Package: Spectra (via r-universe)

June 8, 2026

Title Spectra Infrastructure for Mass Spectrometry Data

Version 1.22.2

Description The Spectra package defines an efficient infrastructure for storing and handling mass spectrometry spectra and functionality to subset, process, visualize and compare spectra data. It provides different implementations (backends) to store mass spectrometry data. These comprise backends tuned for fast data access and processing and backends for very large data sets ensuring a small memory footprint.

Depends R (>= 4.1.0), S4Vectors, BiocParallel

Imports ProtGenerics (>= 1.39.2), methods, IRanges, MsCoreUtils (>= 1.23.6), graphics, grDevices, stats, tools, utils, fs, BiocGenerics, MetaboCoreUtils, data.table

Suggests testthat, knitr (>= 1.1.0), MsDataHub, roxygen2, BiocStyle (>= 2.5.19), mzR (>= 2.19.6), rhdf5 (>= 2.32.0), rmarkdown, vdiffr (>= 1.0.0), msentropy, patrick

License Artistic-2.0

LazyData false

VignetteBuilder knitr

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

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

biocViews Infrastructure, Proteomics, MassSpectrometry, Metabolomics

Encoding UTF-8

Roxygen list(markdown=TRUE)

Collate 'hidden_aliases.R' 'AllGenerics.R' 'MsBackend-functions.R' 'MsBackend.R' 'MsBackendCached.R' 'MsBackendDataFrame-functions.R' 'MsBackendDataFrame.R' 'MsBackendHdf5Peaks-functions.R' 'MsBackendHdf5Peaks.R' 'MsBackendMemory-functions.R' 'MsBackendMemory.R' 'MsBackendMzR-functions.R' 'MsBackendMzR.R' 'Spectra-estimatePrecursorMz.R' 'Spectra-functions.R'

'Spectra.R' 'Spectra-neutralLoss.R' 'Spectra-precursorPurity.R'
'countIdentifications.R' 'fft_spectrum.R' 'functions-util.R'
'mz-delta-functions.R' 'peak-list-functions.R'
'peaks-functions.R' 'plotting-functions.R' 'zzz.R'

Config/roxygen2/version 8.0.0

Config/pak/sysreqs cmake make libuv1-dev

Repository https://bioc-release.r-universe.dev

Date/Publication 2026-06-08 13:37:21 UTC

RemoteUrl https://github.com/bioc/Spectra

RemoteRef RELEASE_3_23

RemoteSha 357d35646e96fe665f7797a62ff449b715bf4632

Contents

chunkapply *Apply a function stepwise to chunks of data*

Description

chunkapply() splits x into chunks and applies the function FUN stepwise to each of these chunks. Depending on the object it is called, this function might reduce memory demand considerably, if for example only the full data for a single chunk needs to be loaded into memory at a time (e.g., for Spectra objects with on-disk or similar backends).

Usage

```
chunkapply(x, FUN, ..., chunkSize = 1000L, chunks = factor())
```

Arguments

x	object to which FUN should be applied. Can be any object that supports split.
FUN	the function to apply to x.
...	additional parameters to FUN.
chunkSize	integer(1) defining the size of each chunk into which x should be splitted.
chunks	optional factor or length equal to length(x) defining the chunks into which x should be splitted.

Value

Depending on FUN, but in most cases a vector/result object of length equal to length(x).

Author(s)

Johannes Rainer

Examples

```
## Apply a function (`sqrt`) to each element in `x`, processed in chunks of
## size 200.
x <- rnorm(n = 1000, mean = 500)
res <- chunkapply(x, sqrt, chunkSize = 200)
length(res)
head(res)

## For such a calculation the vectorized `sqrt` would however be recommended
system.time(sqrt(x))
system.time(chunkapply(x, sqrt, chunkSize = 200))

## Simple example splitting a numeric vector into chunks of 200 and
## aggregating the values within the chunk using the `mean`. Due to the
## `unsplit` the result has the same length than the input with the mean
## value repeated.
```

```
x <- 1:1000
res <- chunkapply(x, mean, chunkSize = 200)
length(res)
head(res)
```

`combinePeaks` *Aggregating and combining mass peaks data*

Description

In addition to aggregating content of spectra variables (describe in `combineSpectra()`) it is also possible to aggregate and combine mass peaks data from individual spectra within a `Spectra`. These `combinePeaks()` function combines mass peaks **within each spectrum** with a difference in their m/z values that is smaller than the maximal acceptable difference defined by `ppm` and `tolerance`. Parameters `intensityFun` and `mzFun` allow to define functions to aggregate the intensity and m/z values for each such group of peaks. With `weighted = TRUE` (the default), the m/z value of the combined peak is calculated using an intensity-weighted mean and parameter `mzFun` is ignored. The `MsCoreUtils::group()` function is used for the grouping of mass peaks. Parameter `msLevel.` allows to define selected MS levels for which peaks should be combined. This function returns a `Spectra` with the same number of spectra than the input object, but with possibly combined peaks within each spectrum. Additional peak variables (other than `"mz"` and `"intensity"`) are dropped (i.e. their values are replaced with `NA`) for combined peaks unless they are constant across the combined peaks. See also `reduceSpectra()` for a function to select a single *representative* mass peak for each peak group.

Usage

```
## S4 method for signature 'Spectra'
combinePeaks(
  object,
  tolerance = 0,
  ppm = 20,
  intensityFun = base::mean,
  mzFun = base::mean,
  weighted = TRUE,
  msLevel. = uniqueMsLevels(object),
  ...
)
```

Arguments

`object`	A `Spectra` object.
`tolerance`	`numeric(1)` allowing to define a constant maximal accepted difference between m/z values for peaks to be grouped. Default is `tolerance = 0`.
`ppm`	`numeric(1)` defining a relative, m/z-dependent, maximal accepted difference between m/z values for peaks to be grouped. Default is `ppm = 20`.

intensityFun	Function to aggregate intensities for all peaks in each peak group into a single intensity value.
mzFun	Function to aggregate m/z values for all mass peaks within each peak group into a single m/z value. This parameter is ignored if weighted = TRUE (the default).
weighted	logical(1) whether m/z values of peaks within each peak group should be aggregated into a single m/z value using an intensity-weighted mean. Defaults to weighted = TRUE.
msLevel.	integer defining the MS level(s) of the spectra to which the function should be applied (defaults to all MS levels of object.
...	ignored.

Author(s)

Sebastian Gibb, Johannes Rainer, Laurent Gatto

See Also

- combineSpectra() for functions to combine or aggregate Spectra's spectra data.
- combinePeaksData() for the function to combine the mass peaks data.
- reduceSpectra() and similar functions to filter mass peaks data.
- Spectra for a general description of the Spectra object.

Examples

```
## Create a Spectra from mzML files and use the `MsBackendMzR` on-disk
## backend. Example mzML files are provided by the *MsDataHub* package.
sciex_file <- c(MsDataHub::X20171016_POOL_POS_1_105.134.mzML(),
                MsDataHub::X20171016_POOL_POS_3_105.134.mzML())
sciex <- Spectra(sciex_file, backend = MsBackendMzR())

## Combine mass peaks per spectrum with a difference in their m/z value
## that is smaller than 20 ppm. The intensity values of such peaks are
## combined by summing their values, while for the m/z values the median
## is reported
sciex_comb <- combinePeaks(sciex, ppm = 20,
    intensityFun = sum, mzFun = median)

## Comparing the number of mass peaks before and after aggregation
lengths(sciex) |> head()
lengths(sciex_comb) |> head()

## Plotting the first spectrum before and after aggregation
par(mfrow = c(1, 2))
plotSpectra(sciex[2L])
plotSpectra(sciex_comb[2L])

## Using `reduceSpectra()` to keep for each group of mass peaks with a
## difference in their m/z values < 20ppm the one with the highest intensity.
sciex_red <- reduceSpectra(sciex, ppm = 20)
```

```
## Comparing the number of mass peaks before and after the operation
lengths(sciex) |> head()
lengths(sciex_red) |> head()
```

`combinePeaksData` *Combine peaks with similar m/z across spectra*

Description

`combinePeaksData()` aggregates provided peak matrices into a single peak matrix. Peaks are grouped by their m/z values with the `group()` function from the `MsCoreUtils` package. In brief, all peaks in all provided spectra are first ordered by their m/z and consecutively grouped into one group if the (pairwise) difference between them is smaller than specified with parameter `tolerance` and `ppm` (see `MsCoreUtils::group()` for grouping details and examples).

The m/z and intensity values for the resulting peak matrix are calculated using the `mzFun` and `intensityFun` on the grouped m/z and intensity values.

Note that only the grouped m/z and intensity values are used in the aggregation functions (`mzFun` and `intensityFun`) but not the number of spectra.

The function supports also different strategies for peak combinations which can be specified with the `peaks` parameter:

- `peaks = "union"` (default): report all peaks from all input spectra.
- `peaks = "intersect"`: keep only peaks in the resulting peak matrix that are present in `>= minProp` proportion of input spectra. This would generate a *consensus* or *representative* spectra from a set of e.g. fragment spectra measured from the same precursor ion.

As a special case it is possible to report only peaks in the resulting matrix from peak groups that contain a peak from one of the input spectra, which can be specified with parameter `main`. Thus, if e.g. `main = 2` is specified, only (grouped) peaks that have a peak in the second input matrix are returned.

Setting `timeDomain` to `TRUE` causes grouping to be performed on the square root of the m/z values (assuming a TOF instrument was used to create the data).

Usage

```
combinePeaksData(
  x,
  intensityFun = base::mean,
  mzFun = base::mean,
  weighted = FALSE,
  tolerance = 0,
  ppm = 0,
  timeDomain = FALSE,
  peaks = c("union", "intersect"),
  main = integer(),
```

```
  minProp = 0.5,
  ...
)
```

Arguments

x	list of peak matrices.
intensityFun	function to be used to combine intensity values for matching peaks. By default the mean intensity value is returned.
mzFun	function to be used to combine m/z values for matching peaks. By default the mean m/z value is returned.
weighted	logical(1) defining whether m/z values for matching peaks should be calculated by an intensity-weighted average of the individuak m/z values. This overrides parameter mzFun.
tolerance	numeric(1) defining the (absolute) maximal accepted difference between mass peaks to group them into the same final peak.
ppm	numeric(1) defining the m/z-relative maximal accepted difference between mass peaks (expressed in parts-per-million) to group them into the same final peak.
timeDomain	logical(1) whether grouping of mass peaks is performed on the m/z values (timeDomain = FALSE) or on sqrt(mz) (timeDomain = TRUE).
peaks	character(1) specifying how peaks should be combined. Can be either "peaks = "union" (default) or peaks = "intersect". See function description for details.
main	optional integer(1) to force the resulting peak list to contain only peaks that are present in the specified input spectrum. See description for details.
minProp	numeric(1) for ‘peaks = "intersect": the minimal required proportion of input spectra (peak matrices) a mass peak has to be present to be included in the consensus peak matrix.
...	additional parameters to the mzFun and intensityFun functions.

Details

For general merging of spectra, the tolerance and/or ppm should be manually specified based on the precision of the MS instrument. Peaks from spectra with a difference in their m/z being smaller than tolerance or smaller than ppm of their m/z are grouped into the same final peak.

Some details for the combination of consecutive spectra of an LC-MS run:

The m/z values of the same ion in consecutive scans (spectra) of a LC-MS run will not be identical. Assuming that this random variation is much smaller than the resolution of the MS instrument (i.e. the difference between m/z values within each single spectrum), m/z value groups are defined across the spectra and those containing m/z values of the main spectrum are retained. Intensities and m/z values falling within each of these m/z groups are aggregated using the intensityFun and mzFun, respectively. It is highly likely that all QTOF profile data is collected with a timing circuit that collects data points with regular intervals of time that are then later converted into m/z values based on the relationship t = k * sqrt(m/z). The m/z scale is thus non-linear and the m/z scattering (which is in fact caused by small variations in the time circuit) will thus be different in the lower and upper m/z scale. m/z-intensity pairs from consecutive scans to be combined are therefore defined by default on the square root of the m/z values. With timeDomain = FALSE, the actual m/z values will be used.

Value

Peaks `matrix` with m/z and intensity values representing the aggregated values across the provided peak matrices.

Author(s)

Johannes Rainer

Examples

```
set.seed(123)
mzs <- seq(1, 20, 0.1)
ints1 <- abs(rnorm(length(mzs), 10))
ints1[11:20] <- c(15, 30, 90, 200, 500, 300, 100, 70, 40, 20) # add peak
ints2 <- abs(rnorm(length(mzs), 10))
ints2[11:20] <- c(15, 30, 60, 120, 300, 200, 90, 60, 30, 23)
ints3 <- abs(rnorm(length(mzs), 10))
ints3[11:20] <- c(13, 20, 50, 100, 200, 100, 80, 40, 30, 20)

## Create the peaks matrices
p1 <- cbind(mz = mzs + rnorm(length(mzs), sd = 0.01),
    intensity = ints1)
p2 <- cbind(mz = mzs + rnorm(length(mzs), sd = 0.01),
    intensity = ints2)
p3 <- cbind(mz = mzs + rnorm(length(mzs), sd = 0.009),
    intensity = ints3)

## Combine the spectra. With `tolerance = 0` and `ppm = 0` only peaks with
## **identical** m/z are combined. The result will be a single spectrum
## containing the *union* of mass peaks from the individual input spectra.
p <- combinePeaksData(list(p1, p2, p3))

## Plot the spectra before and after combining
par(mfrow = c(2, 1), mar = c(4.3, 4, 1, 1))
plot(p1[, 1], p1[, 2], xlim = range(mzs[5:25]), type = "h", col = "red")
points(p2[, 1], p2[, 2], type = "h", col = "green")
points(p3[, 1], p3[, 2], type = "h", col = "blue")

plot(p[, 1], p[, 2], xlim = range(mzs[5:25]), type = "h",
    col = "black")
## The peaks were not merged, because their m/z differs too much.

## Combine spectra with `tolerance = 0.05`. This will merge all triplets.
p <- combinePeaksData(list(p1, p2, p3), tolerance = 0.05)

## Plot the spectra before and after combining
par(mfrow = c(2, 1), mar = c(4.3, 4, 1, 1))
plot(p1[, 1], p1[, 2], xlim = range(mzs[5:25]), type = "h", col = "red")
points(p2[, 1], p2[, 2], type = "h", col = "green")
points(p3[, 1], p3[, 2], type = "h", col = "blue")

plot(p[, 1], p[, 2], xlim = range(mzs[5:25]), type = "h",
```

```
    col = "black")

## With `intensityFun = max` the maximal intensity per peak is reported.
p <- combinePeaksData(list(p1, p2, p3), tolerance = 0.05,
    intensityFun = max)

## Create *consensus*/representative spectrum from a set of spectra

p1 <- cbind(mz = c(12, 45, 64, 70), intensity = c(10, 20, 30, 40))
p2 <- cbind(mz = c(17, 45.1, 63.9, 70.2), intensity = c(11, 21, 31, 41))
p3 <- cbind(mz = c(12.1, 44.9, 63), intensity = c(12, 22, 32))

## No mass peaks identical thus consensus peaks are empty
combinePeaksData(list(p1, p2, p3), peaks = "intersect")

## Reducing the minProp to 0.2. The consensus spectrum will contain all
## peaks
combinePeaksData(list(p1, p2, p3), peaks = "intersect", minProp = 0.2)

## With a tolerance of 0.1 mass peaks can be matched across spectra
combinePeaksData(list(p1, p2, p3), peaks = "intersect", tolerance = 0.1)

## Report the minimal m/z and intensity
combinePeaksData(list(p1, p2, p3), peaks = "intersect", tolerance = 0.1,
    intensityFun = min, mzFun = min)
```

`compareSpectra` *Spectra similarity calculations*

Description

`compareSpectra()` compares each spectrum in `x` with each spectrum in `y` using the function provided with `FUN` (defaults to `MsCoreUtils::ndotproduct()`). If `y` is missing, each spectrum in `x` is compared with each other spectrum in `x`. The matching/mapping of peaks between the compared spectra is done with the `MAPFUN` function. The default `joinPeaks()` matches peaks of both spectra and allows to keep all peaks from the first spectrum (`type = "left"`), from the second (`type = "right"`), from both (`type = "outer"`) and to keep only matching peaks (`type = "inner"`); see `joinPeaks()` for more information and examples). The `MAPFUN` function should have parameters `x`, `y`, `xPrecursorMz` and `yPrecursorMz` as these values are passed to the function.

In addition to `joinPeaks()` also `joinPeaksGnps()` is supported for GNPS-like similarity score calculations (i.e., the *modified cosine* similarity). Note that `joinPeaksGnps()` should only be used in combination with `FUN = MsCoreUtils::gnps` (see `joinPeaksGnps()` for more information and details). Use `MAPFUN = joinPeaksNone` to disable internal peak matching/mapping if a similarity scoring function is used that performs the matching internally (such as e.g. the `msentropy_similarity()` function from the *msentropy* package).

`FUN` is supposed to be a function to compare intensities of (matched) peaks of the two spectra that are compared. The function needs to take two matrices with columns `"mz"` and `"intensity"` as input and is supposed to return a single numeric as result. In addition to the two peak matrices the

spectra's precursor m/z values are passed to the function as parameters `xPrecursorMz` (precursor m/z of the `x` peak matrix) and `yPrecursorMz` (precursor m/z of the `y` peak matrix). Finally, the parameter `matchedPeaksCount` is passed to the function. Additional parameters to functions `FUN` and `MAPFUN` can be passed with `...`. Parameters `ppm` and `tolerance` are passed to both `MAPFUN` and `FUN`.

By default, with `matchedPeaksCount = FALSE`, `compareSpectra()` returns a `matrix` with the results of `FUN` for each pairwise spectra comparison. The number of rows of this `matrix` is equal to `length(x)` and the number of columns equal to `length(y)` (i.e., the value reported in row 2 and column 3 is the result from the comparison of `x[2]` with `y[3]`).

For `matchedPeaksCount = TRUE` a 3-dimensional `array` is returned, with the first `matrix` in z-dimension (i.e., `res[, , 1L]`) being the similarity matrix and the second `matrix` (i.e., `res[, , 2L]`) the number of matched peaks for each compared peak pair.

If `SIMPLIFY = TRUE` and `matchedPeaksCount = FALSE`, the `matrix` is *simplified* to a `numeric` if length of `x` or `y` is one.

See also the vignette for additional examples, such as using spectral entropy similarity in the scoring.

Usage

```
## S4 method for signature 'Spectra,Spectra'
compareSpectra(
  x,
  y,
  MAPFUN = joinPeaks,
  tolerance = 0,
  ppm = 20,
  FUN = ndotproduct,
  ...,
  matchedPeaksCount = FALSE,
  SIMPLIFY = TRUE
)

## S4 method for signature 'Spectra,missing'
compareSpectra(
  x,
  y = NULL,
  MAPFUN = joinPeaks,
  tolerance = 0,
  ppm = 20,
  FUN = ndotproduct,
  ...,
  matchedPeaksCount = FALSE,
  SIMPLIFY = TRUE
)
```

Arguments

x	A `Spectra` object.

y — A Spectra object.

MAPFUN — function to map/match peaks between the two compared spectra. See joinPeaks() for more information and possible functions. Defaults to MAPFUN = joinPeaks.

tolerance — numeric(1) allowing to define a constant maximal accepted difference between m/z values for peaks to be matched. This parameter is directly passed to MAPFUN.

ppm — numeric(1) defining a relative, m/z-dependent, maximal accepted difference between m/z values for peaks to be matched. This parameter is directly passed to MAPFUN.

FUN — function to compare intensities of peaks between two spectra. See MsCoreUtils::distance() for directly supported similarity functions. Defaults to MsCoreUtils::ndotproduct.

... — Additional arguments passed to the internal functions.

matchedPeaksCount — logical(1) whether the number of matching peaks between the compared pairs of spectra should be returned in addition to the similarity scores. Note that with matchedPeaksCount = TRUE a 3-dimensional array is returned. See examples below for details. This requires that the spectra similarity function defined with FUN has a parameter matchedPeaksCount and that it returns, for matchedPeaksCount = TRUE a numeric(2) with the similarity score and the number of matched peaks.

SIMPLIFY — logical(1) defining whether the result matrix should be *simplified* to a numeric if possible (i.e. if either x or y is of length 1).

Details

compareSpectra() supports custom functions for peak matching and similarity calculation that can be provided with parameters MAPFUN and FUN, respectively. The parameters passed from compareSpectra() to these functions are:

- for MAPFUN: x (peak matrix of the first spectrum), y (peak matrix of the second spectrum), tolerance (the absolute acceptable tolerance for peaks to be considered matching), ppm (the m/z-relative difference), xPrecursorMz (the precursor m/z of the first spectrum), yPrecursorMz (the precursor m/z of the second spectrum).
- for FUN: x (peak matrix aligned with peaks from the second spectrum, i.e., the result from MAPFUN), y (peak matrix aligned with peaks from the first spectrum), xPrecursorMz (precursor m/z of the first spectrum), yPrecursorMz (precursor m/z of the second spectrum), tolerance, ppm, matchedPeaksCount (logical(1) whether the number of peak pairs on which the similarity was calculated should be returned in addition; if set to TRUE the function is expected to return a numeric(2) with the similarity score and the number of matched peaks).

compareSpectra(), before calculating the spectra similarity, will apply all eventually present data modification operations cached in the processing queue of the two compared Spectra. Internally, the calculations are performed in a memory-saving manner loading, for a large number of spectra, data only for a smaller *chunk* of data at a time.

Value

For matchedPeaksCount = FALSE (the default) a matrix with the similarity scores. With matchedPeaksCount = FALSE and SIMPLIFY = TRUE a numeric vector. For matchedPeaksCount = TRUE a 3-dimensional

array with the scores reported in the first matrix in z dimension (`[, , 1]`) and the number of matching peaks in the second matrix in z dimension (`[, , 2]`).

Note

The `compareSpectra(x)` function to calculate similarities between each spectrum within the **same** `Spectra` uses a memory efficient calculation which can however be considerably slower than `compareSpectra(x, x)`.

Author(s)

Sebastian Gibb, Johannes Rainer, Laurent Gatto

Examples

```
## Load a `Spectra` object with LC-MS/MS data.
fl <- MsDataHub::PestMix1_DDA.mzML()
sps_dda <- Spectra(fl)
sps_dda

## Restrict to MS2 (fragment) spectra:
sps_ms2 <- filterMsLevel(sps_dda, msLevel = 2L)

## Compare spectra: comparing spectra 2 and 3 against spectra 10:20 using
## the normalized dotproduct method.
res <- compareSpectra(sps_ms2[2:3], sps_ms2[10:20])
## first row contains comparisons of spectrum 2 with spectra 10 to 20 and
## the second row comparisons of spectrum 3 with spectra 10 to 20
res

## Setting parameter `matchedPeaksCount = TRUE` returns in addition to the
## simialrity score also the number of matching peaks between the compared
## spectra. The results are then returned as a 3-dimensional array, with the
## first matrix in z dimension (`[, , 1]`) containing the scores and the
## second matrix in z dimention (`[, , 2]`) the number of matching peaks:
res <- compareSpectra(sps_ms2[2:3], sps_ms2[10:20], matchedPeaksCount = TRUE)

## the scores
res[, , 1L]

## the number of matching peaks
res[, , 2L]

## We next calculate the pairwise similarity for the first 10 spectra
compareSpectra(sps_ms2[1:10])

## Use compareSpectra to determine the number of common (matching) peaks
## with a ppm of 10:
## type = "inner" uses a *inner join* to match peaks, i.e. keeps only
## peaks that can be mapped betwen both spectra. The provided FUN returns
## simply the number of matching peaks.
compareSpectra(sps_ms2[2:3], sps_ms2[10:20], ppm = 10, type = "inner",
```

```
    FUN = function(x, y, ...) nrow(x))

## We repeat this calculation between all pairwise combinations
## of the first 20 spectra
compareSpectra(sps_ms2[1:20], ppm = 10, type = "inner",
    FUN = function(x, y, ...) nrow(x))
```

---

concatenateSpectra *Merging, aggregating and splitting Spectra*

Description

Various functions are availabe to combine, aggregate or split data from one of more `Spectra` objects. These are:

- `c()` and `concatenateSpectra()`: combines several `Spectra` objects into a single object. The resulting `Spectra` contains all data from all individual `Spectra`, i.e. the union of all their spectra variables. Concatenation will fail if the processing queue of any of the `Spectra` objects is not empty or if different backends are used for the `Spectra` objects. In such cases it is suggested to first change the backends of all `Spectra` to the same type of backend (using the `setBackend()` function and to eventually (if needed) apply the processing queue using the `applyProcessing()` function.
- `cbind2()`: Appends multiple spectra variables from a `data.frame`, `DataFrame` or `matrix` to the `Spectra` object at once. The order of the values (rows) in `y` has to match the order of spectra in `x`. The function does not allow to replace existing spectra variables. `cbind2()` returns a `Spectra` object with the appended spectra variables. For a more controlled way of adding spectra variables, see the `joinSpectraData()` function.
- `combineSpectra()`: combines sets of spectra (defined with parameter `f`) into a single spectrum per set aggregating their MS data (i.e. their *peaks data* matrices with the *m/z* and intensity values of their mass peaks). The spectra variable values of the first spectrum per set are reported for the combined spectrum. The peak matrices of the spectra per set are combined using the function specified with parameter `FUN` which uses by default the `combinePeaksData()` function. See the documentation of `combinePeaksData()` for details on the aggregation of the peak data and the package vignette for examples. The sets of spectra can be specified with parameter `f` which is expected to be a `factor` or `vector` of length equal to the length of the `Spectra` specifying to which set a spectrum belongs to. The function returns a `Spectra` of length equal to the unique levels of `f`. The optional parameter `p` allows to define how the `Spectra` should be split for potential parallel processing. The default is `p = x$dataStorage` and hence a per storage file parallel processing is applied for `Spectra` with on disk data representations (such as the `MsBackendMzR()`). This also prevents that spectra from different data files/samples are combined (eventually use e.g. `p = x$dataOrigin` or any other spectra variables defining the originating samples for a spectrum). Before combining the peaks data, all eventual present processing steps are applied (by calling `applyProcessing()` on the `Spectra`). This function will replace the original *m/z* and intensity values of a `Spectra` hence it can not be called on a `Spectra` with a *read-only* backend. In such cases, the backend should be changed to a *writeable* backend before using the `setBackend()` function (to e.g. a `MsBackendMemory()` backend).

- `joinSpectraData()`: Individual spectra variables can be directly added with the `$<-` or `[[<-` syntax. The `joinSpectraData()` function allows to merge a `DataFrame` to the existing spectra data of a `Spectra`. This function diverges from the `merge()` method in two main ways:
  - The `by.x` and `by.y` column names must be of length 1.
  - If variable names are shared in `x` and `y`, the spectra variables of `x` are not modified. It's only the `y` variables that are appended with the suffix defined in `suffix.y`. This is to avoid modifying any core spectra variables that would lead to an invalid object.
  - Duplicated Spectra keys (i.e. `x[[by.x]]`) are not allowed. Duplicated keys in the `DataFrame` (i.e `y[[by.y]]`) throw a warning and only the last occurrence is kept. These should be explored and ideally be removed using for `QFeatures::reduceDataFrame()`, `PMS::reducePSMs()` or similar functions. For a more general function that allows to append `data.frame`, `DataFrame` and `matrix` see `cbind2()`.
- `split()`: splits the `Spectra` object based on parameter `f` into a `list` of `Spectra` objects.

