vector.

Usage

```r
which.first(x)
which.last(x)
```

Arguments

- `x` logical, vector.

Value

integer, index of the first/last TRUE value. integer(0) if no TRUE (everything FALSE or NA) was found.

Author(s)

Sebastian Gibb

See Also

- `which.min()`
- Other helper functions for developers: `between()`, `isPeaksMatrix()`, `rbindFill()`, `validPeaksMatrix()`, `vapply1c()`

Examples

```r
l <- 2 <= 1:3
which.first(l)
which.last(l)
```
Index

∗. entropy, 17


∗ Quantitative feature aggregation
  aggregate, 4
  colCounts, 14
  medianPolish, 35
  robustSummary, 42

∗ Spectral entropy outperforms MS/MS dot product similarity for
  entropy, 17

∗ #’ @references
  entropy, 17

∗
  c(Sexpr[results=rd](tools:::Rd_expr_doi(\#1)),
    10.1038/s41592-021-01331-z)
  entropy, 17

∗ coerce functions
  coerce, 14

∗ distance/similarity functions
  distance, 15
  gnps, 23

∗ extreme value functions
  .peakRegionMask, 3
  localMaxima, 33
  refineCentroids, 39
  valleys, 48

∗ grouping/matching functions
  bin, 7
  closest, 10
  gnps, 23

∗ helper functions for developers
  between, 6
  isPeaksMatrix, 32
  rbindFill, 38
  validPeaksMatrix, 47
  vapply1c, 49
  which.first, 50

∗ helper functions for users
  entropy, 17
  ppm, 38

∗ internal
  .peakRegionMask, 3

∗ list(list(text), list(doi:10.1038/s41592-021-01331-z)
  <https://doi.org/10.1038/s41592-021-01331-z>), list(list(list(latex),
  list(list(list(https://doi.org/10.1038/s41592-021-01331-z), list(doi:10.1038,
    list(\slash{}), s41592, list(\-), 021, list(\-), 01331, list(\-), z)),
    list(list(list(https://doi.org/10.1038/s41592-021-01331-z),
      list(doi:10.1038/s41592-021-01331-z)))))
  entropy, 17

∗ noise estimation and smoothing functions
  noise, 35
  smooth, 44

∗ small-molecule compound identification.
  entropy, 17

∗
  entropy, 17
  .peakRegionMask, 3, 33, 40, 48
  %between% (between), 6
  %in%, 10–12

  aggregate, 4, 15, 35, 42
  aggregate_by_matrix (aggregate), 4
  aggregate_by_matrix(), 4
  aggregate_by_vector (aggregate), 4
  aggregate_by_vector(), 4
  asInteger (coerce), 14

  base::colMeans(), 5
base::colSums()

between, 6, 32, 39, 47, 49, 50
bin, 7, 12, 24
breaks_ppm, 9
breaks_ppm(), 8
C, 30
closest, 8, 10, 24
closest(), 11
coeffMA (smooth), 44
coeffSG (smooth), 44
colCounts, 5, 14, 35, 42
colMeansMat (aggregate), 4
colSumsMat (aggregate), 4
commom (closest), 10
distance, 15, 24
dotproduct (distance), 15
entropy, 17
estimateBaseline, 18
estimateBaselineConvexHull
  (estimateBaseline), 18
estimateBaselineMedian
  (estimateBaseline), 18
estimateBaselineSnip
  (estimateBaseline), 18
estimateBaselineTopHat
  (estimateBaseline), 18
force_sorted, 22
formatRt (rt2numeric), 43
getImputeMargin (impute_matrix), 27
gnps, 8, 12, 17, 23
group, 25
i2index, 26
impute::impute.knn(), 29
impute_bpca (impute_matrix), 27
impute_fun (impute_matrix), 27
impute_knn (impute_matrix), 27
impute_matrix, 27
impute_min (impute_matrix), 27
impute_MinDet (impute_matrix), 27
impute_MinProb (impute_matrix), 27
impute_mixed (impute_matrix), 27
impute_mle (impute_matrix), 27
impute_neighbour_average
  (impute_matrix), 27
impute_QRILC (impute_matrix), 27
impute_RF (impute_matrix), 27
impute_with (impute_matrix), 27
impute_zero (impute_matrix), 27
imputeLCMD::impute.MinProb(), 30
imputeLCMD::impute.QRILC(), 30
imputeMethods (impute_matrix), 27
isPeaksMatrix, 7, 32, 39, 47, 49, 50
join (closest), 10
join(), 23, 24
join_gnps (gnps), 23
localMaxima, 3, 33, 40, 48
MASS::rlm(), 5, 42
match(), 10–12
matrixStats::colMedians(), 5
max(), 34
 maxi, 34
 maxi(), 46
medianPolish, 5, 15, 35, 42
medianPolish(), 5
missForest::missForest(), 29
navdist (distance), 15
ndotproduct (distance), 15
ndotproduct (distance), 15
nentropy (entropy), 17
neuclidean (distance), 15
noise, 35, 45
norm::em.norm(), 29
norm::imp.norm(), 29
normalize_matrix (normalizeMethods), 36
normalizeMethods, 36
nspectraangle (distance), 15
pcaMethods::pca(), 29
ppm, 38
preprocessCore::normalize.quantiles(), 37
preprocessCore::normalize.quantiles.robust(), 37
rbindFill, 7, 32, 38, 47, 49, 50
refineCentroids, 3, 33, 39, 48
rla, 40
robustSummary, 5, 15, 35, 42
robustSummary(), 5
rowRla(rla), 40
rt2character(rt2numeric), 43
rt2numeric, 43

scale(), 37
seq(), 9
smooth, 36, 44
stats::mad(), 36
stats::medpolish(), 5, 35
stats::runmed(), 20
stats::supsmu(), 36
sum(), 46
sumi, 46
sumi(), 34

validPeaksMatrix, 7, 32, 39, 47, 49, 50
valleys, 3, 33, 40, 48
vapply1c, 7, 32, 39, 47, 49, 50
vapply1d(vapply1c), 49
vapply1l(vapply1c), 49
vsn::vsn2(), 37

which.first, 7, 32, 39, 47, 49, 50
which.last(which.first), 50
which.min(), 50