Package 'Spectra'

March 30, 2021

Title Spectra Infrastructure for Mass Spectrometry Data

Version 1.0.5

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 (>= 4.0.0), S4Vectors, BiocParallel, ProtGenerics (>= 1.17.4)

Imports methods, IRanges, MsCoreUtils (>= 1.1.5), graphics, grDevices, stats, tools, utils, fs

Suggests testthat, knitr (>= 1.1.0), msdata (>= 0.19.3), roxygen2, BiocStyle (>= 2.5.19), mzR (>= 2.19.6), rhdf5 (>= 2.32.0), rmarkdown, vdiffr, magrittr

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

RoxygenNote 7.1.1

Roxygen list(markdown=TRUE)

Collate 'hidden_aliases.R' 'AllGenerics.R' 'MsBackend-functions.R' 'MsBackend.R' 'MsBackendDataFrame-functions.R' 'MsBackendDataFrame.R' 'MsBackendHdf5Peaks-functions.R' 'MsBackendHdf5Peaks.R' 'MsBackendMzR-functions.R' 'MsBackendMzR.R' 'ProcessingStep.R' 'Spectra-functions.R' 'Spectra.R' 'functions-util.R' 'peak-list-functions.R' 'peaks-functions.R' 'plotting-functions.R' 'zzz.R'

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

git_branch RELEASE_3_12

git_last_commit b1c93af

git_last_commit_date 2020-11-25

Date/Publication 2021-03-29

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

Maintainer
RforMassSpectrometry Package Maintainer <maintainer@rformassspectrometry.org>

R topics documented:

---

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

Description

The Spectra class encapsules spectral mass spectrometry data and related metadata.

It supports multiple data backends, e.g. in-memory (MsBackendDataFrame()), on-disk as mzML (MsBackendMzR()) or HDF5 (MsBackendHdf5Peaks()).

Usage

```
addProcessing(object, FUN, ...)

applyProcessing(object, f = dataStorage(object), BPPARAM = bpparam(), ...)

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

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

```
  BPPARAM = bpparam()
)

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

## 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 = MsBackendDataFrame(),
  backend = source,
  ...,
  BPPARAM = bpparam()
)

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

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

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

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

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

## S4 method for signature 'Spectra'
peaksData(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

## 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'
dropNaSpectraVariables(object)

## S4 method for signature 'Spectra'
intensity(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'
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'
spectrapply(
  object,
  FUN,
  f = as.factor(seq_along(object)),
  ...,
  BPPARAM = SerialParam()
)

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

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

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

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

## 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 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'
selectSpectraVariables(object, spectraVariables = spectraVariables(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'
x$name

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

## 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'
filterIntensity(
  object,
  intensity = c(0, Inf),
  msLevel. = unique(msLevel(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. = unique(msLevel(object)))

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

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

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

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

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

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

## S4 method for signature 'Spectra'
bin(x, binSize = 1L, breaks = NULL, msLevel. = unique(msLevel(x)))

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

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

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

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

```
)

## S4 method for signature 'Spectra'
smooth(
  x,
  halfWindowSize = 2L,
  method = c("MovingAverage", "WeightedMovingAverage", "SavitzkyGolay"),
  ...,
  msLevel. = unique(msLevel(x))
)
```

Arguments

object	For Spectra: either a DataFrame or missing. See section on creation of Spectra objects for details. For all other methods a Spectra object.
FUN	For addProcessing: function to be applied to the peak matrix of each spectrum in object. For compareSpectra: function to compare intensities of peaks between two spectra with each other. For combineSpectra: function to combine the (peak matrices) of the spectra. See section *Data manipulations* and examples below for more details.
...	Additional arguments.
f	For split: factor defining how to split x. See base::split() for details. 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. For combineSpectra: factor defining the grouping of the spectra that should be combined. For spectrapply: factor how object should be splitted.
BPPARAM	Parallel setup configuration. See bpparam() for more information. This is passed directly to the backendInitialize() method of the MsBackend.
x	A Spectra object.
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.
processingQueue	For Spectra: optional list of ProcessingStep objects.
metadata	For Spectra: optional list with metadata information.
backend	For Spectra: MsBackend to be used as backend. See section on creation of Spectra objects for details. For setBackend: instance of MsBackend. See section on creation of Spectra objects for details. For export: MsBackend to be used to export the data.
source	For Spectra: instance of MsBackend that can be used to import spectrum data from the provided files. See section *Creation of objects, conversion and changing the backend* for more details.
drop	For [, split: not considered.
value	replacement value for <- methods. See individual method description or expected data type.
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: numeric with the m/z values to match peaks against.