Usage

```
concatenateSpectra(x, ...)

combineSpectra(
  x,
  f = x$dataStorage,
  p = x$dataStorage,
  FUN = combinePeaksData,
  ...,
  BPPARAM = bpparam()
)

joinSpectraData(x, y, by.x = "spectrumId", by.y, suffix.y = ".y")

## S4 method for signature 'Spectra'
c(x, ...)

## S4 method for signature 'Spectra,dataframeOrDataFrameOrmatrix'
cbind2(x, y, ...)

## S4 method for signature 'Spectra,ANY'
split(x, f, drop = FALSE, ...)
```

Arguments

`x`	A `Spectra` object.
`...`	Additional arguments.
`f`	For `split()`: factor defining how to split `x`. See `base::split()` for details. For `combineSpectra()`: `factor` defining the grouping of the spectra that should be combined. Defaults to `x$dataStorage`.
`p`	For `combineSpectra()`: `factor` defining how to split the input `Spectra` for parallel processing. Defaults to `x$dataStorage`, i.e., depending on the used backend, per-file parallel processing will be performed.

FUN	For `combineSpectra()`: function to combine the (peak matrices) of the spectra. Defaults to `combinePeaksData()`.
BPPARAM	Parallel setup configuration. See `processingChunkSize()` and `BiocParallel::bpparam()` for more information.
y	For `joinSpectraData()`: `DataFrame` with the spectra variables to join/add. For `cbind2()`: a `data.frame`, `DataFrame` or `matrix`. The number of rows and their order has to match the number of spectra in `x`, respectively their order.
by.x	A `character(1)` specifying the spectra variable used for merging. Default is `"spectrumId"`.
by.y	A `character(1)` specifying the column used for merging. Set to `by.x` if missing.
suffix.y	A `character(1)` specifying the suffix to be used for making the names of columns in the merged spectra variables unique. This suffix will be used to amend `names(y)`, while `spectraVariables(x)` will remain unchanged.
drop	For `split()`: not considered.

Author(s)

Sebastian Gibb, Johannes Rainer, Laurent Gatto

See Also

- `combinePeaks()` for functions to aggregate mass peaks data.
- Spectra for a general description of the `Spectra` object.

Examples

```
## Create a Spectra providing a `DataFrame` containing a MS data.

spd <- DataFrame(msLevel = c(1L, 2L), rtime = c(1.1, 1.2))
spd$mz <- list(c(100, 103.2, 104.3, 106.5), c(45.6, 120.4, 190.2))
spd$intensity <- list(c(200, 400, 34.2, 17), c(12.3, 15.2, 6.8))

s <- Spectra(spd)
s

## Create a second Spectra from mzML files and use the `MsBackendMzR`
## on-disk backend. Example mzML files are provided by the *MsDataHub*
## package.
sciex_file <- c(MsDataHub::X20171016_POOL_POS_1_105.134.mzML(),
                MsDataHub::X20171016_POOL_POS_3_105.134.mzML())
sciex <- Spectra(sciex_file, backend = MsBackendMzR())
sciex

## Subset to the first 100 spectra to reduce running time of the examples
sciex <- sciex[1:100]


##  --------  COMBINE SPECTRA  --------
```

```
## Combining the `Spectra` object `s` with the MS data from `sciex`.
## Calling directly `c(s, sciex)` would result in an error because
## both backends use a different backend. We thus have to first change
## the backends to the same backend. We change the backend of the `sciex`
## `Spectra` to a `MsBackendMemory`, the backend used by `s`.

sciex <- setBackend(sciex, MsBackendMemory())

## Combine the two `Spectra`
all <- c(s, sciex)
all

## The new `Spectra` objects contains the union of spectra variables from
## both:
spectraVariables(all)

## The spectra variables that were not present in `s`:
setdiff(spectraVariables(all), spectraVariables(s))

## The values for these were filled with missing values for spectra from
## `s`:
all$peaksCount |> head()


##  --------  AGGREGATE SPECTRA  --------

## Sets of spectra can be combined into a single, representative spectrum
## per set using `combineSpectra()`. This aggregates the peaks data (i.e.
## the spectra's m/z and intensity values) while using the values for all
## spectra variables from the first spectrum per set. Below we define the
## sets as all spectra measured in the *same second*, i.e. rounding their
## retention time to the next closer integer value.
f <- round(rtime(sciex))
head(f)

cmp <- combineSpectra(sciex, f = f)

## The length of `cmp` is now equal to the length of unique levels in `f`:
length(cmp)

## The spectra variable value from the first spectrum per set is used in
## the representative/combined spectrum:
cmp$rtime

## The peaks data was aggregated: the number of mass peaks of the first six
## spectra from the original `Spectra`:
lengths(sciex) |> head()

## and for the first aggreagated spectra:
lengths(cmp) |> head()

## The default peaks data aggregation method joins all mass peaks. See
```

```
## documentation of the `combinePeaksData()` function for more options.


##  --------  SPLITTING DATA  --------

## A `Spectra` can be split into a `list` of `Spectra` objects using the
## `split()` function defining the sets into which the `Spectra` should
## be splitted into with parameter `f`.
sciex_split <- split(sciex, f)

length(sciex_split)
sciex_split |> head()


##  --------  ADDING SPECTRA DATA  --------

## Adding new spectra variables
sciex1 <- filterDataOrigin(sciex, dataOrigin(sciex)[1])
spv <- DataFrame(spectrumId = sciex1$spectrumId[3:12], ## used for merging
                 var1 = rnorm(10),
                 var2 = sample(letters, 10))
spv

sciex2 <- joinSpectraData(sciex1, spv, by.y = "spectrumId")

spectraVariables(sciex2)
spectraData(sciex2)[1:13, c("spectrumId", "var1", "var2")]

## Append new spectra variables with cbind2()
df <- data.frame(cola = seq_len(length(sciex1)), colb = "b")
data_append <- cbind2(sciex1, df)
```

`countIdentifications` *Count the number of identifications per scan*

Description

The function takes a `Spectra` object containing identification results as input. It then counts the number of identifications each scan (or their descendants) has lead to - this is either 0 or 1 for MS2 scans, or, for MS1 scans, the number of MS2 scans originating from any MS1 peak that lead to an identification.

This function can be used to generate id-annotated total ion chromatograms, as can illustrated here.

Usage

```
countIdentifications(
  object,
  identification = "sequence",
  f = dataStorage(object),
```

```
  BPPARAM = bpparam()
)
```

Arguments

object	An instance of class Spectra that contains identification data, as defined by the sequence argument.
identification	character(1) with the name of the spectra variable that defines whether a scan lead to an identification (typically containing the idenfified peptides sequence in proteomics). The absence of identification is encode by an NA. Default is "sequence".
f	A factor defining how to split object for parallelized processing. Default is dataOrigin(x), i.e. each raw data files is processed in parallel.
BPPARAM	Parallel setup configuration. See BiocParallel::bpparam() for details.

Details

The computed number of identifications is stored in a new spectra variables named "countIdentifications". If it already exists, the function throws a message and returns the object unchanged. To force the recomputation of the "countIdentifications" variable, users should either delete or rename it.

Value

An updated Spectra() object that now contains an integer spectra variable countIdentifications with the number of identification for each scan.

Author(s)

Laurent Gatto

See Also

addProcessing() for other data analysis functions.

Examples

```
spdf <- new("DFrame", rownames = NULL, nrows = 86L,
   listData = list(
       msLevel = c(1L, 2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L,
                   2L, 2L, 2L, 2L, 2L, 2L, 2L, 1L, 2L, 2L, 2L, 2L,
                   2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L,
                   2L, 1L, 2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L,
                   2L, 2L, 2L, 1L, 2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L,
                   2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L, 1L, 2L,
                   2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L,
                   2L, 2L),
       acquisitionNum = 8975:9060,
       precScanNum = c(NA, 8956L, 8956L, 8956L, 8956L, 8956L, 8956L,
                       8956L, 8956L, 8956L, 8956L, 8956L, 8956L,
                       8956L, 8956L, 8956L, 8956L, 8956L, 8956L, NA,
```

```
                        8975L, 8975L, 8975L, 8975L, 8975L, 8975L,
                        8975L, 8975L, 8975L, 8975L, 8975L, 8975L,
                        8975L, 8975L, 8975L, 8975L, 8975L, NA, 8994L,
                        8994L, 8994L, 8994L, 8994L, 8994L, 8994L,
                        8994L, 8994L, 8994L, 8994L, 8994L, 8994L, NA,
                        9012L, 9012L, 9012L, 9012L, 9012L, 9012L,
                        9012L, 9012L, 9012L, 9012L, 9012L, 9012L,
                        9012L, 9012L, 9012L, 9012L, 9012L, 9012L, NA,
                        9026L, 9026L, 9026L, 9026L, 9026L, 9026L,
                        9026L, 9026L, 9026L, 9026L, 9026L, 9026L,
                        9026L, 9026L, 9026L),
        sequence = c(NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,
                     NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,
                     "LSEHATAPTR", NA, NA, NA, NA, NA, NA, NA,
                     "EGSDATGDGTK", NA, NA, "NEDEDSPNK", NA, NA, NA,
                     NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,
                     NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,
                     NA, NA, NA, NA, NA, NA, NA, NA, NA, "GLTLAQGGVK",
                     NA, NA, NA, NA, "STLPDADRER", NA, NA, NA, NA, NA,
                     NA, NA, NA)),
    elementType = "ANY", elementMetadata = NULL, metadata = list())

sp <- Spectra(spdf)

## We have in this data 5 MS1 and 81 MS2 scans
table(msLevel(sp))

## The acquisition number of the MS1 scans
acquisitionNum(filterMsLevel(sp, 1))

## And the number of MS2 scans with precursor ions selected
## from MS1 scans (those in the data and others)
table(precScanNum(sp))

## Count number of sequences/identifications per scan
sp <- countIdentifications(sp)

## MS2 scans either lead to an identification (5 instances) or none
## (76). Among the five MS1 scans in the experiment, 3 lead to MS2
## scans being matched to no peptides and two MS1 scans produced two
## and three PSMs respectively.
table(sp$countIdentifications, sp$msLevel)
```

`deisotopeSpectra` *Filter and subset Spectra objects*

Description

A variety of functions to filter or subset `Spectra` objects are available. These can be generally separated into two main classes: I) *classical* subset operations that immediately reduce the number

of spectra in the object and II) filters that reduce the **content** of the object without changing its length (i.e. the number of spectra). The latter can be further subdivided into functions that affect the content of the `spectraData` (i.e. the general spectrum metadata) and those that reduce the content of the object's `peaksData` (i.e. the m/z and intensity values of a spectrum's mass peaks).

A description of functions from these 3 different categories are given below in sections *Subset* `Spectra`, *Filter content of* `spectraData()` and *Filter content of* `peaksData()`, respectively.

Usage

```
deisotopeSpectra(
  x,
  substDefinition = isotopicSubstitutionMatrix("HMDB_NEUTRAL"),
  tolerance = 0,
  ppm = 20,
  charge = 1
)

reduceSpectra(x, tolerance = 0, ppm = 20)

filterPrecursorMaxIntensity(x, tolerance = 0, ppm = 20)

filterPrecursorIsotopes(
  x,
  tolerance = 0,
  ppm = 20,
  substDefinition = isotopicSubstitutionMatrix("HMDB_NEUTRAL")
)

filterPrecursorPeaks(
  object,
  tolerance = 0,
  ppm = 20,
  mz = c("==", ">="),
  msLevel. = uniqueMsLevels(object)
)

## S4 method for signature 'Spectra'
dropNaSpectraVariables(object, onlyCore = FALSE)

## S4 method for signature 'Spectra'
selectSpectraVariables(
  object,
  spectraVariables = union(spectraVariables(object), peaksVariables(object))
)

## S4 method for signature 'Spectra'
x[i, j, ..., drop = FALSE]
```

```
## S4 method for signature 'Spectra'
filterAcquisitionNum(
  object,
  n = integer(),
  dataStorage = character(),
  dataOrigin = character()
)

## S4 method for signature 'Spectra'
filterEmptySpectra(object)

## S4 method for signature 'Spectra'
filterDataOrigin(object, dataOrigin = character())

## S4 method for signature 'Spectra'
filterDataStorage(object, dataStorage = character())

## S4 method for signature 'Spectra'
filterFourierTransformArtefacts(
  object,
  halfWindowSize = 0.05,
  threshold = 0.2,
  keepIsotopes = TRUE,
  maxCharge = 5,
  isotopeTolerance = 0.005
)

## S4 method for signature 'Spectra'
filterIntensity(
  object,
  intensity = c(0, Inf),
  msLevel. = uniqueMsLevels(object),
  ...
)

## S4 method for signature 'Spectra'
filterIsolationWindow(object, mz = numeric())

## S4 method for signature 'Spectra'
filterMsLevel(object, msLevel. = integer())

## S4 method for signature 'Spectra'
filterMzRange(
  object,
  mz = numeric(),
  msLevel. = uniqueMsLevels(object),
  keep = TRUE
)
```

```
## S4 method for signature 'Spectra'
filterMzValues(
  object,
  mz = numeric(),
  tolerance = 0,
  ppm = 20,
  msLevel. = uniqueMsLevels(object),
  keep = TRUE
)

## S4 method for signature 'Spectra'
filterPolarity(object, polarity = integer())

## S4 method for signature 'Spectra'
filterPrecursorMz(object, mz = numeric())

## S4 method for signature 'Spectra'
filterPrecursorMzRange(object, mz = numeric())

## S4 method for signature 'Spectra'
filterPrecursorMzValues(object, mz = numeric(), ppm = 20, tolerance = 0)

## S4 method for signature 'Spectra'
filterPrecursorCharge(object, z = integer())

## S4 method for signature 'Spectra'
filterPrecursorScan(object, acquisitionNum = integer(), f = dataOrigin(object))

## S4 method for signature 'Spectra'
filterRt(object, rt = numeric(), msLevel. = integer())

## S4 method for signature 'Spectra'
filterRanges(
  object,
  spectraVariables = character(),
  ranges = numeric(),
  match = c("all", "any")
)

## S4 method for signature 'Spectra'
filterValues(
  object,
  spectraVariables = character(),
  values = numeric(),
  ppm = 0,
  tolerance = 0,
  match = c("all", "any")
```

)

Arguments

x	Spectra object.
substDefinition	For deisotopeSpectra(): matrix or data.frame with definitions of isotopic substitutions. Uses by default isotopic substitutions defined from all compounds in the Human Metabolome Database (HMDB). See MetaboCoreUtils::isotopologues() or MetaboCoreUtils::isotopicSubstitutionMatrix() in the *MetaboCoreUtils* for details.
tolerance	For filterMzValues() and reduceSpectra(): numeric(1) allowing to define a constant maximal accepted difference between m/z values for peaks to be matched (or grouped). For containsMz() it can also be of length equal mz to specify a different tolerance for each m/z value. For filterPrecursorMaxIntensity(): numeric(1) defining the (constant) maximal accepted difference of precursor m/z values of spectra for grouping them into *precursor groups*. For filterPrecursorIsotopes(): passed directly to the MetaboCoreUtils::isotopologues() function. For filterValues(): numeric of any length allowing to define a maximal accepted difference between user input values and the spectraVariables values. If it is not equal to the length of the value provided with parameter spectraVariables, tolerance[1] will be recycled. Default is tolerance = 0.
ppm	For filterMzValues() and reduceSpectra(): numeric(1) defining a relative, m/z-dependent, maximal accepted difference between m/z values for peaks to be matched (or grouped). For filterPrecursorMaxIntensity(): numeric(1) defining the relative maximal accepted difference of precursor m/z values of spectra for grouping them into *precursor groups*. For filterPrecursorIsotopes(): passed directly to the MetaboCoreUtils::isotopologues() function. For filterValues(): numeric of any length allowing to define a maximal accepted difference between user input values and the spectraVariables values. If it is not equal to the length of the value provided with parameter spectraVariables, ppm[1] will be recycled.
charge	For deisotopeSpectra(): expected charge of the ionized compounds. See MetaboCoreUtils::isotopologues() for details.
object	Spectra object.
mz	For filterIsolationWindow(): numeric(1) with the m/z value to filter the object. For filterPrecursorMz() and filterMzRange(): numeric(2) defining the lower and upper m/z boundary. For filterMzValues() and filterPrecursorMzValues(): numeric with the m/z values to match peaks or precursor m/z against. For filterPrecursorPeaks(): character(1) defining whether mass peaks with an m/z matching the spectrum's precursor m/z (mz = "==", the default) or mass peaks with a m/z that is equal or larger (mz = ">=") should be removed.
msLevel.	integer defining the MS level(s) of the spectra to which the function should be applied (defaults to all MS levels of object. For filterMsLevel(): the MS level to which object should be subsetted.
onlyCore	For dropNaSpectraVariables(): logical(1) whether only *core* spectra variables (i.e., coreSpectraVariables()) are evaluated for removal. For onlyCore

= TRUE any user-added spectra variables will be retained even if they contain only missing values. Defaults to onlyCore = FALSE.

spectraVariables
: For selectSpectraVariables(): character with the names of the spectra variables to which the backend should be subsetted. For filterRanges() and filterValues(): character vector specifying the column(s) from spectraData(object) on which to filter the data and that correspond to the the names of the spectra variables that should be used for the filtering.

i
: For [: integer, logical or character to subset the object.

j
: For [: not supported.

...
: Additional arguments.

drop
: For [: not considered.

n
: For filterAcquisitionNum(): integer with the acquisition numbers to filter for.

dataStorage
: For filterDataStorage(): character to define which spectra to keep. For filterAcquisitionNum(): optionally specify if filtering should occur only for spectra of selected dataStorage.

dataOrigin
: For filterDataOrigin(): character to define which spectra to keep. For filterAcquisitionNum(): optionally specify if filtering should occurr only for spectra of selected dataOrigin.

halfWindowSize
: For filterFourierTransformArtefacts(): numeric(1) defining the m/z window left and right of a peak where to remove fourier transform artefacts.

threshold
: For filterFourierTransformArtefacts(): the relative intensity (to a peak) below which peaks are considered fourier artefacts. Defaults to threshold = 0.2 hence removing peaks that have an intensity below 0.2 times the intensity of the tested peak (within the selected halfWindowSize).

keepIsotopes
: For filterFourierTransformArtefacts(): whether isotope peaks should not be removed as fourier artefacts.

maxCharge
: For filterFourierTransformArtefacts(): the maximum charge to be considered for isotopes.

isotopeTolerance
: For filterFourierTransformArtefacts(): the m/z tolerance to be used to define whether peaks might be isotopes of the current tested peak.

intensity
: For filterIntensity(): numeric of length 1 or 2 defining either the lower or the lower and upper intensity limit for the filtering, or a function that takes the intensities as input and returns a logical (same length then peaks in the spectrum) whether the peak should be retained or not. Defaults to intensity = c(0, Inf) thus only peaks with NA intensity are removed.

keep
: For filterMzValues() and filterMzRange(): logical(1) whether the matching peaks should be retained (keep = TRUE, the default) or dropped (keep = FALSE).

polarity
: for filterPolarity(): integer specifying the polarity to to subset object.

z
: For filterPrecursorCharge(): integer() with the precursor charges to be used as filter.

acquisitionNum	for filterPrecursorScan(): integer with the acquisition number of the spectra to which the object should be subsetted.
f	For filterPrecursorScan(): defining which spectra belong to the same original data file (sample): Defaults to f = dataOrigin(x).
rt	For filterRt(): numeric(2) defining the retention time range to be used to subset/filter object.
ranges	For filterRanges(): A numeric vector of paired values (upper and lower boundary) that define the ranges to filter the object. These paired values need to be in the same order as the spectraVariables parameter (see below).
match	For filterRanges() and filterValues(): character(1) defining whether the condition has to match for all provided ranges/values (match = "all"; the default), or for any of them (match = "any") for spectra to be retained.
values	for filterValues(): A numeric vector that define the values to filter the Spectra data. These values need to be in the same order as the spectraVariables parameter.

Subset Spectra

These functions affect the number of spectra in a Spectra object creating a subset of the original object without affecting its content.

- [: subsets the spectra keeping only selected elements (i). The method **always** returns a Spectra object.
- filterAcquisitionNum(): filters the object keeping only spectra matching the provided acquisition numbers (argument n). If dataOrigin or dataStorage is also provided, object is subsetted to the spectra with an acquisition number equal to n **in spectra with matching dataOrigin or dataStorage values** retaining all other spectra. Returns the filtered Spectra.
- filterDataOrigin(): filters the object retaining spectra matching the provided dataOrigin. Parameter dataOrigin has to be of type character and needs to match exactly the data origin value of the spectra to subset. Returns the filtered Spectra object (with spectra ordered according to the provided dataOrigin parameter).
- filterDataStorage(): filters the object retaining spectra stored in the specified dataStorage. Parameter dataStorage has to be of type character and needs to match exactly the data storage value of the spectra to subset. Returns the filtered Spectra object (with spectra ordered according to the provided dataStorage parameter).
- filterEmptySpectra(): removes empty spectra (i.e. spectra without peaks). Returns the filtered Spectra object (with spectra in their original order).
- filterIsolationWindow(): retains spectra that contain mz in their isolation window m/z range (i.e. with an isolationWindowLowerMz <= mz and isolationWindowUpperMz >= mz. Returns the filtered Spectra object (with spectra in their original order).
- filterMsLevel(): filters object by MS level keeping only spectra matching the MS level specified with argument msLevel. Returns the filtered Spectra (with spectra in their original order).
- filterPolarity(): filters the object keeping only spectra matching the provided polarity. Returns the filtered Spectra (with spectra in their original order).

- `filterPrecursorCharge()`: retains spectra with the defined precursor charge(s).
- `filterPrecursorIsotopes()`: groups MS2 spectra based on their precursor m/z and precursor intensity into predicted isotope groups and keep for each only the spectrum representing the monoisotopic precursor. MS1 spectra are returned as is. See documentation for `deisotopeSpectra()` below for details on isotope prediction and parameter description.
- `filterPrecursorMaxIntensity()`: filters the `Spectra` keeping for groups of (MS2) spectra with similar precursor m/z values (given parameters `ppm` and `tolerance`) the one with the highest precursor intensity. The function filters only MS2 spectra and returns all MS1 spectra. If precursor intensities are `NA` for all spectra within a spectra group, the first spectrum of that groups is returned. Note: some manufacturers don't provide precursor intensities. These can however also be estimated with `estimatePrecursorIntensity()`.
- `filterPrecursorMzRange()` (previously `filterPrecursorMz()` which is now deprecated): retains spectra with a precursor m/z within the provided m/z range. See examples for details on selecting spectra with a precursor m/z for a target m/z accepting a small difference in *ppm*.
- `filterPrecursorMzValues()`: retains spectra with precursor m/z matching any of the provided m/z values (given `ppm` and `tolerance`). Spectra with missing precursor m/z value (e.g. MS1 spectra) are dropped.
- `filterPrecursorScan()`: retains parent (e.g. MS1) and children scans (e.g. MS2) of acquisition number `acquisitionNum`. Returns the filtered `Spectra` (with spectra in their original order). Parameter `f` allows to define which spectra belong to the same sample or original data file (defaults to `f = dataOrigin(object)`). See also `fragmentGroupIndex()` for a function to generate an `integer` index grouping MS^n spectra with their corresponding MS1 spectra based on acquisition order.
- `filterRanges()`: allows filtering of the `Spectra` object based on user defined *numeric* ranges (parameter `ranges`) for one or more available spectra variables in object (spectra variable names can be specified with parameter `spectraVariables`). Spectra for which the value of a spectra variable is within it's defined range are retained. If multiple ranges/spectra variables are defined, the `match` parameter can be used to specify whether all conditions (`match = "all"`; the default) or if any of the conditions must match (`match = "any"`; all spectra for which values are within any of the provided ranges are retained).
- `filterRt()`: retains spectra of MS level `msLevel` with retention times (in seconds) within (`>=`) `rt[1]` and (`<=`) `rt[2]`. This retention time filter is applied to all spectra (regardless of their MS level) if `msLevel. = integer()` (the default). Returns the filtered `Spectra` (with spectra in their original order).
- `filterValues()`: allows filtering of the `Spectra` object based on similarities of *numeric* values of one or more `spectraVariables(object)` (parameter `spectraVariables`) to provided values (parameter `values`) given acceptable differences (parameters tolerance and ppm). If multiple values/spectra variables are defined, the `match` parameter can be used to specify whether all conditions (`match = "all"`; the default) or if any of the conditions must match (`match = "any"`; all spectra for which values are within any of the provided ranges are retained).

Filter content of `spectraData()`

The functions described in this section filter the content from a `Spectra`'s spectra data, i.e. affect values of, or complete, spectra variables. None of these functions reduces the object's number of spectra.

- `dropNaSpectraVariables()`: removes spectra variables (i.e. columns in the object's `spectraData` that contain only missing values (`NA`). Note that while columns with only `NA`s are removed, a `spectraData()` call after `dropNaSpectraVariables()` might still show columns containing `NA` values for *core* spectra variables. The total number of spectra is not changed by this function. By setting parameter `onlyCore = TRUE` only core spectra variables (`coreSpectraVariables()`) are evaluated for removal. Any spectra variable added by the user will be retained, even if they contain only `NA` values. Defaults to `onlyCore = FALSE`.
- `selectSpectraVariables()`: reduces the information within the object to the selected spectra variables: all data for variables not specified will be dropped. For mandatory columns (i.e., those listed by `coreSpectraVariables()`, such as *msLevel*, *rtime* ...) only the values will be dropped but not the variable itself. Additional (or user defined) spectra variables will be completely removed. Returns the filtered `Spectra`.

Filter content of `peaksData()`

The functions described in this section filter the content of the `Spectra`'s peaks data, i.e. either the number or the values (*m/z* or intensity values) of the mass peaks. Also, the actual operation is only executed once peaks data is accessed (through `peaksData()`, `mz()` or `intensity()`) or `applyProcessing()` is called. These operations don't affect the number of spectra in the `Spectra` object.

- `deisotopeSpectra()`: *deisotopes* each spectrum keeping only the monoisotopic peak for groups of isotopologues. Isotopologues are estimated using the `MetaboCoreUtils::isotopologues()` function from the *MetaboCoreUtils* package. Note that the default parameters for isotope prediction/detection have been determined using data from the Human Metabolome Database (HMDB) and isotopes for elements other than CHNOPS might not be detected. See parameter `substDefinition` in the documentation of `MetaboCoreUtils::isotopologues()` for more information. The approach and code to define the parameters for isotope prediction is described here.
- `filterFourierTransformArtefacts()`: removes (Orbitrap) fast fourier artefact peaks from spectra (see examples below). The function iterates through all intensity ordered peaks in a spectrum and removes all peaks with an m/z within +/- `halfWindowSize` of the current peak if their intensity is lower than `threshold` times the current peak's intensity. Additional parameters `keepIsotopes`, `maxCharge` and `isotopeTolerance` allow to avoid removing of potential `[13]C` isotope peaks (`maxCharge` being the maximum charge that should be considered and `isotopeTolerance` the absolute acceptable tolerance for matching their m/z). See `filterFourierTransformArtefacts()` for details and background and `deisitopeSpectra()` for an alternative.
- `filterIntensity()`: filters mass peaks in each spectrum keeping only those with intensities that are within the provided range or match the criteria of the provided function. For the former, parameter `intensity` has to be a `numeric` defining the intensity range, for the latter a `function` that takes the intensity values of the spectrum and returns a `logical` whether the peak should be retained or not (see examples below for details) - additional parameters to the function can be passed with `...`. To remove only peaks with intensities below a certain threshold, say 100, use `intensity = c(100, Inf)`. Note: also a single value can be passed with the `intensity` parameter in which case an upper limit of `Inf` is used. Note that this function removes also peaks with missing intensities (i.e. an intensity of `NA`). Parameter `msLevel.` allows to restrict the filtering to spectra of the specified MS level(s).

- `filterMzRange()`: filters mass peaks in the object keeping or removing those in each spectrum that are within the provided m/z range. Whether peaks are retained or removed can be configured with parameter `keep` (default `keep = TRUE`).
- `filterMzValues()`: filters mass peaks in the object keeping all peaks in each spectrum that match the provided m/z value(s) (for `keep = TRUE`, the default) or removing all of them (for `keep = FALSE`). The m/z matching considers also the absolute `tolerance` and m/z-relative `ppm` values. `tolerance` and `ppm` have to be of length 1.
- `filterPeaksRanges()`: filters mass peaks of a `Spectra` object using any set of range-based filters on numeric spectra or peaks variables. See `filterPeaksRanges()` for more information.
- `filterPrecursorPeaks()`: removes peaks from each spectrum in `object` with an m/z equal or larger than the m/z of the precursor, depending on the value of parameter `mz`: for `mz = ==" (the default) peaks wit eranceandppm, respectively) are removed. For mz = ">="all peaks with an m/z larger or equal to the pr` (e.g. typically for MS1 spectra).
- `reduceSpectra()`: keeps for groups of peaks with similar m/z values in (given `ppm` and `tolerance`) in each spectrum only the mass peak with the highest intensity removing all other peaks hence *reducing* each spectrum to the highest intensity peaks per *peak group*. Peak groups are defined using the `MsCoreUtils::group()` function from the *MsCoreUtils* package. See also the `combinePeaks()` function for an alternative function to combine peaks within each spectrum.

Author(s)

Sebastian Gibb, Johannes Rainer, Laurent Gatto, Philippine Louail, Nir Shahaf

See Also

- `combineSpectra()` for functions to combine or aggregate `Spectra`.
- `combinePeaks()` for functions to combine or aggregate a `Spectra`'s `peaksData()`

Examples

```
## Load a `Spectra` object with LC-MS/MS data.
fl <- MsDataHub::PestMix1_DDA.mzML()
sps_dda <- Spectra(fl)
sps_dda


##  --------  SUBSET SPECTRA  --------

## Subset to the first 3 spectra
tmp <- sps_dda[1:3]
tmp
length(tmp)

## Subset to all MS2 spectra; this could be done with [, or, more
## efficiently, with the `filterMsLevel` function:
sps_dda[msLevel(sps_dda) == 2L]
filterMsLevel(sps_dda, 2L)
```

```
## Filter the object keeping only MS2 spectra with an precursor m/z value
## between a specified range:
filterPrecursorMzRange(sps_dda, c(80, 90))

## Filter the object to MS2 spectra with an precursor m/z matching a
## pre-defined value (given ppm and tolerance)
filterPrecursorMzValues(sps_dda, 85, ppm = 5, tolerance = 0.1)

## The `filterRanges()` function allows to filter a `Spectra` based on
## numerical ranges of any of its (numerical) spectra variables.
## First, determine the variable(s) on which to base the filtering:
sv <- c("rtime", "precursorMz", "peaksCount")
## Note that ANY variables can be chosen here, and as many as wanted.

## Define the ranges (pairs of values with lower and upper boundary) to be
## used for the individual spectra variables. The first two values will be
## used for the first spectra variable (e.g., `"rtime"` here), the next two
## for the second (e.g. `"precursorMz"` here) and so on:
ranges <- c(30, 350, 200, 500, 350, 600)

## Input the parameters within the filterRanges function:
filt_spectra <- filterRanges(sps_dda, spectraVariables = sv,
                ranges = ranges)
filt_spectra

## `filterRanges()` can also be used to filter a `Spectra` object with
## multiple ranges for the same `spectraVariable` (e.g, here `"rtime"`)
sv <- c("rtime", "rtime")
ranges <- c(30, 100, 200, 300)
filt_spectra <- filterRanges(sps_dda, spectraVariables = sv,
                ranges = ranges, match = "any")
filt_spectra

## While `filterRanges()` filtered on numeric ranges, `filterValues()`
## allows to filter an object matching spectra variable values to user
## provided values (allowing to configure allowed differences using the
## `ppm` and `tolerance` parameters).
## First determine the variable(s) on which to base the filtering:
sv <- c("rtime", "precursorMz")
## Note that ANY variables can be chosen here, and as many as wanted.

## Define the values that will be used to filter the spectra based on their
## similarities to their respective `spectraVariables`.
## The first values in the parameters values, tolerance and ppm will be
## used for the first spectra variable (e.g. `"rtime"` here), the next for
## the second (e.g. `"precursorMz"` here) and so on:
values <- c(350, 80)
tolerance <- c(100, 0.1)
ppm <- c(0, 50)

## Input the parameters within the `filterValues()` function:
filt_spectra <- filterValues(sps_dda, spectraVariables = sv,
```

```
                values = values, tolerance = tolerance, ppm = ppm)
filt_spectra


##  --------  FILTER SPECTRA DATA  --------

## Remove spectra variables without content (i.e. with only missing values)
sps_noNA <- dropNaSpectraVariables(sps_dda)

## This reduced the size of the object slightly
print(object.size(sps_dda), unit = "MB")
print(object.size(sps_noNA), unit = "MB")

## With the `selectSpectraVariables()` function it is in addition possible
## to subset the data of a `Spectra` to the selected columns/variables,
## keeping only their data:
tmp <- selectSpectraVariables(sps_dda, c("msLevel", "mz", "intensity",
    "scanIndex"))
print(object.size(tmp), units = "MB")

## Except the selected variables, all data is now removed. Accessing
## core spectra variables still works, but returns only NA
rtime(tmp) |> head()


##  --------  FILTER PEAKS DATA  --------

## `filterMzValues()` filters the mass peaks data of a `Spectra` retaining
## only those mass peaks with an m/z value matching the provided value(s).
sps_sub <- filterMzValues(sps_dda, mz = c(103, 104), tolerance = 0.3)

## The filtered `Spectra` has the same length
length(sps_dda)
length(sps_sub)

## But the number of mass peaks changed
lengths(sps_dda) |> head()
lengths(sps_sub) |> head()

## This function can also be used to remove specific peaks from a spectrum
## by setting `keep = FALSE`.
sps_sub <- filterMzValues(sps_dda, mz = c(103, 104),
    tolerance = 0.3, keep = FALSE)
lengths(sps_sub) |> head()

## With the `filterMzRange()` function it is possible to keep (or remove)
## mass peaks with m/z values within a specified numeric range.
sps_sub <- filterMzRange(sps_dda, mz = c(100, 150))
lengths(sps_sub) |> head()

## See also the `filterPeaksRanges()` function for a more flexible framework
## to filter mass peaks
```

```
## Removing fourier transform artefacts seen in Orbitra data.

## Loading an Orbitrap spectrum with artefacts.
data(fft_spectrum)
plotSpectra(fft_spectrum, xlim = c(264.5, 265.5))
plotSpectra(fft_spectrum, xlim = c(264.5, 265.5), ylim = c(0, 5e6))

fft_spectrum <- filterFourierTransformArtefacts(fft_spectrum)
fft_spectrum
plotSpectra(fft_spectrum, xlim = c(264.5, 265.5), ylim = c(0, 5e6))

## Using a few examples peaks in your data you can optimize the parameters
fft_spectrum_filtered <- filterFourierTransformArtefacts(fft_spectrum,
                                                         halfWindowSize = 0.2,
                                                         threshold = 0.005,
                                                         keepIsotopes = TRUE,
                                                         maxCharge = 5,
                                                         isotopeTolerance = 0.005
                                                         )

fft_spectrum_filtered
length(mz(fft_spectrum_filtered)[[1]])
plotSpectra(fft_spectrum_filtered, xlim = c(264.5, 265.5), ylim = c(0, 5e6))


## *Reducing* a `Spectra` keeping for groups of mass peaks (characterized
## by similarity of their m/z values) only one representative peak. This
## function helps cleaning fragment spectra.
## Filter the data set to MS2 spectra
ms2 <- filterMsLevel(sps_dda, 2L)

## For groups of fragment peaks with a difference in m/z < 0.1, keep only
## the largest one.
ms2_red <- reduceSpectra(ms2, ppm = 0, tolerance = 0.1)
lengths(ms2) |> tail()
lengths(ms2_red) |> tail()
```

---

estimatePrecursorIntensity,Spectra-method

Estimate Precursor Intensities

Description

Some MS instrument manufacturers don't provide precursor intensities for fragment spectra. These can however be estimated, given that also MS1 spectra are available. The `estimatePrecursorIntensity()` funtion defines the precursor intensities for MS2 spectra using the intensity of the matching MS1 peak from the closest MS1 spectrum (i.e. the last MS1 spectrum measured before the respective MS2 spectrum). With `method = "interpolation"` it is also possible to calculate the precursor intensity based on an interpolation of intensity values (and retention times) of the matching MS1 peaks from the previous and next MS1 spectrum. See below for an example.

Usage

```
## S4 method for signature 'Spectra'
estimatePrecursorIntensity(
  object,
  ppm = 20,
  tolerance = 0,
  method = c("previous", "interpolation"),
  msLevel. = 2L,
  f = dataOrigin(object),
  BPPARAM = bpparam()
)
```

Arguments

object	Spectra with MS1 and MS2 spectra.
ppm	numeric(1) with the maximal allowed relative difference of m/z values between the precursor m/z of a spectrum and the m/z of the respective ion on the MS1 scan.
tolerance	numeric(1) with the maximal allowed difference of m/z values between the precursor m/z of a spectrum and the m/z of the respective ion on the MS1 scan.
method	character(1) defining whether the precursor intensity should be estimated on the previous MS1 spectrum (method = "previous", the default) or based on an interpolation on the previous and next MS1 spectrum (method = "interpolation").
msLevel.	integer(1) the MS level for which precursor intensities should be estimated. Defaults to 2L.
f	factor (or vector to be coerced to factor) defining which spectra belong to the same original data file (sample). Defaults to f = dataOrigin(x).
BPPARAM	Parallel setup configuration. See processingChunkSize() and BiocParallel::bpparam() for more information.

Author(s)

Johannes Rainer with feedback and suggestions from Corey Broeckling

Examples

```
#' ## Calculating the precursor intensity for MS2 spectra:
##
## Some MS instrument manufacturer don't report the precursor intensities
## for MS2 spectra. The `estimatePrecursorIntensity` function can be used
## in these cases to calculate the precursor intensity on MS1 data. Below
## we load an mzML file from a vendor providing precursor intensities and
## compare the estimated and reported precursor intensities.
f <- MsDataHub::TMT_Erwinia_1uLSike_Top10HCD_isol2_45stepped_60min_01.20141210.mzML.gz()
tmt <- Spectra(f, backend = MsBackendMzR())
pmi <- estimatePrecursorIntensity(tmt)
plot(pmi, precursorIntensity(tmt))
```

```
## We can also replace the original precursor intensity values with the
## newly calculated ones
tmt$precursorIntensity <- pmi
```

`estimatePrecursorMz` *Estimating precursor m/z valus for DDA data*

Description

MS data from Waters instruments are calibrated through the *Lock Mass*, but, while all m/z values of mass peaks in each spectrum will be calibrated by this method, the reported precursor m/z might not. The precursor m/z in the converted mzML file will have m/z values from quadrupole isolation windows instead of accurate m/z values. See also the GNPS documentation for more information.

The `estimatePrecursorMz()` function estimates/adjusts the reported precursor m/z of a fragment spectrum using the following approach: in data dependent acquisition (DDA) mode, the MS instrument will select ions with the highest intensities in one MS scan for fragmentation. Thus, for each fragment spectrum, this method identifies in the previous MS1 spectrum the peak with the highest intensity and an m/z value similar to the fragment spectrum's reported precursor m/z (given parameters `tolerance` and `ppm`). This m/z value is then reported. Since the fragment spectrum's potential MS1 mass peak is selected based on its intensity, this method should **only be used for DDA data**.

Usage

```
estimatePrecursorMz(object, tolerance = 0.3, ppm = 10, BPPARAM = SerialParam())
```

Arguments

`object`	`Spectra()` object with **DDA** data.
`tolerance`	`numeric(1)` defining an absolute acceptable difference in m/z between the fragment spectra's reported precursor m/z and the MS1 peaks considered as the precursor peak. All MS1 peaks from the previous MS1 scan with an m/z between the fragment spectrum's precursorMz +/- (tolerance + ppm(precursorMz, ppm)) are considered.
`ppm`	`numeric(1)` defining the m/z dependent acceptable difference in m/z. See documentation of parameter `tolerance` for more information.
`BPPARAM`	parallel processing setup. Defaults to `BPPARAM = SerialParam()`. See `BiocParallel::SerialParam()` for more information.

Value

`numeric` of length equal to the number of spectra in `object` with the fragment spectra's estimated precursor m/z values. For MS1 spectra `NA_real_` values are returned. The original precursor m/z is reported for MS2 spectra for which no matching MS1 peak was found.

Note

This approach is applicable only when fragment spectra are obtained through data-dependent acquisition (DDA), as it assumes that the peak with the highest intensity within the given isolation m/z window (from the previous MS1 spectrum) corresponds to the precursor ion.

The spectra in `object` have to be ordered by their retention time.

Users of this function should evaluate and compare the estimated precursor m/z values with the originally reported one and only consider adjusted values they feel comfortable with.

Author(s)

Mar Garcia-Aloy, Johannes Rainer

See Also

`addProcessing()` for other data analysis and manipulation functions.

Examples

```
## Load a DDA test data set. For the present data set no large differences
## between the reported and the *actual* precursor m/z are expected.
fl <- MsDataHub::PestMix1_DDA.mzML()
s <- Spectra(fl)

pmz <- estimatePrecursorMz(s)

## plot the reported and estimated precursor m/z values against each other
plot(precursorMz(s), pmz)
abline(0, 1)

## They seem highly similar, but are they identical?
identical(precursorMz(s), pmz)
all.equal(precursorMz(s), pmz)

## Plot also the difference of m/z values against the m/z value
plot(precursorMz(s), precursorMz(s) - pmz, xlab = "precursor m/z",
    ylab = "difference reported - estimated precursor m/z")

## we could then replace the reported precursor m/z values
s$precursorMz <- pmz
```

`fillCoreSpectraVariables` *Fill spectra data with columns for missing core variables*

Description

`fillCoreSpectraVariables()` fills a provided `data.frame` with columns for eventually missing *core* spectra variables. The missing core variables are added as new columns with missing values (`NA`) of the correct data type. Use `coreSpectraVariables()` to list the set of core variables and their data types.

Usage

```
fillCoreSpectraVariables(
  x = data.frame(),
  columns = names(coreSpectraVariables())
)
```

Arguments

`x`	`data.frame` or `DataFrame` with potentially present core variable columns.
`columns`	`character` with the names of the (core) spectra variables that should be added if not already present in `x`. Defaults to `columns = names(coreSpectraVariables())`.

Value

input data frame `x` with missing core variables added (with the correct data type).

Examples

```
## Define a data frame
a <- data.frame(msLevel = c(1L, 1L, 2L), other_column = "b")

## Add missing core chromatogram variables to this data frame
fillCoreSpectraVariables(a)

## The data.frame thus contains columns for all core spectra
## variables in the respective expected data type (but filled with
## missing values).
```

`filterFourierTransformArtefacts`

Fast fourier transform artefact filter

Description

The `filterFourierTransformArtefacts()` function removes (Orbitrap) fast fourier artefact peaks from spectra. Such artefacts (also referred to as *ripples*) seem to be related to the *ringing* phenomenon and are frequently seen in Orbitrap data as small random mass peaks ~ 0.01 Da from a main peak with a very large intensity. See also here for more details and information. The data set `fft_spectrum` represents a `Spectra()` object with a single Orbitrap spectrum with such artefacts (see examples below).

See also `Spectra()` (section *Data subsetting, filtering and merging) for the definition of the function.

Details

The current implementation iterates through all intensity ordered peaks in a spectrum and removes all peaks with an m/z within +/- `halfWindowSize` of the current peak if their intensity is lower than `threshold` times the current peak's intensity. Additional parameters `keepIsotopes`, `maxCharge` and `isotopeTolerance` allow to avoid removing of potential `[13]C` isotope peaks (`maxCharge` being the maximum charge that should be considered and `isotopeTolerance` the absolute acceptable tolerance for matching their m/z).

Author(s)

Jan Stanstrup, Johannes Rainer

Examples

```
library(Spectra)
data(fft_spectrum)

plotSpectra(fft_spectrum)

## Focus on an artefact
plotSpectra(fft_spectrum, xlim = c(264.5, 265.5))
plotSpectra(fft_spectrum, xlim = c(264.5, 265.5), ylim = c(0, 5e6))

fft_spectrum <- filterFourierTransformArtefacts(fft_spectrum)
fft_spectrum
plotSpectra(fft_spectrum, xlim = c(264.5, 265.5), ylim = c(0, 5e6))

## R code to download/extract the data.

## Not run:
library(Spectra)
# get orbitrap data
download.file("https://ftp.ebi.ac.uk/pub/databases/metabolights/studies/public/MTBLS469/AV_01_v2_male_arm1_jui
data <- Spectra("AV_01_v2_male_arm1_juice.mzXML")
extracted_spectrum <- data[195]

## End(Not run)
```

`filterPeaksRanges` *Filter peaks based on spectra and peaks variable ranges*

Description

The `filterPeaksRanges()` function allows to filter the peaks matrices of a Spectra object using any set of range-based filters on numeric spectra variables or peaks variables. These ranges can be passed to the function using the `...` as `<variable name> = <range>` pairs. `<variable name>` has to be an available spectra or peaks variable. `<range>` can be a `numeric` of length 2 defining the lower and upper boundary, or a `numeric` two-column matrix (multi-row matrices are also supported, see further below). `filterPeaksRanges(s, mz = c(200, 300))` would for example reduce

the peaks matrices of the Spectra object s to mass peaks with an m/z value between 200 and 300. filterPeaksRanges() returns the original Spectra object with the filter operation added to the processing queue. Thus, the filter gets **only** applied when the peaks data gets extracted with mz(), intensity() or peaksData(). If ranges for both spectra **and** peaks variables are defined, the function evaluates first whether the spectra variable value for a spectrum is within the provided range and, if so, applies also the peaks variable-based filter (otherwise an empty peaks matrix is returned).

If more than one spectra variable and/or peaks variable are defined, their filter results are combined with a logical AND: a peak matrix is only returned for a spectrum if all values of spectra variables are within the provided (respective) ranges for spectra variables, and this matrix is further filtered to contain only those peaks which values are within the provided peaks variable ranges.

Filtering with multiple ranges per spectra and peaks variables is also supported: ranges can also be provided as multi-row numeric (two-column) matrices. In this case, the above described procedure is applied for each row separately and their results are combined with a logical OR, i.e. peaks matrices are returned that match any of the conditions/filters of a row. The number of rows of the provided ranges (being it for spectra or peaks variables) have to match.

Missing value handling: any comparison which involves a missing value (being it a spectra variable value, a peaks variable value or a value in one of the provided ranges) is treated as a logical FALSE. For example, if the retention time of a spectrum is NA and the data is filtered using a retention time range, an empty peaks matrix is returned (for keep = TRUE, for keep = FALSE the full peaks matrix is returned).

Usage

```
filterPeaksRanges(object, ..., keep = TRUE)
```

Arguments

object	A Spectra object.
...	the ranges for the spectra and/or peaks variables. Has to be provided as <name> = <range> pairs with <name> being the name of a spectra or peaks variable (of numeric data type) and <range> being either a numeric of length 2 or a numeric two column matrix (see function desription above for details),
keep	logical(1) whether to keep (default) or remove peaks that match the provided range(s).

Note

In contrast to some other *filter* functions, this function does not provide a msLevel parameter that allows to define the MS level of spectra on which the filter should be applied. The filter(s) will always be applied to **all** spectra (irrespectively of their MS level). Through combination of multiple filter ranges it is however possible to apply MS level-dependent filters (see examples below for details).

The filter will not be applied immediately to the data but only executed when the mass peak data is accessed (through peaksData(), mz() or intensity()) or by calling applyProcessing().

Author(s)

Johannes Rainer

Examples

```
## Define a test Spectra
d <- data.frame(rtime = c(123.2, 134.2), msLevel = c(1L, 2L))
d$mz <- list(c(100.1, 100.2, 100.3, 200.1, 200.2, 300.3),
    c(100.3, 100.4, 200.2, 400.3, 400.4))
## Use the index of the mass peak within the spectrum as index for
## better illustration of filtering results
d$intensity <- list(c(1:6), 1:5)
s <- Spectra(d)
s

## Filter peaks removing all mass peaks with an m/z between 200 and 300
res <- filterPeaksRanges(s, mz = c(200, 300), keep = FALSE)
res

## The Spectra object has still the same length and spectra variables
length(res)
res$rtime

## The filter gets applied when mass peak data gets extracted, using either
## `mz()`, `intensity()` or `peaksData()`. The filtered peaks data does
## not contain any mass peaks with m/z values between 200 and 300:
peaksData(res)[[1L]]
peaksData(res)[[2L]]

## We next combine spectra and filter variables. We want to keep only mass
## peaks of MS2 spectra that have an m/z between 100 and 110.
res <- filterPeaksRanges(s, mz = c(100, 110), msLevel = c(2, 2))
res
length(res)

## Only data for peaks are returned for which the spectra's MS level is
## between 2 and 2 and with an m/z between 100 and 110. The peaks data for
## the first spectrum, that has MS level 1, is thus empty:
peaksData(res)[[1L]]

## While the peaks matrix for the second spectrum (with MS level 2) contains
## the mass peaks with m/z between 100 and 110.
peaksData(res)[[2L]]

## To keep also the peaks data for the first spectrum, we need to define
## an additional set of ranges, which we define using a second row in each
## ranges matrix. We use the same filter as above, i.e. keeping only mass
## peaks with an m/z between 100 and 110 for spectra with MS level 2, but
## add an additional row for MS level 1 spectra keeping mass peaks with an
## m/z between 0 and 2000. Filter results of different rows are combined
## using a logical OR, i.e. peaks matrices with mass peaks are returned
## matching either the first, or the second row.
res <- filterPeaksRanges(s, mz = rbind(c(100, 110), c(0, 1000)),
    msLevel = rbind(c(2, 2), c(1, 1)))

## The results for the MS level 2 spectrum are the same as before, but with
```

```
## the additional row we keep the full peaks matrix of the MS1 spectrum:
peaksData(res)[[1L]]
peaksData(res)[[2L]]

## As a last example we define a filter that keeps all mass peaks with an
## m/z either between 100 and 200, or between 300 and 400.
res <- filterPeaksRanges(s, mz = rbind(c(100, 200), c(300, 400)))
peaksData(res)[[1L]]
peaksData(res)[[2L]]

## Such filters could thus be defined to restrict/filter the MS data to
## specific e.g. retention time and m/z ranges.
```

`fragmentGroupIndex` *Mass fragmentation collections of each full scan*

Description

This function generates an `integer` index grouping MS^n spectra (MS level > 1) with their corresponding MS1 spectra based on acquisition order. Each group contains exactly one MS1 spectrum and all subsequent higher-level spectra (MS2, MS3, ...) acquired until the next MS1 scan. MS1-only spectra are also assigned sequential group IDs.

Note that this function:

- does not consider the direct relationship between a precursor scan and the associated product scans,
- and does not distinguish between different fragmentation trees.

For example, all MS3 scans measured after a given MS1 are grouped together with all MS2 scans from that MS1, regardless of which MS2 spectrum they originated from. See `filterPrecursorScan()` for a function that considers relationships between fragment and precursor scans.

Usage

```
fragmentGroupIndex(object, BPPARAM = SerialParam())
```

Arguments

`object`	A `Spectra` object (from the **Spectra** package) containing MS data. Must include at least two MS levels (`msLevel`) and be ordered by `acquisitionNum` within each `dataOrigin`.
`BPPARAM`	A `BiocParallelParam` object for parallel execution. Defaults to `SerialParam()`.

Value

An `integer` vector of the same length as `object`. Each element gives the group index associated with the corresponding spectrum (MS1 or MS^n). Group indices are unique across all files (`dataOrigin` values).

Note

- Each file (`dataOrigin`) must contain at least one MS1 spectrum.
- If a group contains only MS1 spectra, each MS1 is assigned a unique group ID.
- The user is responsible for ensuring that spectra are correctly ordered. Improper ordering may lead to incorrect groupings.

Author(s)

Philippine Louail

See Also

`filterPrecursorScan()` for a function that instead returns a `Spectra` object containing each parent (e.g., MS1) and its direct child scans (e.g., MS2) according to their acquisition numbers.

Examples

```
fl <- MsDataHub::PestMix1_DDA.mzML()
sps_dda <- Spectra(fl)
idx <- fragmentGroupIndex(sps_dda)
head(idx)
```

`joinPeaks` *Join (map) peaks of two spectra*

Description

These functions map peaks from two spectra with each other if the difference between their m/z values is smaller than defined with parameters `tolerance` and `ppm`. All functions take two matrices

- `joinPeaks()`: maps peaks from two spectra allowing to specify the type of *join* that should be performed: `type = "outer"` each peak in `x` will be matched with each peak in `y`, for peaks that do not match any peak in the other spectra an `NA` intensity is returned. With `type = "left"` all peaks from the left spectrum (`x`) will be matched with peaks in `y`. Peaks in `y` that do not match any peak in `x` are omitted. `type = "right"` is the same as `type = "left"` only for `y`. Only peaks that can be matched between `x` and `y` are returned by `type = "inner"`, i.e. only peaks present in both spectra are reported.
- `joinPeaksGnps()`: matches/maps peaks between spectra with the same approach used 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 spectras' 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 peaks in the returned matrices might be reported multiple times. Note that if one of `xPrecursorMz` or `yPrecursorMz` are `NA` or if both are the same, the results are the same as with `joinPeaks()`. To calculate GNPS similarity scores, `MsCoreUtils::gnps()` should be called on the aligned peak matrices (i.e. `compareSpectra` should be called with `MAPFUN = joinPeaksGnps` and `FUN = MsCoreUtils::gnps`).

- `joinPeaksNone()`: does not perform any peak matching but simply returns the peak matrices in a `list`. This function should be used with the `MAPFUN` parameter of `compareSpectra()` if the spectra similarity function used (parameter `FUN` of `compareSpectra()`) performs its own peak matching and does hence not expect matched peak matrices as an input.

Usage

```
joinPeaks(x, y, type = "outer", tolerance = 0, ppm = 10, ...)

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

joinPeaksNone(x, y, ...)
```

Arguments

`x`	`matrix` with two columns `"mz"` and `"intensity"` containing the m/z and intensity values of the mass peaks of a spectrum.
`y`	`matrix` with two columns `"mz"` and `"intensity"` containing the m/z and intensity values of the mass peaks of a spectrum.
`type`	For `joinPeaks()` and `joinPeaksGnps()`: `character(1)` specifying the type of join that should be performed. See function description for details.
`tolerance`	`numeric(1)` defining a constant maximal accepted difference between m/z values of peaks from the two spectra to be matched/mapped.
`ppm`	`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.
`...`	optional parameters passed to the `MsCoreUtils::join()` function.
`xPrecursorMz`	for `joinPeaksGnps()`: `numeric(1)` with the precursor m/z of the spectrum `x`.
`yPrecursorMz`	for `joinPeaksGnps()`: `numeric(1)` with the precursor m/z of the spectrum `y`.

Value

All functions return a `list` of elements `"x"` and `"y"` each being a two column matrix with m/z (first column) and intensity values (second column). The two matrices contain the matched peaks between input matrices `x` and `y` and hence have the same number of rows. Peaks present in `x` but not in the `y` input matrix have m/z and intensity values of `NA` in the result matrix for `y` (and *vice versa*).

Implementation notes

A mapping function must take two numeric matrices `x` and `y` as input and must return `list` with two elements named `"x"` and `"y"` that represent the aligned input matrices. The function should also have `...` in its definition. Parameters `ppm` and `tolerance` are suggested but not required.

Author(s)

Johannes Rainer, Michael Witting

See Also

- `compareSpectra()` for the function to calculate similarities between spectra.
- `MsCoreUtils::gnps()` in the *MsCoreUtils* package for more information on the GNPS similarity score.

Examples

```
x <- cbind(c(31.34, 50.14, 60.3, 120.9, 230, 514.13, 874.1),
    1:7)
y <- cbind(c(12, 31.35, 70.3, 120.9 + ppm(120.9, 5),
    230 + ppm(230, 10), 315, 514.14, 901, 1202),
    1:9)

## No peaks with identical m/z
joinPeaks(x, y, ppm = 0, type = "inner")

## With ppm 10 two peaks are overlapping
joinPeaks(x, y, ppm = 10, type = "inner")

## Outer join: contain all peaks from x and y
joinPeaks(x, y, ppm = 10, type = "outer")

## Left join: keep all peaks from x and those from y that match
joinPeaks(x, y, ppm = 10, type = "left")

## Right join: keep all peaks from y and those from x that match. Using
## a constant tolerance of 0.01
joinPeaks(x, y, tolerance = 0.01, type = "right")

## GNPS-like peak matching

## 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 joinPeaks identifies only 2 matching peaks: 1 and 5
joinPeaks(x, y)
```

```
## joinPeaksGnps finds 4 matches
joinPeaksGnps(x, y, pmz_x, pmz_y)

## with one of the two precursor m/z being NA, the result are the same as
## with joinPeaks (with type = "left").
joinPeaksGnps(x, y, pmz_x, yPrecursorMz = NA)
```

MsBackend *Mass spectrometry data backends*

Description

Note that the classes described here are not meant to be used directly by the end-users and the material in this man page is aimed at package developers.

MsBackend is a virtual class that defines what each different backend needs to provide. MsBackend objects provide access to mass spectrometry data. Such backends can be classified into *in-memory* or *on-disk* backends, depending on where the data, i.e spectra (m/z and intensities) and spectra annotation (MS level, charge, polarity, ...) are stored.

Typically, in-memory backends keep all data in memory ensuring fast data access, while on-disk backends store (parts of) their data on disk and retrieve it on demand.

The *Backend functions and implementation notes for new backend classes* section documents the API that a backend must implement.

Currently available backends are:

- MsBackendMemory and MsBackendDataFrame: store all data in memory. The MsBackendMemory is optimized for accessing and processing the peak data (i.e. the numerical matrices with the m/z and intensity values) while the MsBackendDataFrame keeps all data in a DataFrame.
- MsBackendMzR: stores the m/z and intensities on-disk in raw data files (typically mzML or mzXML) and the spectra annotation information (header) in memory in a DataFrame. This backend requires the mzR package.
- MsBackendHdf5Peaks: stores the m/z and intensities on-disk in custom hdf5 data files and the remaining spectra variables in memory (in a DataFrame). This backend requires the rhdf5 package.

See below for more details about individual backends.

Usage

```
## S4 method for signature 'MsBackend'
backendBpparam(object, BPPARAM = bpparam())

## S4 method for signature 'MsBackend'
backendInitialize(object, ...)

## S4 method for signature 'list'
backendMerge(object, ...)
```

```
## S4 method for signature 'MsBackend'
backendMerge(object, ...)

## S4 method for signature 'MsBackend'
backendParallelFactor(object, ...)

## S4 method for signature 'MsBackend'
export(object, ...)

## S4 method for signature 'MsBackend'
acquisitionNum(object)

## S4 method for signature 'MsBackend'
peaksData(object, columns = c("mz", "intensity"))

## S4 method for signature 'MsBackend'
peaksVariables(object)

## S4 method for signature 'MsBackend,dataframeOrDataFrameOrmatrix'
cbind2(x, y = data.frame(), ...)

## S4 method for signature 'MsBackend'
centroided(object)

## S4 replacement method for signature 'MsBackend'
centroided(object) <- value

## S4 method for signature 'MsBackend'
collisionEnergy(object)

## S4 replacement method for signature 'MsBackend'
collisionEnergy(object) <- value

## S4 method for signature 'MsBackend'
dataOrigin(object)

## S4 replacement method for signature 'MsBackend'
dataOrigin(object) <- value

## S4 method for signature 'MsBackend'
dataStorage(object)

## S4 replacement method for signature 'MsBackend'
dataStorage(object) <- value

## S4 method for signature 'MsBackend'
dropNaSpectraVariables(object, onlyCore = FALSE)
```

```
## S4 method for signature 'MsBackend,ANY'
extractByIndex(object, i)

## S4 method for signature 'MsBackend,missing'
extractByIndex(object, i)

## S4 method for signature 'MsBackend'
filterAcquisitionNum(object, n, file, ...)

## S4 method for signature 'MsBackend'
filterDataOrigin(object, dataOrigin = character())

## S4 method for signature 'MsBackend'
filterDataStorage(object, dataStorage = character())

## S4 method for signature 'MsBackend'
filterEmptySpectra(object, ...)

## S4 method for signature 'MsBackend'
filterIsolationWindow(object, mz = numeric(), ...)

## S4 method for signature 'MsBackend'
filterMsLevel(object, msLevel = integer())

## S4 method for signature 'MsBackend'
filterPolarity(object, polarity = integer())

## S4 method for signature 'MsBackend'
filterPrecursorMzRange(object, mz = numeric())

## S4 method for signature 'MsBackend'
filterPrecursorMz(object, mz = numeric())

## S4 method for signature 'MsBackend'
filterPrecursorMzValues(object, mz = numeric(), ppm = 20, tolerance = 0)

## S4 method for signature 'MsBackend'
filterPrecursorCharge(object, z = integer())

## S4 method for signature 'MsBackend'
filterPrecursorScan(object, acquisitionNum = integer(), f = dataOrigin(object))

## S4 method for signature 'MsBackend'
filterRanges(
  object,
  spectraVariables = character(),
  ranges = numeric(),
```

```
  match = c("all", "any")
)

## S4 method for signature 'MsBackend'
filterRt(object, rt = numeric(), msLevel. = integer())

## S4 method for signature 'MsBackend'
filterValues(
  object,
  spectraVariables = character(),
  values = numeric(),
  ppm = 0,
  tolerance = 0,
  match = c("all", "any")
)

## S4 method for signature 'MsBackend'
intensity(object)

## S4 replacement method for signature 'MsBackend'
intensity(object) <- value

## S4 method for signature 'MsBackend'
ionCount(object)

## S4 method for signature 'MsBackend'
isCentroided(object, ...)

## S4 method for signature 'MsBackend'
isEmpty(x)

## S4 method for signature 'MsBackend'
isolationWindowLowerMz(object)

## S4 replacement method for signature 'MsBackend'
isolationWindowLowerMz(object) <- value

## S4 method for signature 'MsBackend'
isolationWindowTargetMz(object)

## S4 replacement method for signature 'MsBackend'
isolationWindowTargetMz(object) <- value

## S4 method for signature 'MsBackend'
isolationWindowUpperMz(object)

## S4 replacement method for signature 'MsBackend'
isolationWindowUpperMz(object) <- value
```

```
## S4 method for signature 'MsBackend'
isReadOnly(object)

## S4 method for signature 'MsBackend'
length(x)

## S4 method for signature 'MsBackend'
msLevel(object)

## S4 replacement method for signature 'MsBackend'
msLevel(object) <- value

## S4 method for signature 'MsBackend'
mz(object)

## S4 replacement method for signature 'MsBackend'
mz(object) <- value

## S4 method for signature 'MsBackend'
lengths(x, use.names = FALSE)

## S4 method for signature 'MsBackend'
polarity(object)

## S4 replacement method for signature 'MsBackend'
polarity(object) <- value

## S4 method for signature 'MsBackend'
precScanNum(object)

## S4 method for signature 'MsBackend'
precursorCharge(object)

## S4 method for signature 'MsBackend'
precursorIntensity(object)

## S4 method for signature 'MsBackend'
precursorMz(object)

## S4 replacement method for signature 'MsBackend'
precursorMz(object, ...) <- value

## S4 replacement method for signature 'MsBackend'
peaksData(object) <- value

## S4 method for signature 'MsBackend'
reset(object)
```

```
## S4 method for signature 'MsBackend'
rtime(object)

## S4 replacement method for signature 'MsBackend'
rtime(object) <- value

## S4 method for signature 'MsBackend'
scanIndex(object)

## S4 method for signature 'MsBackend'
selectSpectraVariables(object, spectraVariables = spectraVariables(object))

## S4 method for signature 'MsBackend'
smoothed(object)

## S4 replacement method for signature 'MsBackend'
smoothed(object) <- value

## S4 method for signature 'MsBackend'
spectraData(object, columns = spectraVariables(object))

## S4 replacement method for signature 'MsBackend'
spectraData(object) <- value

## S4 method for signature 'MsBackend'
spectraNames(object)

## S4 replacement method for signature 'MsBackend'
spectraNames(object) <- value

## S4 method for signature 'MsBackend'
spectraVariables(object)

## S4 method for signature 'MsBackend,ANY'
split(x, f, drop = FALSE, ...)

## S4 method for signature 'MsBackend'
supportsSetBackend(object, ...)

## S4 method for signature 'MsBackend'
tic(object, initial = TRUE)

## S4 method for signature 'MsBackend'
x[i, j, ..., drop = FALSE]

## S4 method for signature 'MsBackend'
x$name
```

```
## S4 replacement method for signature 'MsBackend'
x$name <- value

## S4 method for signature 'MsBackend'
x[[i, j, ...]]

## S4 replacement method for signature 'MsBackend'
x[[i, j, ...]] <- value

## S4 method for signature 'MsBackend'
uniqueMsLevels(object, ...)

## S4 method for signature 'MsBackend'
dataStorageBasePath(object)

## S4 replacement method for signature 'MsBackend'
dataStorageBasePath(object) <- value

## S4 method for signature 'MsBackend'
longForm(object, columns = spectraVariables(object))

MsBackendDataFrame()

## S4 method for signature 'MsBackendDataFrame'
backendInitialize(object, data, peaksVariables = c("mz", "intensity"), ...)

MsBackendHdf5Peaks()

MsBackendMemory()

## S4 method for signature 'MsBackendMemory'
backendInitialize(object, data, peaksVariables = c("mz", "intensity"), ...)

MsBackendMzR()
```

Arguments

object	Object extending MsBackend.
BPPARAM	for backendBpparam(): parameter object from the BiocParallel package defining the parallel processing setup. Defaults to BPPARAM = bpparam(). See BiocParallel::bpparam() for more information.
...	Additional arguments.
columns	For spectraData() accessor: optional character with column names (spectra variables) that should be included in the returned DataFrame. By default, all columns are returned. For peaksData() accessor: optional character with requested columns in the individual matrix of the returned list. Defaults to peaksVariables(object) and depends on what *peaks variables* the backend

	provides. For `longForm()`: the spectra and peaks variables that should be included in the returned `data.frame`. Defaults to `spectraVariables(object)` and is thus the union of spectra and peaks variables.
`x`	Object extending `MsBackend`.
`y`	For `cbind2()`: A `data.frame` or `DataFrame` with the spectra variables to be added to the backend. The number of rows of `y` and their order have to match the number of spectra and their order in `x`.
`value`	replacement value for `<-` methods. See individual method description or expected data type.
`onlyCore`	For `dropNaSpectraVariables()`: `logical(1)` whether only *core* spectra variables (i.e., `coreSpectraVariables()`) are evaluated for removal. For `onlyCore = TRUE` any user-added spectra variables will be retained even if they contain only missing values. Defaults to `onlyCore = FALSE`.
`i`	For `[`: `integer`, `logical` or `character` to subset the object.
`n`	for `filterAcquisitionNum()`: `integer` with the acquisition numbers to filter for.
`file`	For `filterFile()`: index or name of the file(s) to which the data should be subsetted. For `export()`: `character` of length 1 or equal to the number of spectra.
`dataOrigin`	For `filterDataOrigin()`: `character` to define which spectra to keep. For `filterAcquisitionNum()`: optionally specify if filtering should occur only for spectra of selected `dataOrigin`.
`dataStorage`	For `filterDataStorage()`: `character` to define which spectra to keep. For `filterAcquisitionNum()`: optionally specify if filtering should occur only for spectra of selected `dataStorage`.
`mz`	For `filterIsolationWindow()`: `numeric(1)` with the m/z value to filter the object. For `filterPrecursorMzRange()`: `numeric(2)` with the lower and upper m/z boundary. For `filterPrecursorMzValues()`: `numeric` with the m/z value(s) to filter the object.
`msLevel`	`integer` defining the MS level of the spectra to which the function should be applied. For `filterMsLevel()`: the MS level to which `object` should be subsetted.
`polarity`	For `filterPolarity()`: `integer` specifying the polarity to to subset `object`.
`ppm`	For `filterPrecursorMzValues()`: `numeric(1)` with the m/z-relative maximal acceptable difference for a m/z to be considered matching. See `MsCoreUtils::closest()` for details. For `filterValues()`: `numeric` of any length allowing to define a maximal accepted difference between user input `values` and the `spectraVariables` values. If it is not equal to the length of the value provided with parameter `spectraVariables`, `ppm[1]` will be recycled.
`tolerance`	For `filterPrecursorMzValues()`: `numeric(1)` with the maximal absolute acceptable difference for a m/z value to be considered matching. See documentation `MsCoreUtils::closest()` for details. For `filterValues()`: `numeric` accepted tolerance between the `values` and the spectra variables. Defaults to `tolerance = 0`. If it is not equal to the length of the value provided with parameter `spectraVariables`, `tolerance[1]` will be recycled.

z	For filterPrecursorCharge(): integer() with the precursor charges to be used as filter.
acquisitionNum	for filterPrecursorScan(): integer with the acquisition number of the spectra to which the object should be subsetted.
f	factor defining the grouping to split x. See split(). For filterPrecursorScan(): factor defining from which original data files the spectra derive to avoid selecting spectra from different samples/files. Defaults to f = dataOrigin(object).
spectraVariables	For selectSpectraVariables(): character with the names of the spectra variables to which the backend should be subsetted. For filterRanges() and filterValues(): character vector specifying the column(s) from spectraData(object) on which to filter the data and that correspond to the the names of the spectra variables that should be used for the filtering.
ranges	for filterRanges(): A numeric vector of paired values (upper and lower boundary) that define the ranges to filter the object. These paired values need to be in the same order as the spectraVariables parameter (see below).
match	For filterRanges() and filterValues(): character(1) defining whether the condition has to match for all provided ranges/values (match = "all"; the default), or for any of them (match = "any") for spectra to be retained.
rt	for filterRt(): numeric(2) defining the retention time range to be used to subset/filter object.
msLevel.	same as msLevel above.
values	For filterValues(): A numeric vector that define the values to filter the object. values needs to be of same length than parameter spectraVariables and in the same order.
use.names	For lengths(): whether spectrum names should be used.
drop	For [: not considered.
initial	For tic(): logical(1) whether the initially reported total ion current should be reported, or whether the total ion current should be (re)calculated on the actual data (initial = FALSE).
j	For [: not supported.
name	For $ and $<-: the name of the spectra variable to return or set.
data	For backendInitialize(): DataFrame with spectrum metadata/data. This parameter can be empty for MsBackendMzR backends but needs to be provided for MsBackendDataFrame backends.
peaksVariables	For backendInitialize() for MsBackendMemory: character specifying which of the columns of the provided data contain *peaks variables* (i.e. information for individual mass peaks). Defaults to peaksVariables = c("mz", "intensity"). "mz" and "intensity" should **always** be specified.

Value

See documentation of respective function.

Implementation notes

Backends extending MsBackend **must** implement all of its methods (listed above). Developers of new MsBackends should follow the MsBackendMemory implementation. To ensure a new implementation being conform with the MsBackend definition, developers should included test suites provided by this package in their unit test setup. For that a variable be should be created in the package's "testthat.R" file that represents a (initialized) instance of the developed backend. Then the path to the test suites should be defined with test_suite <- system.file("test_backends", "test_MsBackend", package = "Spectra") followed by test_dir(test_suite) to run all test files in that directory. Individual unit test files could be run with test_file(file.path(test_suite, "test_spectra_variables.R"), stop_on_failure = TRUE) (note that without stop_on_failure = TRUE tests would fail silently) . Adding this code to the packages "testthat.R" file ensures that all tests checking the validity of an MsBackend instance defined in the Spectra package are also run on the newly develped backend class.

The MsBackend defines the following slots:

- @readonly: logical(1) whether the backend supports writing/replacing of m/z or intensity values.

Backends extending MsBackend **must** implement all of its methods (listed above). Developers of new MsBackends should follow the MsBackendDataFrame implementation.

The MsBackendCached() backend provides a caching mechanism to allow *read only* backends to add or change spectra variables. This backend shouldn't be used on its own, but is meant to be extended. See MsBackendCached() for details.

The MsBackend defines the following slots:

- @readonly: logical(1) whether the backend supports writing/replacing of m/z or intensity values.

Backend functions

New backend classes **must** extend the base MsBackend class will have to implement some of the following methods (see the MsBackend vignette for detailed description and examples):

- [: subset the backend. Only subsetting by element (*row*/i) is allowed. Parameter i should support integer indices and logical and should throw an error if i is out of bounds. The MsCoreUtils::i2index could be used to check the input i. For i = integer() an empty backend should be returned. Implementation of this method is optional, as the default calls the extractByIndex() method (which has to be implemented as the main subsetting method).
- $, $<-: access or set/add a single spectrum variable (column) in the backend. Using a value of NULL should allow deleting the specified spectra variable. An error should be thrown if the spectra variable is not available.
- [[, [[<-: access or set/add a single spectrum variable (column) in the backend. The default implementation uses $, thus these methods don't have to be implemented for new classes extending MsBackend.
- acquisitionNum(): returns the acquisition number of each spectrum. Returns an integer of length equal to the number of spectra (with NA_integer_ if not available).

- backendBpparam(): return the parallel processing setup supported by the backend class. This function can be used by any higher level function to evaluate whether the provided parallel processing setup (or the default one returned by bpparam()) is supported by the backend. Backends not supporting parallel processing (e.g. because they contain a connection to a database that can not be shared across processes) should extend this method to return only SerialParam() and hence disable parallel processing for (most) methods and functions. See also backendParallelFactor() for a function to provide a preferred splitting of the backend for parallel processing.
- backendInitialize(): initialises the backend. This method is supposed to be called rights after creating an instance of the backend class and should prepare the backend (e.g. set the data for the memory backend or read the spectra header data for the MsBackendMzR backend). Parameters can be defined freely for each backend, depending on what is needed to initialize the backend. It is however suggested to also support a parameter data that can be used to submit the full spectra data as a DataFrame to the backend. This would allow the backend to be also usable for the setBackend() function from Spectra. Note that eventually (for *read-only* backends) also the supportsSetBackend method would need to be implemented to return TRUE. The backendInitialize() method has also to ensure to correctly set spectra variable dataStorage.
- backendMerge(): merges (combines) MsBackend objects into a single instance. All objects to be merged have to be of the same type (e.g. MsBackendDataFrame()).
- backendParallelFactor(): returns a factor defining an optimal (preferred) way how the backend can be split for parallel processing used for all peak data accessor or data manipulation functions. The default implementation returns a factor of length 0 (factor()) providing thus no default splitting. backendParallelFactor() for MsBackendMzR on the other hand returns factor(dataStorage(object)) hence suggesting to split the object by data file.
- backendRequiredSpectraVariables(): returns a character with spectra variable names that are mandatory for a specific backend. The default returns an empty character(). The implementation for MsBackendMzR returns c("dataStorage", "scanIndex") as these two spectra variables are required to load the MS data on-the-fly. This method needs only to be implemented if a backend requires specific variables to be defined.
- cbind2(): allows to appends multiple new spectra variables to the backend at once. The values for the new spectra variables have to be in the same order as the spectra in x. Replacing existing spectra variables is not supported through this function. For a more controlled way of adding spectra variables, the joinSpectraData() should be used.
- centroided(), centroided<-: gets or sets the centroiding information of the spectra. centroided() returns a logical vector of length equal to the number of spectra with TRUE if a spectrum is centroided, FALSE if it is in profile mode and NA if it is undefined. See also isCentroided() for estimating from the spectrum data whether the spectrum is centroided. value for centroided<- is either a single logical or a logical of length equal to the number of spectra in object.
- collisionEnergy(), collisionEnergy<-: gets or sets the collision energy for all spectra in object. collisionEnergy() returns a numeric with length equal to the number of spectra (NA_real_ if not present/defined), collisionEnergy<- takes a numeric of length equal to the number of spectra in object.
- dataOrigin(): gets a character of length equal to the number of spectra in object with the *data origin* of each spectrum. This could e.g. be the mzML file from which the data was read.

- dataStorage(): gets a character of length equal to the number of spectra in object with the data storage of each spectrum. Note that missing values (NA_character_) are not supported for dataStorage.
- dataStorageBasePath(), dataStorageBasePath<-: gets or sets the common *base* path of the directory c acterof length 1. Most backends (such as for example theMsBackendMemorywill not support this functi BackendMzR, this function allows to get or change the path to the directory containing the original BackendMzR‘ instance gets copied to another computer or file system.
- dropNaSpectraVariables(): removes spectra variables (i.e. columns in the object's spectraData that contain only missing values (NA). Note that while columns with only NAs are removed, a spectraData() call after dropNaSpectraVariables() might still show columns containing NA values for *core* spectra variables. With parameter onlyCore = TRUE only *core* spectra variables are evaluated for removal. Any other spectra variable added by the user with only NA values will be retained.
- export(): exports data from a Spectra class to a file. This method is called by the export,Spectra method that passes itself as a second argument to the function. The export,MsBackend implementation is thus expected to take a Spectra class as second argument from which all data is exported. Taking data from a Spectra class ensures that also all eventual data manipulations (cached in the Spectra's lazy evaluation queue) are applied prior to export - this would not be possible with only a MsBackend class. An example implementation is the export() method for the MsBackendMzR backend that supports export of the data in *mzML* or *mzXML* format. See the documentation for the MsBackendMzR class below for more information.
- extractByIndex(): function to subset a backend to selected elements defined by the provided index. Similar to [, this method should allow extracting (or to subset) the data in any order. In contrast to [, however, i is expected to be an integer (while [should also support logical and eventually character). While being apparently redundant to [, this methods avoids package namespace errors/problems that can result in implementations of [being not found by R (which can happen sometimes in parallel processing using the BiocParallel::SnowParam()). This method is used internally by Spectra to extract/subset its backend. Implementation of this method is mandatory.
- filterAcquisitionNum(): filters the object keeping only spectra matching the provided acquisition numbers (argument n). If dataOrigin or dataStorage is also provided, object is subsetted to the spectra with an acquisition number equal to n **in spectra with matching dataOrigin or dataStorage values** retaining all other spectra.
- filterDataOrigin(): filters the object retaining spectra matching the provided dataOrigin. Parameter dataOrigin has to be of type character and needs to match exactly the data origin value of the spectra to subset. filterDataOrigin() should return the data ordered by the provided dataOrigin parameter, i.e. if dataOrigin = c("2", "1") was provided, the spectra in the resulting object should be ordered accordingly (first spectra from data origin "2" and then from "1"). Implementation of this method is optional since a default implementation for MsBackend is available.
- filterDataStorage(): filters the object retaining spectra matching the provided dataStorage. Parameter dataStorage has to be of type character and needs to match exactly the data storage value of the spectra to subset. filterDataStorage() should return the data ordered by the provided dataStorage parameter, i.e. if dataStorage = c("2", "1") was provided, the spectra in the resulting object should be ordered accordingly (first spectra from data storage "2" and then from "1"). Implementation of this method is optional since a default implementation for MsBackend is available.

- `filterEmptySpectra()`: removes empty spectra (i.e. spectra without peaks). Implementation of this method is optional since a default implementation for `MsBackend` is available.
- `filterFile()`: retains data of files matching the file index or file name provided with parameter `file`.
- `filterIsolationWindow()`: retains spectra that contain `mz` in their isolation window m/z range (i.e. with an `isolationWindowLowerMz <= mz` and `isolationWindowUpperMz >= mz`. Implementation of this method is optional since a default implementation for `MsBackend` is available.
- `filterMsLevel()`: retains spectra of MS level `msLevel`. Implementation of this method is optional since a default implementation for `MsBackend` is available.
- `filterPolarity()`: retains spectra of polarity `polarity`. Implementation of this method is optional since a default implementation for `MsBackend` is available.
- `filterPrecursorMzRange()` (previously `filterPrecursorMz`): retains spectra with a precursor m/z within the provided m/z range. Implementation of this method is optional since a default implementation for `MsBackend` is available.
- `filterPrecursorMzValues()`: retains spectra with a precursor m/z matching any of the provided m/z values (given `ppm` and `tolerance`). Implementation of this method is optional since a default implementation for `MsBackend` is available.
- `filterPrecursorCharge()`: retains spectra with the defined precursor charge(s). Implementation of this method is optional since a default implementation for `MsBackend` is available.
- `filterPrecursorScan()`: retains parent (e.g. MS1) and children scans (e.g. MS2) of acquisition number `acquisitionNum`. Parameter `f` is supposed to define the origin of the spectra (i.e. the original data file) to ensure related spectra from the same file/sample are selected and retained. Implementation of this method is optional since a default implementation for `MsBackend` is available.
- `filterRanges()`: allows filtering of the `Spectra` object based on user defined *numeric* ranges (parameter `ranges`) for one or more available spectra variables in object (spectra variable names can be specified with parameter `spectraVariables`). Spectra for which the value of a spectra variable is within it's defined range are retained. If multiple ranges/spectra variables are defined, the `match` parameter can be used to specify whether all conditions (`match = "all"`; the default) or if any of the conditions must match (`match = "any"`; all spectra for which values are within any of the provided ranges are retained). Implementation of this method is optional since a default implementation for `MsBackend` is available.
- `filterRt()`: retains spectra of MS level `msLevel` with retention times within (`>=`) `rt[1]` and (`<=`) `rt[2]`. The filter is applied to all spectra if no MS level is specified (the default, `msLevel. = integer()`). Implementation of this method is optional since a default implementation for `MsBackend` is available.
- `filterValues()`: allows filtering of the `Spectra` object based on similarities of *numeric* values of one or more `spectraVariables(object)` (parameter `spectraVariables`) to provided values (parameter `values`) given acceptable differences (parameters tolerance and ppm). If multiple values/spectra variables are defined, the `match` parameter can be used to specify whether all conditions (`match = "all"`; the default) or if any of the conditions must match (`match = "any"`; all spectra for which values are within any of the provided ranges are retained). Implementation of this method is optional since a default implementation for `MsBackend` is available.

- `intensity()`: gets the intensity values from the spectra. Returns a `IRanges::NumericList()` of `numeric` vectors (intensity values for each spectrum). The length of the `list` is equal to the number of `spectra` in `object`.
- `intensity<-`: replaces the intensity values. `value` has to be a `list` (or `IRanges::NumericList()`) of length equal to the number of spectra and the number of values within each list element identical to the number of peaks in each spectrum (i.e. the `lengths(x)`). Note that just writeable backends support this method.
- `ionCount()`: returns a `numeric` with the sum of intensities for each spectrum. If the spectrum is empty (see `isEmpty()`), `NA_real_` is returned.
- `isCentroided()`: a heuristic approach assessing if the spectra in `object` are in profile or centroided mode. The function takes the `qtl` th quantile top peaks, then calculates the difference between adjacent m/z value and returns `TRUE` if the first quartile is greater than `k`. (See `Spectra:::.peaks_is_centroided` for the code.)
- `isEmpty()`: checks whether a spectrum in `object` is empty (i.e. does not contain any peaks). Returns a `logical` vector of length equal number of spectra.
- `isolationWindowLowerMz()`, `isolationWindowLowerMz<-`: gets or sets the lower m/z boundary of the isolation window.
- `isolationWindowTargetMz()`, `isolationWindowTargetMz<-`: gets or sets the target m/z of the isolation window.
- `isolationWindowUpperMz()`, `isolationWindowUpperMz<-`: gets or sets the upper m/z boundary of the isolation window.
- `isReadOnly()`: returns a `logical(1)` whether the backend is *read only* or does allow also to write/update data.
- `length()`: returns the number of spectra in the object.
- `lengths()`: gets the number of peaks (m/z-intensity values) per spectrum. Returns an `integer` vector (length equal to the number of spectra). For empty spectra, `0` is returned.
- `longForm()`: extract the MS data in *long form*, i.e., as a `data.frame` with columns being requested spectra and peak variables and one row per mass peak. Parameter `columns` can be used to specify the columns (i.e., spectra or peaks variables) that should be returned. The default is `columns = spectraVariables(object)` and **all** spectra and peak variables are returned. It is strongly suggested to extract only selected columns and not the full data to avoid potential out-of-memory problems. Implementation of this method is optional as a default implementation for `MsBackend` is available which converts the `DataFrame` returned by `spectraData()` into long form.
- `msLevel()`: gets the spectra's MS level. Returns an `integer` vector (of length equal to the number of spectra) with the MS level for each spectrum (or `NA_integer_` if not available).
- `msLevel<-`: replaces the spectra's MS level.
- `mz()`: gets the mass-to-charge ratios (m/z) from the spectra. Returns a `IRanges::NumericList()` or length equal to the number of spectra, each element a `numeric` vector with the m/z values of one spectrum.
- `mz<-`: replaces the m/z values. `value` has to be a `list` of length equal to the number of spectra and the number of values within each list element identical to the number of peaks in each spectrum (i.e. the `lengths(x)`). Note that just writeable backends support this method.

- `polarity()`, `polarity<-`: gets or sets the polarity for each spectrum. `polarity()` returns an `integer` vector (length equal to the number of spectra), with `0` and `1` representing negative and positive polarities, respectively. `polarity<-` expects an integer vector of length 1 or equal to the number of spectra.
- `precursorCharge()`, `precursorIntensity()`, `precursorMz()`, `precScanNum()`, `precAcquisitionNum()`: get the charge (`integer`), intensity (`numeric`), m/z (`numeric`), scan index (`integer`) and acquisition number (`interger`) of the precursor for MS level 2 and above spectra from the object. Returns a vector of length equal to the number of spectra in `object`. `NA` are reported for MS1 spectra of if no precursor information is available.
- `peaksData()` returns a `list` with the spectras' peak data, i.e. m/z and intensity values or other *peak variables*. The length of the list is equal to the number of spectra in `object`. Each element of the list has to be a two-dimensional array (`matrix` or `data.frame`) with columns depending on the provided `columns` parameter (by default `"mz"` and `"intensity"`, but depends on the backend's available `peaksVariables`). For an empty spectrum, a `matrix` (`data.frame`) with 0 rows and columns according to `columns` is returned. The optional parameter `columns`, if supported by the backend, allows to define which peak variables should be returned in the `numeric` peak `matrix`. As a default `c("mz", "intensity")` should be used.
- `peaksData<-` replaces the peak data (m/z and intensity values) of the backend. This method expects a `list` of two dimensional arrays (`matrix` or `data.frame`) with columns representing the peak variables. All existing peaks data is expected to be replaced with these new values. The length of the `list` has to match the number of spectra of `object`. Note that only writeable backends need to support this method.
- `peaksVariables()`: lists the available variables for mass peaks. Default peak variables are `"mz"` and `"intensity"` (which all backends need to support and provide), but some backends might provide additional variables. All these variables are expected to be returned (if requested) by the `peaksData()` function.
- `reset()` a backend (if supported). This method will be called on the backend by the `reset,Spectra` method that is supposed to restore the data to its original state (see `reset,Spectra` for more details). The function returns the *reset* backend. The default implementation for `MsBackend` returns the backend as-is.
- `rtime()`, `rtime<-`: gets or sets the retention times for each spectrum (in seconds). `rtime()` returns a `numeric` vector (length equal to the number of spectra) with the retention time for each spectrum. `rtime<-` expects a numeric vector with length equal to the number of spectra.
- `scanIndex()`: returns an `integer` vector with the *scan index* for each spectrum. This represents the relative index of the spectrum within each file. Note that this can be different to the `acquisitionNum()` of the spectrum which is the index of the spectrum as reported in the mzML file.
- `selectSpectraVariables()`: reduces the information within the backend to the selected spectra variables. It is suggested to **not** remove values for the `"dataStorage"` variable, since this might be required for some backends to work properly (such as the `MsBackendMzR`).
- `smoothed()`,`smoothed<-`: gets or sets whether a spectrum is *smoothed*. `smoothed()` returns a `logical` vector of length equal to the number of spectra. `smoothed<-` takes a `logical` vector of length 1 or equal to the number of spectra in `object`.
- `spectraData()`, `spectraData<-`: gets or sets general spectrum metadata (annotation, also called header). `spectraData()` returns a `DataFrame`, `spectraData<-` expects a `DataFrame`

with the same number of rows as there are spectra in `object`. Note that `spectraData()` has to return the full data, i.e. also the m/z and intensity values (as a `list` or `SimpleList` in columns `"mz"` and `"intensity"`. See also `fillCoreSpectraVariables()` for a function that can *complete* a spectra data data frame with eventually missing *core* spectra variables.

- `spectraNames()`: returns a `character` vector with the names of the spectra in `object` or `NULL` if not set. `spectraNames<-` allows to set spectra names (if the object is not read-only).
- `spectraVariables()`: returns a `character` vector with the available spectra variables (columns, fields or attributes) available in `object`. This should return **all** spectra variables which are present in `object`, also `"mz"` and `"intensity"` (which are by default not returned by the `spectraVariables,Spectra` method).
- `split()`: splits the backend into a `list` of backends (depending on parameter `f`). The default method for `MsBackend` uses `split.default()`, thus backends extending `MsBackend` don't necessarily need to implement this method.
- `supportsSetBackend()`: whether a `MsBackend` supports the `Spectra` `setBackend()` function. For a `MsBackend` to support `setBackend()` it needs to have a parameter called `data` in its `backendInitialize()` method that support receiving all spectra data as a `DataFrame` from another backend and to initialize the backend with this data. In general *read-only* backends do not support `setBackend()` hence, the default implementation of `supportsSetBackend()` returns `!isReadOnly(object)`. If a read-only backend would support the `setBackend()` and being initialized with a `DataFrame` an implementation of this method for that backend could be defined that returns `TRUE` (see also the `MsBackend` vignette for details and examples).
- `tic()`: gets the total ion current/count (sum of signal of a spectrum) for all spectra in `object`. By default, the value reported in the original raw data file is returned. For an empty spectrum, `NA_real_` is returned.
- `uniqueMsLevels()`: gets the unique MS levels of all spectra in `object`. The default implementation calls `unique(msLevel(object))` but more efficient implementations could be defined for specific backends.

Subsetting and merging backend classes

Backend classes must support (implement) the `[` method to subset the object. This method should only support subsetting by spectra (rows, `i`) and has to return a `MsBackend` class.

Backends extending `MsBackend` should also implement the `backendMerge()` method to support combining backend instances (only backend classes of the same type should be merged). Merging should follow the following rules:

- The whole spectrum data of the various objects should be merged. The resulting merged object should contain the union of the individual objects' spectra variables (columns/fields), with eventually missing variables in one object being filled with `NA`.

In-memory data backends

`MsBackendMemory` and `MsBackendDataFrame`:

The `MsBackendMemory` and `MsBackendDataFrame` objects keep all MS data in memory are thus ideal for fast data processing. Due to their large memory footprint they are however not suited for large scale experiments. The two backends store the data different. The `MsBackendDataFrame` stores all data in a `DataFrame` and thus supports also S4-classes as spectra variables. Also, sepratate

access to m/z or intensity values (i.e. using the `mz()` and `intensity()` methods) is faster for the `MsBackendDataFrame`. The `MsBackendMemory` on the other hand, due to the way the data is organized internally, provides much faster access to the full peak data (i.e. the numerical matrices of m/z and intensity values). Also subsetting and access to any spectra variable (except `"mz"` and `"intensity"`) is fastest for the `MsBackendMemory`.

Thus, for most use cases, the `MsBackendMemory` provides a higher performance and flexibility than the `MsBackendDataFrame` and should thus be preferred. See also issue 246 for a performance comparison.

New objects can be created with the `MsBackendMemory()` and `MsBackendDataFrame()` function, respectively. Both backends can be subsequently initialized with the `backendInitialize()` method, taking a `DataFrame` (or `data.frame`) with the (full) MS data as first parameter `data`. The second parameter `peaksVariables` allows to define which columns in `data` contain *peak variables* such as the m/z and intensity values of individual peaks per spectrum. The default for this parameter is `peaksVariables = c("mz", "intensity")`. Note that it is not supported to provide either `"mz"` or `"intensity"`, if provided, both need to be present in the data frame. Alternatively, the function also supports a data frame without m/z and intensity values, in which case a `Spectra` without mass peaks is created.

Suggested columns of this `DataFrame` are:

- `"msLevel"`: `integer` with MS levels of the spectra.
- `"rt"`: `numeric` with retention times of the spectra.
- `"acquisitionNum"`: `integer` with the acquisition number of the spectrum.
- `"scanIndex"`: `integer` with the index of the scan/spectrum within the *mzML/mzXML/CDF* file.
- `"dataOrigin"`: `character` defining the *data origin*.
- `"dataStorage"`: `character` indicating grouping of spectra in different e.g. input files. Note that missing values are not supported.
- `"centroided"`: `logical` whether the spectrum is centroided.
- `"smoothed"`: `logical` whether the spectrum was smoothed.
- `"polarity"`: `integer` with the polarity information of the spectra.
- `"precScanNum"`: `integer` specifying the index of the (MS1) spectrum containing the precursor of a (MS2) spectrum.
- `"precursorMz"`: `numeric` with the m/z value of the precursor.
- `"precursorIntensity"`: `numeric` with the intensity value of the precursor.
- `"precursorCharge"`: `integer` with the charge of the precursor.
- `"collisionEnergy"`: `numeric` with the collision energy.
- `"mz"`: `IRanges::NumericList()` of `numeric` vectors representing the m/z values for each spectrum.
- `"intensity"`: `IRanges::NumericList()` of `numeric` vectors representing the intensity values for each spectrum.

Additional columns are allowed too.

The `peaksData()` function for `MsBackendMemory` and `MsBackendDataFrame` returns a `list` of `numeric matrix` by default (with parameter `columns = c("mz", "intensity")`). If other peak variables are requested, a `list` of `data.frame` is returned (ensuring m/z and intensity values are always `numeric`).

MsBackendMzR, on-disk MS data backend

The MsBackendMzR keeps only a limited amount of data in memory, while the spectra data (m/z and intensity values) are fetched from the raw files on-demand. This backend uses the mzR package for data import and retrieval and hence requires that package to be installed. Also, it can only be used to import and represent data stored in *mzML*, *mzXML* and *CDF* files.

The MsBackendMzR backend extends the MsBackendDataFrame backend using its DataFrame to keep spectra variables (except m/z and intensity) in memory.

New objects can be created with the MsBackendMzR() function which can be subsequently filled with data by calling backendInitialize() passing the file names of the input data files with argument files.

This backend provides an export() method to export data from a Spectra in *mzML* or *mzXML* format. The definition of the function is:

```
export(object, x, file = tempfile(), format = c("mzML", "mzXML"), copy = FALSE)
```

The parameters are:

- object: an instance of the MsBackendMzR class.
- x: the Spectra object to be exported.
- file: character with the (full) output file name(s). Should be of length 1 or equal length(x). If a single file is specified, all spectra are exported to that file. Alternatively it is possible to specify for each spectrum in x the name of the file to which it should be exported (and hence file has to be of length equal length(x)).
- format: character(1), either "mzML" or "mzXML" defining the output file format.
- copy: logical(1) whether general file information should be copied from the original MS data files. This only works if x uses a MsBackendMzR backend and if dataOrigin(x) contains the original MS data file names.
- BPPARAM: parallel processing settings.

See examples in Spectra or the vignette for more details and examples.

The MsBackendMzR ignores parameter columns of the peaksData() function and returns **always** m/z and intensity values.

MsBackendHdf5Peaks, on-disk MS data backend

The MsBackendHdf5Peaks keeps, similar to the MsBackendMzR, peak data (i.e. m/z and intensity values) in custom data files (in HDF5 format) on disk while the remaining spectra variables are kept in memory. This backend supports updating and writing of manipulated peak data to the data files.

New objects can be created with the MsBackendHdf5Peaks() function which can be subsequently filled with data by calling the object's backendInitialize() method passing the desired file names of the HDF5 data files along with the spectra variables in form of a DataFrame (see MsBackendDataFrame for the expected format). An optional parameter hdf5path allows to specify the folder where the HDF5 data files should be stored to. If provided, this is added as the path to the submitted file names (parameter files).

By default backendInitialize() will store all peak data into a single HDF5 file which name has to be provided with the parameter files. To store peak data across several HDF5 files data has to contain a column "dataStorage" that defines the grouping of spectra/peaks into files: peaks for

spectra with the same value in "dataStorage" are saved into the same HDF5 file. If parameter files is omitted, the value in dataStorage is used as file name (replacing any file ending with ".h5". To specify the file names, files' length has to match the number of unique elements in "dataStorage".

For details see examples on the Spectra() help page.

The MsBackendHdf5Peaks ignores parameter columns of the peaksData() function and returns **always** m/z and intensity values.

Author(s)

Johannes Rainer, Sebastian Gibb, Laurent Gatto, Philippine Louail

Examples

```
## The MsBackend class is a virtual class and can not be instantiated
## directly. Below we define a new backend class extending this virtual
## class
MsBackendDummy <- setClass("MsBackendDummy", contains = "MsBackend")
MsBackendDummy()

## This class inherits now all methods from `MsBackend`, all of which
## however throw an error. These methods would have to be implemented
## for the new backend class.
try(mz(MsBackendDummy()))

## See `MsBackendDataFrame` as a reference implementation for a backend
## class (in the *R/MsBackendDataFrame.R* file).

## MsBackendDataFrame
##
## The `MsBackendDataFrame` uses a `S4Vectors::DataFrame` to store all MS
## data. Below we create such a backend by passing a `DataFrame` with all
## data to it.
data <- DataFrame(msLevel = c(1L, 2L, 1L), scanIndex = 1:3)
data$mz <- list(c(1.1, 1.2, 1.3), c(1.4, 54.2, 56.4, 122.1), c(15.3, 23.2))
data$intensity <- list(c(3, 2, 3), c(45, 100, 12.2, 1), c(123, 12324.2))

## Backends are supposed to be created with their specific constructor
## function
be <- MsBackendDataFrame()

be

## The `backendInitialize()` method initializes the backend filling it with
## data. This method can take any parameters needed for the backend to
## get loaded with the data (e.g. a file name from which to load the data,
## a database connection or, in this case, a data frame containing the data).
be <- backendInitialize(be, data)

be
```

```
## Data can be accessed with the accessor methods
msLevel(be)

mz(be)

## Even if no data was provided for all spectra variables, its accessor
## methods are supposed to return a value.
precursorMz(be)

## The `peaksData()` method is supposed to return the peaks of the spectra as
## a `list`.
peaksData(be)

## List available peaks variables
peaksVariables(be)

## Use columns to extract specific peaks variables. Below we extract m/z and
## intensity values, but in reversed order to the default.
peaksData(be, columns = c("intensity", "mz"))

## List available spectra variables (i.e. spectrum metadata)
spectraVariables(be)

## Extract precursor m/z, rtime, MS level spectra variables
spectraData(be, c("precursorMz", "rtime", "msLevel"))

## MsBackendMemory
##
## The `MsBackendMemory` uses a more efficient internal data organization
## and allows also adding arbitrary additional peaks variables (annotations)
## Below we thus add a column "peak_ann" with arbitrary names/ids for each
## peak and add the name of this column to the `peaksVariables` parameter
## of the `backendInitialize()` method (in addition to `"mz"` and
## `"intensity"` that should **always** be specified.
data$peak_ann <- list(c("a", "", "d"), c("", "d", "e", "f"), c("h", "i"))
be <- backendInitialize(MsBackendMemory(), data,
    peaksVariables = c("mz", "intensity", "peak_ann"))
be

spectraVariables(be)

## peak_ann is also listed as a peaks variable
peaksVariables(be)

## The additional peaks variable can be accessed using the `peaksData()`
## function
peaksData(be, "peak_ann")

## The $<- method can be used to replace values of an existing peaks
## variable. It is important that the number of elements matches the
## number of peaks per spectrum.
be$peak_ann <- list(1:3, 1:4, 1:2)
```

```
## A peaks variable can again be removed by setting it to NULL
be$peak_ann <- NULL

peaksVariables(be)
```

MsBackendCached *Base MsBackend class providing data caching mechanism*

Description

The MsBackendCached class is a rudimentary implementation of the MsBackend providing a simple mechanism to cache spectra data locally. This class is thought to be used as a base class for other MsBackend implementations to reuse its caching mechanism and avoid having to re-implement commonly used methods. This class is thus not thought to be used directly by a user.

The MsBackendCached caching mechanism allows MsBackend instances to add or replace spectra variables even if the backend used by them does not allow to alter values (e.g. if a SQL database is used as a backend). Any replacement operation with $<- will add the specified values to a local data.frame within the MsBackendCached class that allows to *cache* these values (increasing obviously the memory demand of the object).

Any data accessor functions of the extending MsBackend class (such as $ or msLevel() or spectraData()) should first use callNextMethod() to call the respective accessor of MsBackendCached that will evaluate if the requested spectra variable(s) are in the local cache and return these. If the requested spectra variables are neither in the local cache, nor listed in the @spectraVariables slot (which defines all spectra variables that can be requested from the extending MsBackend class) but are *core spectra variables* then missing values of the correct data type are returned.

Usage

```
MsBackendCached()

## S4 method for signature 'MsBackendCached'
backendInitialize(
  object,
  data = data.frame(),
  nspectra = 0L,
  spectraVariables = character(),
  ...
)

## S4 method for signature 'MsBackendCached'
dataStorage(object)

## S4 method for signature 'MsBackendCached,ANY'
extractByIndex(object, i)

## S4 method for signature 'MsBackendCached'
length(x)
```

```
## S4 method for signature 'MsBackendCached'
spectraVariables(object)

## S4 method for signature 'MsBackendCached'
spectraData(object, columns = spectraVariables(object))

## S4 replacement method for signature 'MsBackendCached'
spectraData(object) <- value

## S4 method for signature 'MsBackendCached'
x[i, j, ..., drop = FALSE]

## S4 method for signature 'MsBackendCached'
x$name

## S4 replacement method for signature 'MsBackendCached'
x$name <- value

## S4 method for signature 'MsBackendCached'
selectSpectraVariables(object, spectraVariables = spectraVariables(object))

## S4 method for signature 'MsBackendCached'
show(object)

## S4 method for signature 'MsBackendCached'
intensity(object)

## S4 method for signature 'MsBackendCached'
ionCount(object)

## S4 method for signature 'MsBackendCached'
mz(object)
```

Arguments

object	A MsBackendCached object.
data	For backendInitialize(): (optional) data.frame with cached values. The number of rows (and their order) has to match the number of spectra.
nspectra	For backendInitialize(): integer with the number of spectra.
spectraVariables	For backendInitialize(): character with the names of the spectra variables that are provided by the extending backend. For selectSpectraVariables(): character specifying the spectra variables to keep.
...	ignored
i	For [: integer with the indices to subset the object.
x	A MsBackendCached object.

columns	For spectraData(): character with the names of the spectra variables to retrieve.
value	replacement value for <- methods. See individual method description or expected data type.
j	For [: ignored.
drop	For [: not considered.
name	For $<-: the name of the spectra variable to set.

Value

See documentation of respective function.

Implementation notes

Classes extending the MsBackendCached need to

- call the backendInitialize() method of this class in their own backendInitialize() method and set at least the number of spectra with the nspectra parameter and the spectraVariables that are available to the (extending) backend class.
- implement the spectraData() method that also calls the spectraData() method from MsBackendCached to also retrieve cached values (e.g. using res <- callNextMethod() at the beginning of the spectraData function). The spectraData,MsBackendCached method will return NULL if the selected spectra variables were not cached and are not *core spectra variables* not being provided by the extending backend. Thus, the extending backend can then proceed to retrieve the respective values from its own backend/data storage.
- implement eventually the [method that calls in addition the [from the MsBackendCached.

All other methods accessing or setting spectra variables don't need to be implemented by the extending backend class (the default implementations of the MsBackendCached will then be used instead; these ensure that cached values are returned first). Spectra variables can be modified or added using the $<- method of the MsBackendCached. Replacing or adding multiple variables using the spectraData<- is not supported by MsBackendCached. The extending backend might however implement such a method that internally uses $<- to add/replace single variables.

The MsBackendCached has the following slots:

- nspectra: integer(1) defining the number of spectra of the backend. This variable needs to be set and must match the number of rows of localData and the actual number of spectra in the (extending) backend.
- localData: data.frame with the cached local data. Any replacement operation with $<- will set/add a column with the respective values.
- spectraVariables: character defining the spectra variables that are provided by the extending MsBackend class (e.g. all spectra variables that can be retrieved from the data base or original data files).

Available methods

- `acquisitionNum()`: returns the acquisition number of each spectrum. Returns an `integer` of length equal to the number of spectra (with `NA_integer_` if not available).
- `backendInitialize()`: *initializes* the backend. The method takes parameters `data` (`data.frame` with cached data), `nspectra` (`integer` defining the number of spectra) and `spectraVariables` (`character` with the spectra variables that are provided by the extending backend.
- `centroided()`, `centroided<-`: gets or sets the centroiding information of the spectra. `centroided` returns a `logical` vector of length equal to the number of spectra with `TRUE` if a spectrum is centroided, `FALSE` if it is in profile mode and `NA` if it is undefined. See also `isCentroided` for estimating from the spectrum data whether the spectrum is centroided. `value` for `centroided<-` is either a single `logical` or a `logical` of length equal to the number of spectra in `object`.
- `collisionEnergy()`, `collisionEnergy<-`: gets or sets the collision energy for all spectra in `object`. `collisionEnergy()` returns a `numeric` with length equal to the number of spectra (`NA_real_` if not present/defined), `collisionEnergy<-` takes a `numeric` of length equal to the number of spectra in `object`.
- `dataOrigin()`: gets a `character` of length equal to the number of spectra in `object` with the *data origin* of each spectrum. This could e.g. be the mzML file from which the data was read.
- `intensity()`: gets the intensity values from the spectra. Returns a `IRanges::NumericList()` of `numeric` vectors (intensity values for each spectrum). The length of the `list` is equal to the number of `spectra` in `object`.
- `ionCount()`: returns a `numeric` with the sum of intensities for each spectrum. If the spectrum is empty (see `isEmpty()`), `NA_real_` is returned.
- `isEmpty()`: checks whether a spectrum in `object` is empty (i.e. does not contain any peaks). Returns a `logical` vector of length equal number of spectra.
- `isolationWindowLowerMz()`, `isolationWindowLowerMz<-`: gets or sets the lower m/z boundary of the isolation window.
- `isolationWindowTargetMz()`, `isolationWindowTargetMz<-`: gets or sets the target m/z of the isolation window.
- `isolationWindowUpperMz()`, `isolationWindowUpperMz<-`: gets or sets the upper m/z boundary of the isolation window.
- `length()`: returns the number of spectra (i.e. the `@nspectra`).
- `lengths()`: gets the number of peaks (m/z-intensity values) per spectrum. Returns an `integer` vector (length equal to the number of spectra). For empty spectra, `0` is returned.
- `msLevel()`: gets the spectra's MS level. Returns an `integer` vector (of length equal to the number of spectra) with the MS level for each spectrum (or `NA_integer_` if not available).
- `mz()`: gets the mass-to-charge ratios (m/z) from the spectra. Returns a `IRanges::NumericList()` or length equal to the number of spectra, each element a `numeric` vector with the m/z values of one spectrum.
- `polarity()`, `polarity<-`: gets or sets the polarity for each spectrum. `polarity` returns an `integer` vector (length equal to the number of spectra), with `0` and `1` representing negative and positive polarities, respectively. `polarity<-` expects an integer vector of length 1 or equal to the number of spectra.

- precursorCharge(), precursorIntensity(), precursorMz(), precScanNum(), precAcquisitionNum(): get the charge (integer), intensity (numeric), m/z (numeric), scan index (integer) and acquisition number (interger) of the precursor for MS level 2 and above spectra from the object. Returns a vector of length equal to the number of spectra in object. NA are reported for MS1 spectra of if no precursor information is available.
- rtime(), rtime<-: gets or sets the retention times for each spectrum (in seconds). rtime() returns a numeric vector (length equal to the number of spectra) with the retention time for each spectrum. rtime<- expects a numeric vector with length equal to the number of spectra.
- scanIndex(): returns an integer vector with the *scan index* for each spectrum. This represents the relative index of the spectrum within each file. Note that this can be different to the acquisitionNum() of the spectrum which is the index of the spectrum as reported in the mzML file.
- selectSpectraVariables(): subset the object to specified spectra variables. This will eventually remove spectra variables listed in @spectraVariables and will also drop columns from the local cache if not among spectraVariables.
- smoothed(),smoothed<-: gets or sets whether a spectrum is *smoothed*. smoothed() returns a logical vector of length equal to the number of spectra. smoothed<- takes a logical vector of length 1 or equal to the number of spectra in object.
- spectraVariables(): returns the available spectra variables, i.e. the unique set of *core spectra variables*, cached spectra variables and spectra variables defined in the @spectraVariables slot (i.e. spectra variables thought to be provided by the extending MsBackend instance).
- spectraData(): returns a DataFrame with cached spectra variablers or initialized *core spectra variables*. Parameter spectraVariables allows to specify the variables to retrieve. The function returns NULL if the requested variables are not cached and are not provided by the extending backend. Note that this method **only** returns cached spectra variables or core spectra variables **not** provided by the extending backend. It is the responsibility of the extending backend to add/provide these.
- [: subsets the cached data. Parameter i needs to be an integer vector.
- $, $<-: access or set/add a single spectrum variable (column) in the backend.

Author(s)

Johannes Rainer

See Also

MsBackend for the documentation of MS backends.

---

neutralLoss *Calculate Neutral Loss Spectra*

Description

This help page lists functions that convert MS/MS spectra to neutral loss spectra. The main function for this is `neutralLoss` and the specific algorithm to be used is defined (and configured) with dedicated *parameter* objects (paramer `param` of the `neutralLoss()` function).

The parameter objects for the different algorithms are:

- `PrecursorMzParam()`: calculates neutral loss spectra as in Aisporna *et al.* 2022 by subtracting the (fragment's) peak m/z value from the precursor m/z value of each spectrum (precursor m/z - fragment m/z). Parameter `msLevel` allows to restrict calculation of neutral loss spectra to specified MS level(s). Spectra from other MS level(s) are returned as-is. Parameter `filterPeaks` allows to remove certain peaks from the neutral loss spectra. By default (`filterPeaks = "none"`) no filtering takes place. With `filterPeaks = "removePrecursor"` all fragment peaks with an m/z value matching the precursor m/z (considering also `ppm` and `tolerance` are removed. With `filterPeaks = "abovePrecursor"`, all fragment peaks with an m/z larger than the precursor m/z (m/z > precursor m/z - `tolerance` - `ppm` of the precursor m/z) are removed (thus removing also in most cases the fragment peaks representing the precursor). Finally, with `filterPeaks = "belowPrecursor"` all fragment peaks with an m/z smaller than the precursor m/z (m/z < precursor m/z + `tolerance` + `ppm` of the precursor m/z) are removed. Also in this case the precursor fragment peak is (depending on the values of `ppm` and `tolerance`) removed.

Usage

```
PrecursorMzParam(
  filterPeaks = c("none", "abovePrecursor", "belowPrecursor", "removePrecursor"),
  msLevel = c(2L, NA_integer_),
  ppm = 10,
  tolerance = 0
)

## S4 method for signature 'Spectra,PrecursorMzParam'
neutralLoss(object, param, ...)
```

Arguments

`filterPeaks`	For `PrecursorMzParam()`: `character(1)` or `function` defining if and how fragment peaks should be filtered before calculation. Pre-defined options are: `"none"` (keep all peaks), `"abovePrecursor"` (removes all fragment peaks with an m/z >= precursor m/z), `"belowPrecursor"` (removes all fragment peaks with an m/z <= precursor m/z). In addition, it is possible to pass a custom function with this parameter with arguments `x` (two column peak matrix) and `precursorMz` (the precursor m/z) that returns the sub-setted two column peak matrix.
`msLevel`	`integer` defining for which MS level(s) the neutral loss spectra should be calculated. Defaults to `msLevel = c(2L, NA)` thus, neutral loss spectra will be calculated for all spectra with MS level equal to 2 or with missing/undefined MS level. All spectra with a MS level different than `msLevel` will be returned unchanged.

ppm	numeric(1) with m/z-relative acceptable difference in m/z values to filter peaks. Defaults to ppm = 10. See function description for details.
tolerance	numeric(1) with absolute acceptable difference in m/z values to filter peaks. Defaults to tolerance = 0. See function description for details.
object	Spectra() object with the fragment spectra for which neutral loss spectra should be calculated.
param	One of the *parameter* objects discussed below.
...	Currently ignored.

Value

A Spectra() object with calculated neutral loss spectra.

Note

By definition, mass peaks in a Spectra object need to be ordered by their m/z value (in increasing order). Thus, the order of the peaks in the calculated neutral loss spectra might not be the same than in the original Spectra object.

Note also that for spectra with a missing precursor m/z empty spectra are returned (i.e. spectra without peaks) since it is not possible to calcualte the neutral loss spectra.

Author(s)

Johannes Rainer

References

Aisporna A, Benton PH, Chen A, Derks RJE, Galano JM, Giera M and Siuzdak G (2022). Neutral Loss Mass Spectral Data Enhances Molecular Similarity Analysis in METLIN. Journal of the American Society for Mass Spectrometry. doi:10.1021/jasms.1c00343

See Also

addProcessing() for other data analysis and manipulation functions.

Examples

```
## Create a simple example Spectra object with some MS1, MS2 and MS3 spectra.
DF <- DataFrame(msLevel = c(1L, 2L, 3L, 1L, 2L, 3L),
                precursorMz = c(NA, 40, 20, NA, 300, 200))
DF$mz <- IRanges::NumericList(
                      c(3, 12, 14, 15, 16, 200),
                      c(13, 23, 39, 86),
                      c(5, 7, 20, 34, 50),
                      c(5, 7, 9, 20, 100),
                      c(15, 53, 299, 300),
                      c(34, 56, 100, 200, 204, 309)
                    , compress = FALSE)
DF$intensity <- IRanges::NumericList(1:6, 1:4, 1:5, 1:5, 1:4, 1:6,
```

```
                                    compress = FALSE)
sps <- Spectra(DF, backend = MsBackendDataFrame())

## Calculate neutral loss spectra for all MS2 spectra, keeping MS1 and MS3
## spectra unchanged.
sps_nl <- neutralLoss(sps, PrecursorMzParam(msLevel = 2L))
mz(sps)
mz(sps_nl)

## Calculate neutral loss spectra for MS2 and MS3 spectra, removing peaks
## with an m/z >= precursorMz
sps_nl <- neutralLoss(sps, PrecursorMzParam(
    filterPeaks = "abovePrecursor", msLevel = 2:3))
mz(sps_nl)
## This removed also the peak with m/z 39 from the second spectrum

## Removing all fragment peaks matching the precursor m/z with a tolerance
## of 1 and ppm 10
sps_nl <- neutralLoss(sps, PrecursorMzParam(
    filterPeaks = "removePrecursor", tolerance = 1, ppm = 10, msLevel = 2:3))
mz(sps_nl)

## Empty spectra are returned for MS 2 spectra with undefined precursor m/z.
sps$precursorMz <- NA_real_
sps_nl <- neutralLoss(sps, PrecursorMzParam())
mz(sps_nl)
```

`plotMzDelta` *MZ delta Quality Control*

Description

The M/Z delta plot illustrates the suitability of MS2 spectra for identification by plotting the M/Z differences of the most intense peaks. The resulting histogram should optimally show modes at amino acid residu masses. The plots have been described in Foster et al. 2011.

Only a certain percentage of most intense MS2 peaks are taken into account to use the most significant signal. Default value is 20% (see `percentage` argument). The difference between peaks is then computed for all individual spectra and their distribution is plotted as a histogram. Delta M/Z between 40 and 200 are plotted by default, to encompass the residue masses of all amino acids and several common contaminants, although this can be changes with the `mzRange` argument.

In addition to the processing described above, isobaric reporter tag peaks and the precursor peak can also be removed from the MS2 spectrum, to avoid interence with the fragment peaks.

Note that figures in Foster et al. 2011 have been produced and optimised for centroided data. While running the function on profile mode is likely fine, it is recommended to use centroided data.

A `ggplot2` based function called `ggMzDeltaPlot()` to visualise the M/Z delta distributions is available at https://gist.github.com/lgatto/c72b1ff5a4116118dbb34d9d2bc3470a.

Usage

```
computeMzDeltas(
  object,
  percentage = 0.2,
  mzRange = c(40, 200),
  BPPARAM = BiocParallel::bpparam()
)

plotMzDelta(x, aaLabels = TRUE)
```

Arguments

object	An instance of class Spectra().
percentage	numeric(1) between 0 and 1 indicating the percentage of the most intense peaks in each MS2 spectrum to include in the calculation. Default is 0.2.
mzRange	numeric(2) with the upper and lower M/Z to be used to the MZ deltas. Default is c(40, 200).
BPPARAM	An optional BiocParallelParam instance determining the parallel back-end to be used during evaluation. Default is to use BiocParallel::bpparam(). See ?BiocParallel::bpparam for details.
x	A list of M/Z delta values, as returned by computeMzDeltas().
aaLabels	logical(1) defining whether the amino acids should be labelled on the histogram. Default is TRUE.

Value

computeMzDeltas() returns a list of numeric vectors. plotMzDelta() is used to visualise of M/Z delta distributions.

Author(s)

Laurent Gatto with contributions (to MSnbase) of Guangchuang Yu.

References

Foster JM, Degroeve S, Gatto L, Visser M, Wang R, Griss J, et al. A posteriori quality control for the curation and reuse of public proteomics data. Proteomics. 2011;11: 2182-2194. http://dx.doi.org/10.1002/pmic.201000602

Examples

```
f <- MsDataHub::TMT_Erwinia_1uLSike_Top10HCD_isol2_45stepped_60min_01.20141210.mzML.gz()
sp <- Spectra(f)

d <- computeMzDeltas(sp[1:1000])
plotMzDelta(d)
```

precursorPurity *Calculating Precursor Purity for MS2 spectra*

Description

MS instruments generally collect precursor ions in a discrete *m/z isolation window* before fragmenting them and recording the respective fragment (MS2) spectrum. Ideally, only a single ion species is fragmented, depending also on the size of the isolation window, different ions (with slightly different *m/z*) might be fragmented. The resulting MS2 spectrum might thus contain fragments from different ions and hence be less *pure*.

The `precursorPurity()` function calculates the **precursor purity** of MS2 (fragment) spectra expressed as the ratio between the itensity of the highest signal in the isolation window to the sum of intensities of all MS1 peaks in the isolation window. This is similar to the calculation performed in the *msPurity* Bioconductor package.

The peak intensities within the isolation window is extracted from the last MS1 spectrum before the respective MS2 spectrum. The spectra are thus expected to be ordered by retention time. For the isolation window either the isolation window reported in the `Spectra` object is used, or it is calculated based on the MS2 spectra's precursor m/z. By default, the isolation window is calculated based on the precursor m/z and parameters `tolerance` and `ppm`: precursorMz +/- (`tolerance` + `ppm(precursorMz, ppm)`). If the actually used precursor isolation window is defined and available in the `Spectra` object, it can be used instead by setting `useReportedIsolationWindow = TRUE` (default is `useReportedIsolationWindow = FALSE`). Note that parameters `tolerance` and `ppm` are ignored for `useReportedIsolationWindow = TRUE`.

Usage

```
precursorPurity(
  object,
  tolerance = 0.05,
  ppm = 0,
  useReportedIsolationWindow = FALSE,
  BPPARAM = SerialParam()
)
```

Arguments

`object`	`Spectra()` object with LC-MS/MS data.
`tolerance`	`numeric(1)` defining an absolute value (in Da) to be used to define the isolation window. For the precursor purity calculation of an MS2 spectrum, all MS1 peaks from the previous MS1 scan with an m/z between the fragment spectrum's precursorMz +/- (tolerance + ppm(precursorMz, ppm)) are considered.
`ppm`	`numeric(1)` defining the m/z dependent acceptable difference in m/z. See documentation of parameter `tolerance` for more information.
`useReportedIsolationWindow`	`logical(1)` whether the reported isolation window, defined by spectra variables `isolationWindowLowerMz` and `isolationWindowUpperMz` in the input Spectra

object, should be used instead of calculating the isolation window from the reported precursor m/z and parameters tolerance and ppm. Only few manufacturers report the isolation window with the spectra variables isolationWindowLowerMz and isolationWindowTargetMz, thus the default for this parameter is FALSE.

BPPARAM parallel processing setup. Defaults to BPPARAM = SerialParam(). See BiocParallel::SerialParam() for more information.

Value

numeric vector of length equal to the number of spectra in object, with values representing the calculated precursor purity for each spectrum. For MS1 spectra, NA_real_ is returned. For MS2 spectra, the purity is defined as the proportion of maximum signal to the total ion current within the isolation window that is attributable to the selected precursor ion. If no matching MS1 scan is found or the precursor peak is missing, NA_real_ is returned.

Note

This approach is applicable only when fragment spectra are obtained through data-dependent acquisition (DDA), as it assumes that the peak with the highest intensity within the given isolation m/z window (from the previous MS1 spectrum) corresponds to the precursor ion.

The spectra in object have to be ordered by their retention time.

Author(s)

Ahlam Mentag, Johannes Rainer

See Also

addProcessing() for other data analysis and manipulation functions.

Examples

```
## Load a test DDA file
fl <- MsDataHub::PestMix1_DDA.mzML()
sps_dda <- Spectra(fl)

## Define the isolation window based on the MS2 spectra's precursor m/z
## and parameter `tolerance`: isolation window with size 1Da:
pp <- precursorPurity(sps_dda, tolerance = 0.5)

## values for MS1 spectra are NA
head(pp[msLevel(sps_dda) == 1])

head(pp[msLevel(sps_dda) == 2])

## Use the reported isolation window (if defined in the `Spectra`):
filterMsLevel(sps_dda, 2L) |>
    isolationWindowLowerMz() |>
    head()
filterMsLevel(sps_dda, 2L) |>
    isolationWindowUpperMz() |>
```

```
    head()

pp_2 <- precursorPurity(sps_dda, useReportedIsolationWindow = TRUE)

head(pp_2[msLevel(sps_dda) == 2])
```

`processingChunkSize,Spectra-method`
Parallel and chunk-wise processing of `Spectra`

Description

Many operations on `Spectra` objects, specifically those working with the actual MS data (peaks data), allow a chunk-wise processing in which the `Spectra` is splitted into smaller parts (chunks) that are iteratively processed. This enables parallel processing of the data (by data chunk) and also reduces the memory demand since only the MS data of the currently processed subset is loaded into memory and processed. This chunk-wise processing, which is by default disabled, can be enabled by setting the processing chunk size of a `Spectra` with the `processingChunkSize()` function to a value which is smaller than the length of the `Spectra` object. Setting `processingChunkSize(sps) <- 1000` will cause any data manipulation operation on the `sps`, such as `filterIntensity()` or `bin()`, to be performed eventually in parallel for sets of 1000 spectra in each iteration.

Such chunk-wise processing is specifically useful for `Spectra` objects using an *on-disk* backend or for very large experiments. For small data sets or `Spectra` using an in-memory backend, a direct processing might however be more efficient. Setting the chunk size to `Inf` will disable the chunk-wise processing.

For some backends a certain type of splitting and chunk-wise processing might be preferable. The `MsBackendMzR` backend for example needs to load the MS data from the original (mzML) files, hence chunk-wise processing on a per-file basis would be ideal. The `backendParallelFactor()` function for `MsBackend` allows backends to suggest a preferred splitting of the data by returning a `factor` defining the respective data chunks. The `MsBackendMzR` returns for example a `factor` based on the *dataStorage* spectra variable. A `factor` of length 0 is returned if no particular preferred splitting should be performed. The suggested chunk definition will be used if no finite `processingChunkSize()` is defined. Setting the `processingChunkSize` overrides `backendParallelFactor`.

See the *Large-scale data handling and processing with Spectra* for more information and examples.

Functions to configure parallel or chunk-wise processing:

- `processingChunkSize()`: allows to get or set the size of the chunks for parallel processing or chunk-wise processing of a `Spectra` in general. With a value of `Inf` (the default) no chunk-wise processing will be performed.
- `processingChunkFactor()`: returns a `factor` defining the chunks into which a `Spectra` will be split for chunk-wise (parallel) processing. A `factor` of length 0 indicates that no chunk-wise processing will be performed.

Usage

```
## S4 method for signature 'Spectra'
processingChunkSize(object)

## S4 replacement method for signature 'Spectra'
processingChunkSize(object) <- value

## S4 method for signature 'Spectra'
processingChunkFactor(object)

## S4 method for signature 'Spectra'
backendBpparam(object, BPPARAM = bpparam())
```

Arguments

object	Spectra object.
value	integer(1) defining the chunk size.
BPPARAM	Parallel setup configuration. Defaults to BPPARAM = bpparam() hence uses a globally defined setup. See *notes* and BiocParallel::bpparam() for more information.

Value

processingChunkSize() returns the currently defined processing chunk size (or Inf if it is not defined). processingChunkFactor() returns a factor defining the chunks into which x will be split for (parallel) chunk-wise processing or a factor of length 0 if no splitting is defined.

Note

Some backends might not support parallel processing at all. For these, the backendBpparam() function will always return a SerialParam() independently on how parallel processing was defined.

Also, on Windows, the only supported parallel processing options is BiocParallel::SnowParam() which requires starting and registering a separate R process for each parallel process. This can take up to 30-40 seconds. In Windows, it is thus advisable to either register and initiate a *global* parallel processing setup (e.g. register(bpstart(SnowParam(4))) to register and start 4 parallel processes) or to use BPPARAM = SerialParam() to disable parallel processing alltogether.

Author(s)

Johannes Rainer

processingLog *Data manipulation and analysis methods*

Description

Various data analysis functions are available for `Spectra` objects. These can be categorized into functions that either return a `Spectra` object (with the manipulated data) and functions that directly return the result from the calculation. For the former category, the data manipulations are cached in the result object's *processing queue* and only exectuted on-the-fly when the respective data gets extracted from the `Spectra` (see section *The processing queue* for more information).

For the second category, the calculations are directly executed and the result, usually one value per spectrum, returned. Generally, to reduce memory demand, a chunk-wise processing of the data is performed.

Usage

```
processingLog(x)

scalePeaks(x, by = sum, msLevel. = uniqueMsLevels(x))

shiftPeaks(object, offset = 0, direction = c("right", "left"), ...)

## S4 method for signature 'Spectra'
addProcessing(object, FUN, ..., spectraVariables = character())

## S4 method for signature 'Spectra'
applyProcessing(
  object,
  f = processingChunkFactor(object),
  BPPARAM = bpparam(),
  ...
)

## S4 method for signature 'Spectra'
bin(
  x,
  binSize = 1L,
  breaks = NULL,
  msLevel. = uniqueMsLevels(x),
  FUN = sum,
  zero.rm = TRUE
)

## S4 method for signature 'Spectra'
containsMz(
  object,
```

```
  mz = numeric(),
  tolerance = 0,
  ppm = 20,
  which = c("any", "all"),
  BPPARAM = bpparam()
)

## S4 method for signature 'Spectra'
containsNeutralLoss(
  object,
  neutralLoss = 0,
  tolerance = 0,
  ppm = 20,
  BPPARAM = bpparam()
)

## S4 method for signature 'Spectra'
entropy(object, normalized = TRUE)

## S4 method for signature 'ANY'
entropy(object, ...)

## S4 method for signature 'Spectra'
pickPeaks(
  object,
  halfWindowSize = 2L,
  method = c("MAD", "SuperSmoother"),
  snr = 0,
  k = 0L,
  descending = FALSE,
  threshold = 0,
  msLevel. = uniqueMsLevels(object),
  ...
)

## S4 method for signature 'Spectra'
replaceIntensitiesBelow(
  object,
  threshold = min,
  value = 0,
  msLevel. = uniqueMsLevels(object)
)

## S4 method for signature 'Spectra'
reset(object, ...)

## S4 method for signature 'Spectra'
smooth(
```

```
  x,
  halfWindowSize = 2L,
  method = c("MovingAverage", "WeightedMovingAverage", "SavitzkyGolay"),
  msLevel. = uniqueMsLevels(x),
  ...
)

## S4 method for signature 'Spectra'
spectrapply(
  object,
  FUN,
  ...,
  chunkSize = integer(),
  f = factor(),
  BPPARAM = SerialParam()
)
```

Arguments

x	A Spectra.
by	For scalePeaks(): function to calculate a single numeric from intensity values of a spectrum by which all intensities (of that spectrum) should be divided by. The default by = sum will divide intensities of each spectrum by the sum of intensities of that spectrum.
msLevel.	integer defining the MS level(s) of the spectra to which the function should be applied (defaults to all MS levels of object.
object	A Spectra object.
offset	For shiftPeaks(): numeric(1) offset or character(1) with the name of a spectra variable containing an (per spectrum) offset value to shift the peaks.
direction	For shiftPeaks(): character(1) defining the direction of the shift. Can be either shiftPeaks = "right" (the default) which will shift peaks to the right (i.e., peaks' m/z + offset) or shiftPeaks = "left" that will shift peaks to the left by subtracting offset from the peaks' m/z.
...	Additional arguments passed to internal and downstream functions.
FUN	For addProcessing(): function to be applied to the peak matrix of each spectrum in object. For bin(): function to aggregate intensity values of peaks falling into the same bin. Defaults to FUN = sum thus summing up intensities. For spectrapply() and chunkapply(): function to be applied to each individual or each chunk of Spectra.
spectraVariables	For addProcessing(): character with additional spectra variables that should be passed along to the function defined with FUN. See function description for details.
f	For spectrapply() and applyProcessing(): factor defining how object should be splitted for eventual parallel processing. Defaults to factor() for spectrapply() hence the object is not splitted while it defaults to f = processingChunkSize(object)

	for applyProcessing() splitting thus the object by default into chunks depending on processingChunkSize().
BPPARAM	Parallel setup configuration. See processingChunkSize() and BiocParallel::bpparam() for more information.
binSize	For bin(): numeric(1) defining the size for the m/z bins. Defaults to binSize = 1.
breaks	For bin(): numeric defining the m/z breakpoints between bins.
zero.rm	For bin(): logical(1) indicating whether to remove bins with zero intensity. Defaults to TRUE, meaning the function will discard bins created with an intensity of 0 to enhance memory efficiency.
mz	For containsMz(): numeric with the m/z value(s) of the mass peaks to check.
tolerance	For containsMz() and neutralLoss(): numeric(1) allowing to define a constant maximal accepted difference between m/z values for peaks to be matched.
ppm	For containsMz() and neutralLoss(): numeric(1) defining a relative, m/z-dependent, maximal accepted difference between m/z values for peaks to be matched.
which	For containsMz(): either "any" or "all" defining whether any (the default) or all provided mz have to be present in the spectrum.
neutralLoss	for containsNeutralLoss(): numeric(1) defining the value which should be subtracted from the spectrum's precursor m/z.
normalized	for entropy(): logical(1) whether the normalized entropy should be calculated (default). See also MsCoreUtils::nentropy() for details.
halfWindowSize	For pickPeaks(): integer(1), used in the identification of the mass peaks: a local maximum has to be the maximum in the window from (i - halfWindowSize):(i + halfWindowSize). For smooth(): integer(1), used in the smoothing algorithm, the window reaches from (i - halfWindowSize):(i + halfWindowSize).
method	For pickPeaks(): character(1), the noise estimators that should be used, currently the the *M*edian *A*bsolute *D*eviation (method = "MAD") and Friedman's Super Smoother (method = "SuperSmoother") are supported. For smooth(): character(1), the smoothing function that should be used, currently, the Moving-Average- (method = "MovingAverage"), Weighted-Moving-Average- (method = "WeightedMovingAver Savitzky-Golay-Smoothing (method = "SavitzkyGolay") are supported.
snr	For pickPeaks(): double(1) defining the *S*ignal-to-*N*oise-*R*atio. The intensity of a local maximum has to be higher than snr * noise to be considered as peak.
k	For pickPeaks(): integer(1), number of values left and right of the peak that should be considered in the weighted mean calculation.
descending	For pickPeaks(): logical, if TRUE just values between the nearest valleys around the peak centroids are used.
threshold	For pickPeaks(): a numeric(1) defining the proportion of the maximal peak intensity. Only values above the threshold are used for the weighted mean calculation. For replaceIntensitiesBelow(): a numeric(1) defining the threshold or a function to calculate the threshold for each spectrum on its intensity values. Defaults to threshold = min.

value	For replaceIntensitiesBelow(): numeric(1) defining the value with which intensities should be replaced with.
chunkSize	For spectrapply(): size of the chunks into which the Spectra should be split. This parameter overrides parameters f and BPPARAM.

Value

See the documentation of the individual functions for a description of the return value.

Data analysis methods returning a Spectra

The methods listed here return a Spectra object as a result.

- addProcessing(): adds an arbitrary function that should be applied to the peaks matrix of every spectrum in object. The function (can be passed with parameter FUN) is expected to take a peaks matrix as input and to return a peaks matrix. A peaks matrix is a numeric matrix with two columns, the first containing the m/z values of the peaks and the second the corresponding intensities. The function has to have ... in its definition. Additional arguments can be passed with .... With parameter spectraVariables it is possible to define additional spectra variables from object that should be passed to the function FUN. These will be passed by their name (e.g. specifying spectraVariables = "precursorMz" will pass the spectra's precursor m/z as a parameter named precursorMz to the function. The only exception is the spectra's MS level, these will be passed to the function as a parameter called spectrumMsLevel (i.e. with spectraVariables = "msLevel" the MS levels of each spectrum will be submitted to the function as a parameter called spectrumMsLevel). Examples are provided in the package vignette.
- bin(): aggregates individual spectra into discrete (m/z) bins. Binning is performed only on spectra of the specified MS level(s) (parameter msLevel, by default all MS levels of x). The bins can be defined with parameter breaks which by default are equally sized bins, with size being defined by parameter binSize, from the minimal to the maximal m/z of all spectra (of MS level msLevel) within x. The same bins are used for all spectra in x. All intensity values for peaks falling into the same bin are aggregated using the function provided with parameter FUN (defaults to FUN = sum, i.e. all intensities are summed up). Note that the binning operation is applied to the peak data on-the-fly upon data access and it is possible to *revert* the operation with the reset() function (see description of reset() below).
- countIdentifications: counts the number of identifications each scan has led to. See countIdentifications() for more details.
- pickPeaks(): picks peaks on individual spectra using a moving window-based approach (window size = 2 * halfWindowSize). For noisy spectra there are currently two different noise estimators available, the *M*edian *A*bsolute *D*eviation (method = "MAD") and Friedman's Super Smoother (method = "SuperSmoother"), as implemented in the MsCoreUtils::noise(). The method supports also to optionally *refine* the m/z value of the identified centroids by considering data points that belong (most likely) to the same mass peak. Therefore the m/z value is calculated as an intensity weighted average of the m/z values within the peak region. The peak region is defined as the m/z values (and their respective intensities) of the 2 * k closest signals to the centroid or the closest valleys (descending = TRUE) in the 2 * k region. For the latter the k has to be chosen general larger. See MsCoreUtils::refineCentroids() for details. If the ratio of the signal to the highest intensity of the peak is below threshold it will be ignored for the weighted average.

- `replaceIntensitiesBelow()`: replaces intensities below a specified threshold with the provided `value`. Parameter `threshold` can be either a single numeric value or a function which is applied to all non-`NA` intensities of each spectrum to determine a threshold value for each spectrum. The default is `threshold = min` which replaces all values which are `<=` the minimum intensity in a spectrum with `value` (the default for `value` is `0`). Note that the function specified with `threshold` is expected to have a parameter `na.rm` since `na.rm = TRUE` will be passed to the function. If the spectrum is in profile mode, ranges of successive non-0 peaks `<= threshold` are set to 0. Parameter `msLevel.` allows to apply this to only spectra of certain MS level(s).
- `scalePeaks()`: scales intensities of peaks within each spectrum depending on parameter `by`. With `by = sum` (the default) peak intensities are divided by the sum of peak intensities within each spectrum. The sum of intensities is thus 1 for each spectrum after scaling. Parameter `msLevel.` allows to apply the scaling of spectra of a certain MS level. By default (`msLevel. = uniqueMsLevels(x)`) intensities for all spectra will be scaled.
- `shiftPeaks()`: shifts peaks of each spectrum along *m/z* dimension by an offset. The *direction* of this shift can be defined with parameter `direction`. The default `direction = "right"` will replace the peak's m/z with m/z + `offset`, `direction = "left"` subtracts the offset from the peak's m/z (m/z - `offset`). Parameter `offset` can be a `numeric(1)`, or a `character(1)` with the name of a spectra variable containing an `offset` for each spectrum. For example, to add the precursor m/z value of a spectrum to the peak's m/z values use `offset = "precursorMz"`. See examples below for more details.
- `smooth()`: smooths individual spectra using a moving window-based approach (window size = `2 * halfWindowSize`). Currently, the Moving-Average- (`method = "MovingAverage"`), Weighted-Moving-Average- (`method = "WeightedMovingAverage"`), weights depending on the distance of the center and calculated `1/2^(-halfWindowSize:halfWindowSize)`) and Savitzky-Golay-Smoothing (`method = "SavitzkyGolay"`) are supported. For details how to choose the correct `halfWindowSize` please see `MsCoreUtils::smooth()`.

Data analysis methods returning the result from the calculation

The functions listed in this section return immediately the result from the calculation. To reduce memory demand (and allow parallel processing) the calculations a chunk-wise processing is generally performed.

- `chunkapply()`: apply an arbitrary function to chunks of spectra. See `chunkapply()` for details and examples.
- `containsMz()`: checks for each of the spectra whether they contain mass peaks with an m/z equal to `mz` (given acceptable difference as defined by parameters `tolerance` and `ppm` - see `MsCoreUtils::common()` for details). Parameter `which` allows to define whether any (`which = "any"`, the default) or all (`which = "all"`) of the `mz` have to match. The function returns `NA` if `mz` is of length 0 or is `NA`.
- `containsNeutralLoss()`: checks for each spectrum in `object` if it has a peak with an m/z value equal to its precursor m/z - `neutralLoss` (given acceptable difference as defined by parameters `tolerance` and `ppm`). Returns `NA` for MS1 spectra (or spectra without a precursor m/z).
- `entropy()`: calculates the entropy of each spectra based on the metrics suggested by Li et al. (https://doi.org/10.1038/s41592-021-01331-z). See also `MsCoreUtils::nentropy()` in the *MsCoreUtils* package for details.

- estimatePrecursorIntensity(): defines the precursor intensities for MS2 spectra using the intensity of the matching MS1 peak from the closest MS1 spectrum (i.e. the last MS1 spectrum measured before the respective MS2 spectrum). With method = "interpolation" it is also possible to calculate the precursor intensity based on an interpolation of intensity values (and retention times) of the matching MS1 peaks from the previous and next MS1 spectrum. See estimatePrecursorIntensity() for examples and more details.
- estimatePrecursorMz(): **for DDA data**: allows to estimate a fragment spectra's precursor m/z based on the reported precursor m/z and the data from the previous MS1 spectrum. See estimatePrecursorMz() for details.
- neutralLoss(): calculates neutral loss spectra for fragment spectra. See neutralLoss() for detailed documentation.
- spectrapply(): applies a given function to each individual spectrum or sets of a Spectra object. By default, the Spectra is split into individual spectra (i.e. Spectra of length 1) and the function FUN is applied to each of them. An alternative splitting can be defined with parameter f. Parameters for FUN can be passed using .... The returned result and its order depend on the function FUN and how object is split (hence on f, if provided). Parallel processing is supported and can be configured with parameter BPPARAM, is however only suggested for computational intense FUN. As an alternative to the (eventual parallel) processing of the full Spectra, spectrapply() supports also a chunk-wise processing. For this, parameter chunkSize needs to be specified. object is then split into chunks of size chunkSize which are then (stepwise) processed by FUN. This guarantees a lower memory demand (especially for on-disk backends) since only the data for one chunk needs to be loaded into memory in each iteration. Note that by specifying chunkSize, parameters f and BPPARAM will be ignored. See also chunkapply() above or examples below for details on chunk-wise processing.

The processing queue

Operations that modify mass peak data, i.e. the m/z and intensity values of a Spectra are generally not applied immediately to the data but are *cached* within the object's *processing queue*. These operations are then applied to the data only upon request, for example when m/z and/or intensity values are extracted. This lazy execution guarantees that the same functionality can be applied to any Spectra object, regardless of the type of backend that is used. Thus, data manipulation operations can also be applied to data that is *read only*. As a side effect, this enables also to *undo* operations using the reset() function.

Functions related to the processing queue are:

- applyProcessing(): for Spectra objects that use a **writeable** backend only: apply all steps from the lazy processing queue to the peak data and write it back to the data storage. Parameter f allows to specify how object should be split for parallel processing. This should either be equal to the dataStorage, or f = rep(1, length(object)) to disable parallel processing alltogether. Other partitionings might result in errors (especially if a MsBackendHdf5Peaks backend is used).
- processingLog(): returns a character vector with the processing log messages.
- reset(): restores the data to its original state (as much as possible): removes any processing steps from the lazy processing queue and calls reset() on the backend which, depending on the backend, can also undo e.g. data filtering operations. Note that a reset*(call after applyProcessing() will not have any effect. See examples below for more information.

Author(s)

Sebastian Gibb, Johannes Rainer, Laurent Gatto, Philippine Louail, Nir Shahaf, Mar Garcia-Aloy

See Also

- compareSpectra() for calculation of spectra similarity scores.
- processingChunkSize() for information on parallel and chunk-wise data processing.
- Spectra for a general description of the Spectra object.

Examples

```
## Load a `Spectra` object with LC-MS/MS data.
fl <- MsDataHub::PestMix1_DDA.mzML()
sps_dda <- Spectra(fl)
sps_dda


##  --------  FUNCTIONS RETURNING A SPECTRA  --------

## Replace peak intensities below 40 with a value of 1
sps_mod <- replaceIntensitiesBelow(sps_dda, threshold = 20, value = 1)
sps_mod

## Get the intensities of the first spectrum before and after the
## operation
intensity(sps_dda[1])
intensity(sps_mod[1])

## Remove all peaks with an intensity below 5.
sps_mod <- filterIntensity(sps_dda, intensity = c(5, Inf))

intensity(sps_mod)

## In addition it is possible to pass a function to `filterIntensity()`: in
## the example below we want to keep only peaks that have an intensity which
## is larger than one third of the maximal peak intensity in that spectrum.
keep_peaks <- function(x, prop = 3) {
    x > max(x, na.rm = TRUE) / prop
}
sps_mod <- filterIntensity(sps_dda, intensity = keep_peaks)
intensity(sps_mod)

## We can also change the proportion by simply passing the `prop` parameter
## to the function. To keep only peaks that have an intensity which is
## larger than half of the maximum intensity:
sps_mod <- filterIntensity(sps_dda, intensity = keep_peaks, prop = 2)
intensity(sps_mod)

## With the `scalePeaks()` function we can alternatively scale the
## intensities of mass peaks per spectrum to relative intensities. This
## is specifically useful for fragment (MS2) spectra. We below thus
```

```
## scale the intensities per spectrum by the total sum of intensities
## (such that the sum of all intensities per spectrum is 1).
## Below we scale the intensities of all MS2 spectra in our data set.
sps_mod <- scalePeaks(sps_dda, msLevel = 2L)

## MS1 spectra were not affected
sps_mod |>
    filterMsLevel(1L) |>
    intensity()

## Intensities of MS2 spectra were scaled
sps_mod |>
    filterMsLevel(2L) |>
    intensity()

## The `shiftPeaks()` function allows to shift peaks in each spectrum by
## an offset value in m/z dimension. As an example we below substract the
## precursor m/z value from the peaks' m/z of each MS2 spectrum to create
## spectra for a *neutral loss* comparison

nl_ms2 <- sps_mod |>
    filterMsLevel(2L) |>
    shiftPeaks(offset = "precursorMz", direction = "left")
## m/z values are shifted
mz(nl_ms2)

## Since data manipulation operations are by default not directly applied to
## the data but only cached in the internal processing queue, it is also
## possible to remove these data manipulations with the `reset()` function:
tmp <- reset(sps_mod)
tmp
lengths(sps_dda) |> head()
lengths(sps_mod) |> head()
lengths(tmp) |> head()

## Data manipulation operations cached in the processing queue can also be
## applied to the mass peaks data with the `applyProcessing()` function, if
## the `Spectra` uses a backend that supports that (i.e. allows replacing
## the mass peaks data). Below we first change the backend to a
## `MsBackendMemory()` and then use the `applyProcessing()` to modify the
## mass peaks data
sps_dda <- setBackend(sps_dda, MsBackendMemory())
sps_mod <- filterIntensity(sps_dda, intensity = c(5, Inf))
sps_mod <- applyProcessing(sps_mod)
sps_mod

## While we can't *undo* this filtering operation now using the `reset()`
## function, accessing the data would now be faster, because the operation
## does no longer to be applied to the original data before returning to the
## user.


##  --------  FUNCTIONS RETURNING THE RESULT  --------
```

```
## With the `spectrapply()` function it is possible to apply an
## arbitrary function to each spectrum in a Spectra.
## In the example below we calculate the mean intensity for each spectrum
## in a subset of the sciex_im data. Note that we can access all variables
## of each individual spectrum either with the `$` operator or the
## corresponding method.
res <- spectrapply(sps_dda[1:20], FUN = function(x) mean(x$intensity[[1]]))
head(res)

## As an alternative, applying a function `FUN` to a `Spectra` can be
## performed *chunk-wise*. The advantage of this is, that only the data for
## one chunk at a time needs to be loaded into memory reducing the memory
## demand. This type of processing can be performed by specifying the size
## of the chunks (i.e. number of spectra per chunk) with the `chunkSize`
## parameter
spectrapply(sps_dda[1:20], lengths, chunkSize = 5L)

## Precursor intensity estimation. Some manufacturers don't report the
## precursor intensity for MS2 spectra:
sps_dda |>
    filterMsLevel(2L) |>
    precursorIntensity()

## This intensity can however be estimated from the previously measured
## MS1 scan with the `estimatePrecursorIntensity()` function:
pi <- estimatePrecursorIntensity(sps_dda)

## This function returned the result as a `numeric` vector with one
## value per spectrum:
pi

## We can replace the precursor intensity values of the originating
## object:
sps_dda$precursorIntensity <- pi
sps_dda |>
    filterMsLevel(2L) |>
    precursorIntensity()
```

`rbindlistWithRownames` *Fast* rbind-ing `data.frame`*s preserving row names*

Description

The `rbindlistWithRownames()` function uses the `data.table::rbindlist()` function for a fast concatenation (row-binding) of a `list` of provided `data.frames` and in addition also preserves their row names (if set).

The parameters are the same as for `rbindlist()`.

Usage

```
rbindlistWithRownames(
  l,
  use.names = TRUE,
  fill = FALSE,
  idcol = NULL,
  ignore.attr = FALSE
)
```

Arguments

l	A list containing data.frames that should be combined.
use.names	If TRUE binds by matching column names, if FALSE by position. "check" (default) warns if not all items have the same names in the same order. See data.table::rbindlist() for details.
fill	logical(1): if TRUE fills missing columns with NAs or NULL for missing list columns. By defalt FALSE.
idcol	Creates a column in the result showing which list item those rows cam from. See data.table::rbindlist() for details.
ignore.attr	logical(1), default FALSE. When TRUE, allows binding columns with different attributes (e.g. class).

Value

A combined data.frame with row names (if present in all data.frames provided).

Note

Row names are dropped if duplicated row names are present, or if row names are not defined for all data.frames in l.

The function uses .row_names_info() to guess wheter row names are *really* set or just the default seq_len(nrow(x)) are used.

Author(s)

Johannes Rainer

See Also

data.table::rbindlist()

Spectra *The Spectra class to manage and access MS data*

Description

The Spectra class encapsules spectral mass spectrometry (MS) data and related metadata. The MS data is represented by a *backend* extending the virual MsBackend class which provides the data to the Spectra object. The Spectra class implements only data accessor, filtering and analysis methods for the MS data and relies on its *backend* to provide the MS data. This allows to change data representations of a Spectra object depending on the user's needs and properties of the data. Different backends and their properties are explained in the MsBackend documentation.

For information on parallel processing options see processingChunkSize().

Documentation on other topics and functionality of Spectracan be found in:

- spectraData() for accessing and using MS data through Spectra objects.
- filterMsLevel() to subset and filter Spectra objects.
- plotSpectra() for visualization of Spectra objects.
- processingChunkSize() for information on parallel and chunk-wise data processing.
- combineSpectra() for merging, aggregating and splitting of Spectra objects.
- combinePeaks() for merging and aggregating Spectra's mass peaks data.
- addProcessing() for data analysis functions.
- compareSpectra() for spectra similarity calculations.

Usage

```
## S4 method for signature 'missing'
Spectra(
  object,
  processingQueue = list(),
  metadata = list(),
  ...,
  backend = MsBackendMemory(),
  BPPARAM = bpparam()
)

## S4 method for signature 'MsBackend'
Spectra(object, processingQueue = list(), metadata = list(), ...)

## S4 method for signature 'character'
Spectra(
  object,
  processingQueue = list(),
  metadata = list(),
  source = MsBackendMzR(),
```

```
  backend = source,
  ...,
  BPPARAM = bpparam()
)

## S4 method for signature 'ANY'
Spectra(
  object,
  processingQueue = list(),
  metadata = list(),
  source = MsBackendMemory(),
  backend = source,
  ...,
  BPPARAM = bpparam()
)

## S4 method for signature 'Spectra,MsBackend'
setBackend(
  object,
  backend,
  f = processingChunkFactor(object),
  ...,
  BPPARAM = bpparam()
)

## S4 method for signature 'Spectra'
export(object, backend, ...)

## S4 method for signature 'Spectra'
dataStorageBasePath(object)

## S4 replacement method for signature 'Spectra'
dataStorageBasePath(object) <- value
```

Arguments

object	For `Spectra()`: an object to instantiate the `Spectra` object and initialize the with data.. See section on creation of `Spectra` objects for details. For all other methods a `Spectra` object.
processingQueue	For `Spectra()`: optional `list` of ProtGenerics::ProcessingStep objects.
metadata	For `Spectra()`: optional `list` with metadata information.
...	Additional arguments.
backend	For `Spectra()`: MsBackend to be used as backend. See section on creation of `Spectra` objects for details. For `setBackend()`: instance of MsBackend that supports `setBackend()` (i.e. for which `supportsSetBackend()` returns `TRUE`). Such backends have a parameter `data` in their `backendInitialize()` function that support passing the full spectra data to the initialize method. See section on

creation of Spectra objects for details. For export(): MsBackend to be used to export the data.

BPPARAM — Parallel setup configuration. Defaults to BPPARAM = bpparam() hence using the *globally* defined default parallel processing setup. See processingChunkSize() and BiocParallel::bpparam() for more information. This is passed directly to the backendInitialize() method of the MsBackend.

source — For Spectra(): instance of MsBackend that can be used to import spectrum data from the provided files. See section *Creation of objects* for more details.

f — For setBackend(): factor defining how to split the data for parallelized copying of the spectra data to the new backend. For some backends changing this parameter can lead to errors. Defaults to processingChunkFactor().

value — For dataStorageBasePath(): A character vector that defines the base directory where the data storage files can be found.

Details

The Spectra class uses by default a lazy data manipulation strategy, i.e. data manipulations such as performed with replaceIntensitiesBelow() are not applied immediately to the data, but applied on-the-fly to the spectrum data once it is retrieved. This enables data manipulation operations also for *read only* data representations. For some backends that allow to write data back to the data storage (such as the MsBackendMemory(), MsBackendDataFrame() and MsBackendHdf5Peaks()) it is possible to apply to queue with the applyProcessing() function (see the applyProcessing() function for details).

Clarifications regarding scan/acquisition numbers and indices:

- A spectrumId (or spectrumID) is a vendor specific field in the mzML file that contains some information about the run/spectrum, e.g.: controllerType=0 controllerNumber=1 scan=5281 file=2
- acquisitionNum is a more a less sanitize spectrum id generated from the spectrumId field by mzR (see here).
- scanIndex is the mzR generated sequence number of the spectrum in the raw file (which doesn't have to be the same as the acquisitionNum)

See also this issue.

Data stored in a Spectra object

The Spectra object is a container for MS data that includes mass peak data (*m/z* and related intensity values, also referred to as *peaks data* in the context of Spectra) and metadata of individual spectra (so called *spectra variables*). While a core set of spectra variables (the coreSpectraVariables()) are guaranteed to be provided by a Spectra, it is possible to add arbitrary additional spectra variables to a Spectra object.

The Spectra object is designed to contain MS data of a (large) set of mass spectra. The data is organized *linearly* and can be thought of a list of mass spectra, i.e. each element in the Spectra is one spectrum.

Creation of objects

`Spectra` classes can be created with the `Spectra()` constructor function which supports the following formats:

- parameter `object` is a `data.frame` or `DataFrame` containing the full spectrum data (spectra variables in columns as well as columns with the individual MS peak data, *m/z* and intensity). The provided `backend` (by default a MsBackendMemory) will be initialized with that data.
- parameter `object` is a MsBackend (assumed to be already initialized).
- parameter `object` is missing, in which case it is supposed that the data is provided by the MsBackend class passed along with the `backend` argument.
- parameter `object` is of type `character` and is expected to be the file names(s) from which spectra should be imported. Parameter `source` allows to define a MsBackend that is able to import the data from the provided source files. The default value for `source` is `MsBackendMzR()` which allows to import spectra data from mzML, mzXML or CDF files.

With `...` additional arguments can be passed to the backend's `backendInitialize()` method. Parameter `backend` allows to specify which MsBackend should be used for data representation and storage.

Data representation of a `Spectra`

The MS data which can be accessed through the `Spectra` object is *represented* by its backend, which means that this backend defines how and where the data is stored (e.g. in memory or on disk). The `Spectra` object relies on the backend to provide the MS data whenever it needs it for data processing. Different backends with different properties, such as minimal memory requirement or fast data access, are defined in the *Spectra* package or one of the MsBackend* packages. More information on backends and their properties is provided in the documentation of MsBackend.

On-disk backends keep only a limited amount of data in memory retrieving most of the data (usually the MS peak data) upon request on-the-fly from their on-disk data representations. Moving the on-disk data storage of such a backend or a serialized object to a different location in the file system will cause data corruption. The `dataStorageBasePath()` and `dataStorageBasePath<-` functions allow in such cases (and if thebackend classes support this operation), to get or change the *base* path to the directory of the backend's data storage. In-memory backends such as MsBackendMemory or MsBackendDataFrame keeping all MS data in memory don't support, and need, this function, but for MsBackendMzR this function can be used to update/adapt the path to the directory containing the original data files. Thus, for `Spectra` objects (using this backend) that were moved to another file system or computer, these functions allow to adjust/adapt the base file path.

Changing data representation of a `Spectra`

The data representation, i.e. the backend of a `Spectra` object can be changed with the `setBackend()` method that takes an instance of the new backend as second parameter `backend`. A call to `setBackend(sps, backend = MsBackendDataFrame())` would for example change the backend of `sps` to the *in-memory* `MsBackendDataFrame`. Changing to a backend is only supported if that backend has a `data` parameter in its `backendInitialize()` method and if `supportsSetBackend()` returns `TRUE` for that backend. `setBackend()` will transfer the full spectra data from the originating backend as a `DataFrame` to the new backend.

Generally, it is not possible to change **to** a read-only backend such as the `MsBackendMzR()` backend.

The definition of the function is: `setBackend(object, backend, ..., f = dataStorage(object), BPPARAM = bpparam())` and its parameters are:

- `object`: the `Spectra` object.
- `backend`: an instance of the new backend, e.g. `[MsBackendMemory()]`.
- `f`: factor allowing to parallelize the change of the backends. By default the process of copying the spectra data from the original to the new backend is performed separately (and in parallel) for each file. Users are advised to use the default setting.
- `...`: optional additional arguments passed to the `backendInitialize()` method of the new backend.
- `BPPARAM`: setup for the parallel processing. See `BiocParallel::bpparam()` for details.

Exporting data from a `Spectra` object

Data from a `Spectra` object can be **exported** to a file with the `export()` function. The actual export of the data is performed by the `export` method of the MsBackend class defined with the mandatory parameter `backend` which defines also the format in which the data is exported. Note however that not all backend classes support export of data. From the `MsBackend` classes in the `Spectra` package currently only the `MsBackendMzR` backend supports data export (to mzML/mzXML file(s)); see the help page of the MsBackend for information on its arguments or the examples below or the vignette for examples.

The definition of the function is `export(object, backend, ...)` and its parameters are:

- `object`: the `Spectra` object to be exported.
- `backend`: instance of a class extending MsBackend which supports export of the data (i.e. which has a defined `export` method).
- `...`: additional parameters specific for the `MsBackend` passed with parameter `backend`.

Author(s)

Sebastian Gibb, Johannes Rainer, Laurent Gatto, Philippine Louail

Examples

```
##  --------  CREATION OF SPECTRA OBJECTS  --------

## Create a Spectra providing a `DataFrame` containing the spectrum data.

spd <- DataFrame(msLevel = c(1L, 2L), rtime = c(1.1, 1.2))
spd$mz <- list(c(100, 103.2, 104.3, 106.5), c(45.6, 120.4, 190.2))
spd$intensity <- list(c(200, 400, 34.2, 17), c(12.3, 15.2, 6.8))

data <- Spectra(spd)
data

## Create a Spectra from mzML files and use the `MsBackendMzR` on-disk
## backend. Example mzML files are provided by the *MsDataHub* package.
sciex_file <- c(MsDataHub::X20171016_POOL_POS_1_105.134.mzML(),
                MsDataHub::X20171016_POOL_POS_3_105.134.mzML())
```

```
sciex <- Spectra(sciex_file, backend = MsBackendMzR())
sciex


##  --------  CHANGING DATA REPRESENTATIONS  --------

## The MS data is on disk and will be read into memory on-demand. We can
## however change the backend to a MsBackendMemory backend which will
## keep all of the data in memory.
sciex_im <- setBackend(sciex, MsBackendMemory())
sciex_im

## The `MsBackendMemory()` supports the `setBackend()` method:
supportsSetBackend(MsBackendMemory())

## Thus, it is possible to change to that backend with `setBackend()`. Most
## read-only backends however don't support that, such as the
## `MsBackendMzR` and `setBackend()` would fail to change to that backend.
supportsSetBackend(MsBackendMzR())

## The on-disk object `sciex` is light-weight, because it does not keep the
## MS peak data in memory. The `sciex_im` object in contrast keeps all the
## data in memory and its size is thus much larger.
object.size(sciex)
object.size(sciex_im)

## The spectra variable `dataStorage` returns for each spectrum the location
## where the data is stored. For in-memory objects:
head(dataStorage(sciex_im))

## While objects that use an on-disk backend will list the files where the
## data is stored.
head(dataStorage(sciex))

## The spectra variable `dataOrigin` returns for each spectrum the *origin*
## of the data. If the data is read from e.g. mzML files, this will be the
## original mzML file name:
head(dataOrigin(sciex))
head(dataOrigin(sciex_im))


##  -------- DATA EXPORT  --------

## Some `MsBackend` classes provide an `export()` method to export the data
## to the file format supported by the backend.
## The `MsBackendMzR` for example allows to export MS data to mzML or
## mzXML file(s), the `MsBackendMgf` (defined in the MsBackendMgf R package)
## would allow to export the data in mgf file format.
## Below we export the MS data in `data`. We call the `export()` method on
## this object, specify the backend that should be used to export the data
## (and which also defines the output format) and provide a file name.
fl <- tempfile()
export(data, MsBackendMzR(), file = fl)
```

```
## This exported our data in mzML format. Below we read the first 6 lines
## from that file.
readLines(fl, n = 6)

## If only a single file name is provided, all spectra are exported to that
## file. To export data with the `MsBackendMzR` backend to different files, a
## file name for each individual spectrum has to be provided.
## Below we export each spectrum to its own file.
fls <- c(tempfile(), tempfile())
export(data, MsBackendMzR(), file = fls)

## Reading the data from the first file
res <- Spectra(backendInitialize(MsBackendMzR(), fls[1]))

mz(res)
mz(data)
```

`spectra-plotting` *Plotting Spectra*

Description

`Spectra()` can be plotted with one of the following functions

- `plotSpectra()`: plots each spectrum in its separate plot by splitting the plot area into as many panels as there are spectra.
- `plotSpectraOverlay()`: plots all spectra in `x` **into the same** plot (as an overlay).
- `plotSpectraMirror()`: plots a pair of spectra as a *mirror plot*. Parameters `x` and `y` both have to be a `Spectra` of length 1. Matching peaks (considering `ppm` and `tolerance`) are highlighted. See `MsCoreUtils::common()` for details on peak matching. Parameters `matchCol`, `matchLty`, `matchLwd` and `matchPch` allow to customize how matching peaks are indicated.

Usage

```
plotSpectra(
  x,
  xlab = "m/z",
  ylab = "intensity",
  type = "h",
  xlim = numeric(),
  ylim = numeric(),
  main = character(),
  col = "#00000080",
  labels = list(),
  labelCex = 1,
  labelSrt = 0,
  labelAdj = NULL,
```

```
  labelPos = NULL,
  labelOffset = 0.5,
  labelCol = "#00000080",
  asp = 1,
  ...
)

plotSpectraOverlay(
  x,
  xlab = "m/z",
  ylab = "intensity",
  type = "h",
  xlim = numeric(),
  ylim = numeric(),
  main = paste(length(x), "spectra"),
  col = "#00000080",
  labels = list(),
  labelCex = 1,
  labelSrt = 0,
  labelAdj = NULL,
  labelPos = NULL,
  labelOffset = 0.5,
  labelCol = "#00000080",
  axes = TRUE,
  frame.plot = axes,
  ...
)

## S4 method for signature 'Spectra'
plotSpectraMirror(
  x,
  y,
  xlab = "m/z",
  ylab = "intensity",
  type = "h",
  xlim = numeric(),
  ylim = numeric(),
  main = character(),
  col = "#00000080",
  labels = list(),
  labelCex = 1,
  labelSrt = 0,
  labelAdj = NULL,
  labelPos = NULL,
  labelOffset = 0.5,
  labelCol = "#00000080",
  axes = TRUE,
  frame.plot = axes,
```

```
  ppm = 20,
  tolerance = 0,
  matchCol = "#80B1D3",
  matchLwd = 1,
  matchLty = 1,
  matchPch = 16,
  ...
)
```

Arguments

x	a Spectra() object. For plotSpectraMirror() it has to be an object of length 2.
xlab	character(1) with the label for the x-axis (by default xlab = "m/z").
ylab	character(1) with the label for the y-axis (by default ylab = "intensity").
type	character(1) specifying the type of plot. See plot.default() for details. Defaults to type = "h" which draws each peak as a line.
xlim	numeric(2) defining the x-axis limits. The range of m/z values are used by default.
ylim	numeric(2) defining the y-axis limits. The range of intensity values are used by default.
main	character(1) with the title for the plot. By default the spectrum's MS level and retention time (in seconds) is used.
col	color to be used to draw the peaks. Should be either of length 1, or equal to the number of spectra (to plot each spectrum in a different color) or be a list with colors for each individual peak in each spectrum.
labels	allows to specify a label for each peak. Needs to be a list() with length equal to the number of spectra (each element of the list being a character() with length equal to the number of peaks for that spectrum), or, ideally, a function that uses one of the Spectra's variables (see examples below). plotSpectraMirror() supports only labels of type *function*.
labelCex	numeric(1) giving the amount by which the text should be magnified relative to the default. See parameter cex in par().
labelSrt	numeric(1) defining the rotation of the label. See parameter srt in text().
labelAdj	see parameter adj in text().
labelPos	see parameter pos in text().
labelOffset	see parameter offset in text().
labelCol	color for the label(s).
asp	for plotSpectra(): the target ratio (columns / rows) when plotting mutliple spectra (e.g. for 20 spectra use asp = 4/5 for 4 columns and 5 rows or asp = 5/4 for 5 columns and 4 rows; see grDevices::n2mfrow() for details).
...	additional parameters to be passed to the plot.default() function.
axes	logical(1) whether (x and y) axes should be drawn.

frame.plot	logical(1) whether a box should be drawn around the plotting area.
y	for plotSpectraMirror(): Spectra object of length 1 against which x should be plotted against.
ppm	for plotSpectraMirror(): m/z relative acceptable difference (in ppm) for peaks to be considered matching (see MsCoreUtils::common() for more details).
tolerance	for plotSpectraMirror(): absolute acceptable difference of m/z values for peaks to be considered matching (see MsCoreUtils::common() for more details).
matchCol	for plotSpectraMirror(): color for matching peaks.
matchLwd	for plotSpectraMirror(): line width (lwd) to draw matching peaks. See par() for more details.
matchLty	for plotSpectraMirror(): line type (lty) to draw matching peaks. See par() for more details.
matchPch	for plotSpectraMirror(): point character (pch) to label matching peaks. Defaults to matchPch = 16, set to matchPch = NA to disable. See par() for more details.

Value

These functions create a plot.

Author(s)

Johannes Rainer, Sebastian Gibb, Laurent Gatto, Guillaume Deflandre

Examples

```
ints <- list(c(4.3412, 12, 8, 34, 23.4),
    c(8, 25, 16, 32))
mzs <- list(c(13.453421, 43.433122, 46.6653553, 129.111212, 322.24432),
    c(13.452, 43.5122, 129.112, 322.245))

df <- DataFrame(msLevel = c(1L, 1L), rtime = c(123.12, 124))
df$mz <- mzs
df$intensity <- ints
sp <- Spectra(df)

#### ------------------------------------------- ####
##                  plotSpectra                    ##

## Plot one spectrum.
plotSpectra(sp[1])

## Plot both spectra.
plotSpectra(sp)

## Define a color for each peak in each spectrum.
plotSpectra(sp, col = list(c(1, 2, 3, 4, 5), 1:4))
```

```
## Color peaks from each spectrum in different colors.
plotSpectra(sp, col = c("green", "blue"))

## Label each peak with its m/z.
plotSpectra(sp, labels = function(z) lapply(mz(z), format, digits = 4))

## Rotate the labels.
plotSpectra(sp, labels = function(z) lapply(mz(z), format, digits = 4),
    labelPos = 2, labelOffset = 0.1, labelSrt = -30)

## Add a custom annotation for each peak.
sp$label <- list(c("", "A", "B", "C", "D"),
    c("Frodo", "Bilbo", "Peregrin", "Samwise"))

## Plot each peak in a different color
plotSpectra(sp, labels = sp$label,
    col = list(1:5, 1:4))

## Plot a single spectrum specifying the label.
plotSpectra(sp[2], labels = list(c("A", "B", "C", "D")))


#### ---------------------------------------------- ####
##                plotSpectraOverlay                ##

## Plot both spectra overlaying.
plotSpectraOverlay(sp)

## Use a different color for each spectrum.
plotSpectraOverlay(sp, col = c("#ff000080", "#0000ff80"))

## Label also the peaks with their m/z if their intensity is above 15.
plotSpectraOverlay(sp, col = c("#ff000080", "#0000ff80"),
labels = function(z) {
    lapply(seq_along(mz(z)), function(i) {
            lbls <- format(mz(z)[[i]], digits = 4)
            lbls[intensity(z)[[i]] <= 15] <- ""
            lbls
     })
 })
 abline(h = 15, lty = 2)

## Use different asp values
plotSpectra(sp, asp = 1/2)
plotSpectra(sp, asp = 2/1)

#### ---------------------------------------------- ####
##                plotSpectraMirror                 ##

## Plot two spectra against each other.
plotSpectraMirror(sp[1], sp[2])
```

```
## Label the peaks with their m/z
plotSpectraMirror(sp[1], sp[2],
    labels = function(z) list(format(mz(z)[[1L]], digits = 3)),
    labelSrt = -30, labelPos = 2, labelOffset = 0.2)
grid()

## The same plot with a tolerance of 0.1 and using a different color to
## highlight matching peaks
plotSpectraMirror(sp[1], sp[2],
    labels = function(z) list(format(mz(z)[[1L]], digits = 3)),
    labelSrt = -30, labelPos = 2, labelOffset = 0.2, tolerance = 0.1,
    matchCol = "#ff000080", matchLwd = 2)
grid()
```

`spectraData` *Accessing mass spectrometry data*

Description

As detailed in the documentation of the Spectra class, a `Spectra` object is a container for mass spectrometry (MS) data that includes both the mass peaks data (or *peaks data*, generally *m/z* and intensity values) as well as spectra metadata (so called *spectra variables*). Spectra variables generally define one value per spectrum, while for peaks variables one value per mass peak is defined and hence multiple values per spectrum (depending on the number of mass peaks of a spectrum).

Data can be extracted from a `Spectra` object using dedicated accessor functions or also using the `$` operator. Depending on the backend class used by the `Spectra` to represent the data, data can also be added or replaced (again, using dedicated functions or using `$<-`).

Usage

```
asDataFrame(
  object,
  i = seq_along(object),
  spectraVars = spectraVariables(object)
)

## S4 method for signature 'Spectra'
acquisitionNum(object)

## S4 method for signature 'Spectra'
centroided(object)

## S4 replacement method for signature 'Spectra'
centroided(object) <- value

## S4 method for signature 'Spectra'
collisionEnergy(object)
```

```
## S4 replacement method for signature 'Spectra'
collisionEnergy(object) <- value

coreSpectraVariables()

## S4 method for signature 'Spectra'
dataOrigin(object)

## S4 replacement method for signature 'Spectra'
dataOrigin(object) <- value

## S4 method for signature 'Spectra'
dataStorage(object)

## S4 method for signature 'Spectra'
intensity(object, f = processingChunkFactor(object), ...)

## S4 method for signature 'Spectra'
ionCount(object)

## S4 method for signature 'Spectra'
isCentroided(object, ...)

## S4 method for signature 'Spectra'
isEmpty(x)

## S4 method for signature 'Spectra'
isolationWindowLowerMz(object)

## S4 replacement method for signature 'Spectra'
isolationWindowLowerMz(object) <- value

## S4 method for signature 'Spectra'
isolationWindowTargetMz(object)

## S4 replacement method for signature 'Spectra'
isolationWindowTargetMz(object) <- value

## S4 method for signature 'Spectra'
isolationWindowUpperMz(object)

## S4 replacement method for signature 'Spectra'
isolationWindowUpperMz(object) <- value

## S4 method for signature 'Spectra'
length(x)

## S4 method for signature 'Spectra'
```

```
lengths(x, use.names = FALSE)

## S4 method for signature 'Spectra'
longForm(
  object,
  columns = union(spectraVariables(object), peaksVariables(object))
)

## S4 method for signature 'Spectra'
msLevel(object)

## S4 method for signature 'Spectra'
mz(object, f = processingChunkFactor(object), ...)

## S4 method for signature 'Spectra'
peaksData(
  object,
  columns = c("mz", "intensity"),
  f = processingChunkFactor(object),
  return.type = c("SimpleList", "list"),
  ...,
  BPPARAM = bpparam()
)

## S4 method for signature 'Spectra'
peaksVariables(object)

## S4 method for signature 'Spectra'
polarity(object)

## S4 replacement method for signature 'Spectra'
polarity(object) <- value

## S4 method for signature 'Spectra'
precScanNum(object)

## S4 method for signature 'Spectra'
precursorCharge(object)

## S4 method for signature 'Spectra'
precursorIntensity(object)

## S4 method for signature 'Spectra'
precursorMz(object)

## S4 replacement method for signature 'Spectra'
precursorMz(object, ...) <- value
```

```
## S4 method for signature 'Spectra'
rtime(object)

## S4 replacement method for signature 'Spectra'
rtime(object) <- value

## S4 method for signature 'Spectra'
scanIndex(object)

## S4 method for signature 'Spectra'
smoothed(object)

## S4 replacement method for signature 'Spectra'
smoothed(object) <- value

## S4 method for signature 'Spectra'
spectraData(object, columns = spectraVariables(object))

## S4 replacement method for signature 'Spectra'
spectraData(object) <- value

## S4 method for signature 'Spectra'
spectraNames(object)

## S4 replacement method for signature 'Spectra'
spectraNames(object) <- value

## S4 method for signature 'Spectra'
spectraVariables(object)

## S4 method for signature 'Spectra'
tic(object, initial = TRUE)

## S4 method for signature 'Spectra'
uniqueMsLevels(object, ...)

## S4 method for signature 'Spectra'
x$name

## S4 replacement method for signature 'Spectra'
x$name <- value

## S4 method for signature 'Spectra'
x[[i, j, ...]]

## S4 replacement method for signature 'Spectra'
x[[i, j, ...]] <- value
```

Arguments

object	A Spectra object.
i	For asDataFrame(): A numeric indicating which scans to coerce to a DataFrame (default is seq_along(object)).
spectraVars	character() indicating what spectra variables to add to the DataFrame. Default is spectraVariables(object), i.e. all available variables.
value	A vector with values to replace the respective spectra variable. Needs to be of the correct data type for the spectra variable.
f	For intensity(), mz() and peaksData(): factor defining how data should be chunk-wise loaded an processed. Defaults to processingChunkFactor().
...	Additional arguments.
x	A Spectra object.
use.names	For lengths(): ignored.
columns	For spectraData() accessor: optional character with column names (spectra variables) that should be included in the returned DataFrame. By default, all columns are returned. For peaksData() accessor: optional character with requested columns in the individual matrix of the returned list. Defaults to c("mz", "value") but any values returned by peaksVariables(object) with object being the Spectra object are supported. For longForm(): character with the spectra and peaks variables to include in the returned data.frame. Defaults to union(spectraVariables(object), peaksVariables(object)).
return.type	For peaksData(): character(1) allowing to specify if the results should be returned as a SimpleList or as a list. Defaults to return.type = "SimpleList".
BPPARAM	Parallel setup configuration. See processingChunkSize() and BiocParallel::bpparam() for more information.
initial	For tic(): logical(1) whether the initially reported total ion current should be reported, or whether the total ion current should be (re)calculated on the actual data (initial = FALSE, same as ionCount()).
name	For $ and $<-: the name of the spectra variable to return or set.
j	For [: not supported.

Spectra variables

A common set of *core spectra variables* are defined for Spectra. These have a pre-defined data type and each Spectra will return a value for these if requested. If no value for a spectra variable is defined, a missing value (of the correct data type) is returned. The list of core spectra variables and their respective data type is:

- *acquisitionNum* integer(1): the index of acquisition of a spectrum during an MS run.
- *centroided* logical(1): whether the spectrum is in profile or centroid mode.
- *collisionEnergy* numeric(1): collision energy used to create an MSn spectrum.
- *dataOrigin* character(1): the *origin* of the spectrum's data, e.g. the mzML file from which it was read.

- *dataStorage* `character(1)`: the (current) storage location of the spectrum data. This value depends on the backend used to handle and provide the data. For an *in-memory* backend like the `MsBackendDataFrame` this will be `"<memory>"`, for an on-disk backend such as the `MsBackendHdf5Peaks` it will be the name of the HDF5 file where the spectrum's peak data is stored.
- *isolationWindowLowerMz* `numeric(1)`: lower m/z for the isolation window in which the (MSn) spectrum was measured.
- *isolationWindowTargetMz* `numeric(1)`: the target m/z for the isolation window in which the (MSn) spectrum was measured.
- *isolationWindowUpperMz* `numeric(1)`: upper m/z for the isolation window in which the (MSn) spectrum was measured.
- *msLevel* `integer(1)`: the MS level of the spectrum.
- *polarity* `integer(1)`: the polarity of the spectrum (`0` and `1` representing negative and positive polarity, respectively).
- *precScanNum* `integer(1)`: the scan (acquisition) number of the precursor for an MSn spectrum.
- *precursorCharge* `integer(1)`: the charge of the precursor of an MSn spectrum.
- *precursorIntensity* `numeric(1)`: the intensity of the precursor of an MSn spectrum.
- *precursorMz* `numeric(1)`: the m/z of the precursor of an MSn spectrum.
- *rtime* `numeric(1)`: the retention time of a spectrum.
- *scanIndex* `integer(1)`: the index of a spectrum within a (raw) file.
- *smoothed* `logical(1)`: whether the spectrum was smoothed.

For each of these spectra variable a dedicated accessor function is defined (such as `msLevel()` or `rtime()`) that allows to extract the values of that spectra variable for all spectra in a `Spectra` object. Also, replacement functions are defined, but not all backends might support replacing values for spectra variables. As described above, additional spectra variables can be defined or added. The `spectraVariables()` function can be used to

Values for multiple spectra variables, or all spectra vartiables* can be extracted with the `spectraData()` function.

Peaks variables

`Spectra` also provide mass peak data with the *m/z* and intensity values being the *core* peaks variables:

- *intensity* `numeric`: intensity values for the spectrum's peaks.
- *mz* `numeric`: the m/z values for the spectrum's peaks.

Values for these can be extracted with the `mz()` and `intensity()` functions, or the `peaksData()` function. The former functions return a `NumericList` with the respective values, while the latter returns a `List` with `numeric` two-column matrices. The list of peaks matrices can also be extracted using `as(x, "list")` or `as(x, "SimpleList")` with `x` being a `Spectra` object.

Some `Spectra`/backends provide also values for additional peaks variables. The set of available peaks variables can be extracted with the `peaksVariables()` function.

Functions to access MS data

The set of available functions to extract data from, or set data in, a Spectra object are (in alphabetical order) listed below. Note that there are also other functions to extract information from a Spectra object documented in addProcessing().

- $, $<-: gets (or sets) a spectra variable for all spectra in object. See examples for details. Note that replacing values of a peaks variable is not supported with a non-empty processing queue, i.e. if any filtering or data manipulations on the peaks data was performed. In these cases applyProcessing() needs to be called first to apply all cached data operations.
- [[, [[<-: access or set/add a single spectrum variable (column) in the backend.
- acquisitionNum(): returns the acquisition number of each spectrum. Returns an integer of length equal to the number of spectra (with NA_integer_ if not available).
- asDataFrame(): converts the Spectra to a DataFrame (in long format) contining all data. Returns a DataFrame. See also longForm() for a potentially more efficient implementation that returns a data.frame in long form.
- centroided(), centroided<-: gets or sets the centroiding information of the spectra. centroided() returns a logical vector of length equal to the number of spectra with TRUE if a spectrum is centroided, FALSE if it is in profile mode and NA if it is undefined. See also isCentroided() for estimating from the spectrum data whether the spectrum is centroided. value for centroided<- is either a single logical or a logical of length equal to the number of spectra in object.
- collisionEnergy(), collisionEnergy<-: gets or sets the collision energy for all spectra in object. collisionEnergy() returns a numeric with length equal to the number of spectra (NA_real_ if not present/defined), collisionEnergy<- takes a numeric of length equal to the number of spectra in object.
- coreSpectraVariables(): returns the *core* spectra variables along with their expected data type.
- dataOrigin(), dataOrigin<-: gets or sets the *data origin* for each spectrum. dataOrigin() returns a character vector (same length than object) with the origin of the spectra. dataOrigin<- expects a character vector (same length than object) with the replacement values for the data origin of each spectrum.
- dataStorage(): returns a character vector (same length than object) with the data storage location of each spectrum.
- intensity(): gets the intensity values from the spectra. Returns a IRanges::NumericList() of numeric vectors (intensity values for each spectrum). The length of the list is equal to the number of spectra in object.
- ionCount(): returns a numeric with the sum of intensities for each spectrum. If the spectrum is empty (see isEmpty()), NA_real_ is returned.
- isCentroided(): a heuristic approach assessing if the spectra in object are in profile or centroided mode. The function takes the qtlth quantile top peaks, then calculates the difference between adjacent m/z value and returns TRUE if the first quartile is greater than k. (See Spectra:::.isCentroided() for the code.)
- isEmpty(): checks whether a spectrum in object is empty (i.e. does not contain any peaks). Returns a logical vector of length equal number of spectra.
- isolationWindowLowerMz(), isolationWindowLowerMz<-: gets or sets the lower m/z boundary of the isolation window.

- `isolationWindowTargetMz()`, `isolationWindowTargetMz<-`: gets or sets the target m/z of the isolation window.
- `isolationWindowUpperMz()`, `isolationWindowUpperMz<-`: gets or sets the upper m/z boundary of the isolation window.
- `length()`: gets the number of spectra in the object.
- `lengths()`: gets the number of peaks (m/z-intensity values) per spectrum. Returns an `integer` vector (length equal to the number of spectra). For empty spectra, `0` is returned.
- `longForm()`: extract the MS data as a `data.frame` in *long form* with columns being spectra and peaks variables and one row per mass peak. Parameter `columns` allows to define the spectra and peaks variables that should be included in the returned `data.frame` (with the default being `columns = union(spectraVariables(object), peaksVariables(object))`).
- `msLevel()`: gets the spectra's MS level. Returns an integer vector (names being spectrum names, length equal to the number of spectra) with the MS level for each spectrum.
- `mz()`: gets the mass-to-charge ratios (m/z) from the spectra. Returns a `IRanges::NumericList()` or length equal to the number of spectra, each element a `numeric` vector with the m/z values of one spectrum.
- `peaksData()`: gets the *peaks* data for all spectra in `object`. Peaks data consist of the m/z and intensity values as well as possible additional annotations (variables) of all peaks of each spectrum. The function returns a `S4Vectors::SimpleList()` of two dimensional arrays (either `matrix` or `data.frame`), with each array providing the values for the requested *peak variables* (by default `"mz"` and `"intensity"`). Optional parameter `columns` is passed to the backend's `peaksData()` function to allow the selection of specific (or additional) peaks variables (columns) that should be extracted (if available). Importantly, it is **not** guaranteed that each backend supports this parameter (while each backend must support extraction of `"mz"` and `"intensity"` columns). Parameter `columns` defaults to `c("mz", "intensity")` but any value returned by `peaksVariables(object)` is supported. Note also that it is possible to extract the peak data with `as(x, "list")` and `as(x, "SimpleList")` as a `list` and `SimpleList`, respectively. Note however that, in contrast to `peaksData()`, `as()` does not support the parameter `columns`.
- `peaksVariables()`: lists the available variables for mass peaks provided by the backend. Default peak variables are `"mz"` and `"intensity"` (which all backends need to support and provide), but some backends might provide additional variables. These variables correspond to the column names of the peak data array returned by `peaksData()`.
- `polarity()`, `polarity<-`: gets or sets the polarity for each spectrum. `polarity()` returns an `integer` vector (length equal to the number of spectra), with `0` and `1` representing negative and positive polarities, respectively. `polarity<-` expects an `integer` vector of length 1 or equal to the number of spectra.
- `precursorCharge()`, `precursorIntensity()`, `precursorMz()`, `precScanNum()`, `precAcquisitionNum()`: gets the charge (`integer`), intensity (`numeric`), m/z (`numeric`), scan index (`integer`) and acquisition number (`interger`) of the precursor for MS level > 2 spectra from the object. Returns a vector of length equal to the number of spectra in `object`. `NA` are reported for MS1 spectra of if no precursor information is available.
- `rtime()`, `rtime<-`: gets or sets the retention times (in seconds) for each spectrum. `rtime()` returns a `numeric` vector (length equal to the number of spectra) with the retention time for each spectrum. `rtime<-` expects a numeric vector with length equal to the number of spectra.

- `scanIndex()`: returns an `integer` vector with the *scan index* for each spectrum. This represents the relative index of the spectrum within each file. Note that this can be different to the `acquisitionNum` of the spectrum which represents the index of the spectrum during acquisition/measurement (as reported in the mzML file).
- `smoothed()`,`smoothed<-`: gets or sets whether a spectrum is *smoothed*. `smoothed()` returns a `logical` vector of length equal to the number of spectra. `smoothed<-` takes a `logical` vector of length 1 or equal to the number of spectra in `object`.
- `spectraData()`: gets general spectrum metadata (annotation, also called header). `spectraData()` returns a `DataFrame`. Note that this method does by default **not** return m/z or intensity values.
- `spectraData<-`: **replaces** the full spectra data of the `Spectra` object with the one provided with `value`. The `spectraData<-` function expects a `DataFrame` to be passed as value with the same number of rows as there a spectra in `object`. Note that replacing values of peaks variables is not supported with a non-empty processing queue, i.e. if any filtering or data manipulations on the peaks data was performed. In these cases `applyProcessing()` needs to be called first to apply all cached data operations and empty the processing queue.
- `spectraNames()`, `spectraNames<-`: gets or sets the spectra names.
- `spectraVariables()`: returns a `character` vector with the available spectra variables (columns, fields or attributes of each spectrum) available in `object`. Note that `spectraVariables()` does not list the *peak variables* (`"mz"`, `"intensity"` and eventual additional annotations for each MS peak). Peak variables are returned by `peaksVariables()`.
- `tic()`: gets the total ion current/count (sum of signal of a spectrum) for all spectra in `object`. By default, the value reported in the original raw data file is returned. For an empty spectrum, `0` is returned.
- `uniqueMsLevels()`: get the unique MS levels available in `object`. This function is supposed to be more efficient than `unique(msLevel(object))`.

Author(s)

Sebastian Gibb, Johannes Rainer, Laurent Gatto, Philippine Louail

See Also

- `addProcessing()` for functions to analyze `Spectra`.
- Spectra for a general description of the `Spectra` object.

Examples

```
## Create a Spectra from mzML files and use the `MsBackendMzR` on-disk
## backend. Example mzML files are provided by the *MsDataHub* package.
sciex_file <- c(MsDataHub::X20171016_POOL_POS_1_105.134.mzML(),
                MsDataHub::X20171016_POOL_POS_3_105.134.mzML())
sciex <- Spectra(sciex_file, backend = MsBackendMzR())
sciex

## Get the number of spectra in the data set
length(sciex)

## Get the number of mass peaks per spectrum - limit to the first 6
```

```
lengths(sciex) |> head()

## Get the MS level for each spectrum - limit to the first 6 spectra
msLevel(sciex) |> head()

## Alternatively, we could also use $ to access a specific spectra variable.
## This could also be used to add additional spectra variables to the
## object (see further below).
sciex$msLevel |> head()

## Get the intensity and m/z values.
intensity(sciex)
mz(sciex)

## Convert a subset of the Spectra object to a long DataFrame.
asDataFrame(sciex, i = 1:3, spectraVars = c("rtime", "msLevel"))

## Create a Spectra providing a `DataFrame` containing the spectrum data.

spd <- DataFrame(msLevel = c(1L, 2L), rtime = c(1.1, 1.2))
spd$mz <- list(c(100, 103.2, 104.3, 106.5), c(45.6, 120.4, 190.2))
spd$intensity <- list(c(200, 400, 34.2, 17), c(12.3, 15.2, 6.8))

s <- Spectra(spd)
s

## List all available spectra variables (i.e. spectrum data and metadata).
spectraVariables(s)

## For all *core* spectrum variables accessor functions are available. These
## return NA if the variable was not set.
centroided(s)
dataStorage(s)
rtime(s)
precursorMz(s)

## The core spectra variables are:
coreSpectraVariables()

## Add an additional metadata column.
s$spectrum_id <- c("sp_1", "sp_2")

## List spectra variables, "spectrum_id" is now also listed
spectraVariables(s)

## Get the values for the new spectra variable
s$spectrum_id

## Extract specific spectra variables.
spectraData(s, columns = c("spectrum_id", "msLevel"))


##  --------  PEAKS VARIABLES AND DATA  --------
```

```
## Get the peak data (m/z and intensity values).
pks <- peaksData(s)
pks
pks[[1]]
pks[[2]]

## Note that we could get the same resulb by coercing the `Spectra` to
## a `list` or `SimpleList`:
as(s, "list")
as(s, "SimpleList")

## Or use `mz()` and `intensity()` to extract the m/z and intensity values
## separately
mz(s)
intensity(s)

## Some `MsBackend` classes provide support for arbitrary peaks variables
## (in addition to the mandatory `"mz"` and `"intensity"` values. Below
## we create a simple data frame with an additional peak variable `"pk_ann"`
## and create a `Spectra` with a `MsBackendMemory` for that data.
## Importantly the number of values (per spectrum) need to be the same
## for all peak variables.

tmp <- data.frame(msLevel = c(2L, 2L), rtime = c(123.2, 123.5))
tmp$mz <- list(c(103.1, 110.4, 303.1), c(343.2, 453.1))
tmp$intensity <- list(c(130.1, 543.1, 40), c(0.9, 0.45))
tmp$pk_ann <- list(c(NA_character_, "A", "P"), c("B", "P"))

## Create the Spectra. With parameter `peaksVariables` we can define
## the columns in `tmp` that contain peaks variables.
sps <- Spectra(tmp, source = MsBackendMemory(),
    peaksVariables = c("mz", "intensity", "pk_ann"))
peaksVariables(sps)

## Extract just the m/z and intensity values
peaksData(sps)[[1L]]

## Extract the full peaks data
peaksData(sps, columns = peaksVariables(sps))[[1L]]

## Access just the pk_ann variable
sps$pk_ann
```

---

spectraVariableMapping

Mapping between spectra variables and data file fields

Description

The spectraVariableMapping function provides the mapping between *spectra variables* of a Spectra() object with data fields from a data file. Such name mapping is expected to enable an easier import of data files with specific *dialects*, e.g. files in MGF format that use a different naming convention for core spectra variables.

MsBackend() implementations are expected to implement this function (if needed) to enable import of data from file formats with non-standardized data fields.

Usage

```
spectraVariableMapping(object, ...)

spectraVariableMapping(object, ...) <- value

## S4 method for signature 'MsBackend'
spectraVariableMapping(object)

## S4 replacement method for signature 'MsBackend'
spectraVariableMapping(object) <- value

## S4 replacement method for signature 'Spectra'
spectraVariableMapping(object) <- value

## S4 method for signature 'Spectra'
spectraVariableMapping(object)
```

Arguments

object	An instance of an object extending MsBackend().
...	Optional parameters.
value	For spectraVariableMapping<-: a named character vector.

Value

A named character with names being spectra variable names (use spectraVariables() for a list of supported names) and values being the data field names.

Author(s)

Johannes Rainer

Index