tolerance For compareSpectra, containsMz: numeric(1) allowing to define a constant maximal accepted difference between m/z values for peaks to be matched. For containsMz and filterMzValues it can also be of length equal mz to specify a different tolerance for each m/z value.

ppm For compareSpectra, containsMz, filterMzValues: 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.

use.names For lengths: ignored.

spectraVariables
For selectSpectraVariables: character with the names of the spectra variables to which the backend should be subsetted.

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.

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.

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

j For [: not supported.

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.

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.

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.

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

acquisitionNum for filterPrecursorScan: integer with the acquisition number of the spectra to which the object should be subsetted.

rt for filterRt: numeric(2) defining the retention time range to be used to subset/filter object.

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.

y	A Spectra object.
MAPFUN	For compareSpectra: function to map/match peaks between the two compared spectra. See joinPeaks() for more information and possible functions.
SIMPLIFY	For compareSpectra whether the result matrix should be *simplified* to a numeric if possible (i.e. if either x or y is of length 1).
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 = "WeightedMovingAverage"), 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 double(1) defining the proportion of the maximal peak intensity. Just values above 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.

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. For some backends that allow to write data back to the data storage (such as the MsBackendDataFrame() and MsBackendHdf5Peaks()) it is possible to apply to queue with the applyProcessing function. See the *Data manipulation and analysis methods* section below for more details.

For details on plotting spectra, see plotSpectra().

Value

See individual method description for the return value.

Creation of objects, conversion, changing the backend and export

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

- parameter object is a DataFrame containing the spectrum data. The provided backend (by default a MsBackendDataFrame) 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 storage.

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 or `sps` to the *in-memory* `MsBackendDataFrame`. Note that it is not possible to change the backend to a *read-only* backend (such as the `MsBackendMzR()` backend). `setBackend` changes the `"dataOrigin"` variable of the resulting `Spectra` object to the `"dataStorage"` variable of the backend before the switch.

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

- parameter `object`: the `Spectra` object.
- parameter `backend`: an instance of the new backend, e.g. `MsBackendDataFrame()`.
- parameter `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.
- parameter `...`: optional additional arguments passed to the `backendInitialize()` method of the new `backend`.
- parameter `BPPARAM`: setup for the parallel processing. See `bpparam()` for details.

Data from a `Spectra` object can be **exported** to a file with the `export` function. The actual export of the data has to be performed by the `export` method of the MsBackend class defined with the mandatory parameter `backend`. 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`.

Accessing spectra data

- $, $<-: gets (or sets) a spectra variable for all spectra in `object`. See examples for details.
- `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).
- `peaksData`: gets the *peaks* matrices for all spectra in `object`. The function returns a `SimpleList()` of matrices, each `matrix` with columns `"mz"` and `"intensity"` with the m/z and intensity values for all peaks of a spectrum. Note that it is also possible to extract the peaks matrices with `as(x,"list")` and `as(x,"SimpleList")` as a `list` and `SimpleList`, respectively.

- 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, 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 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.
- 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 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).
- 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.
- 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 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: 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 use of this function is disencouraged, as replacing spectra data with values that are in a different can break the linkage with the associated m/z and intensity values. If possible, spectra variables (i.e. *columns* of the Spectra) should be replaced individually. The spectraData<- function expects a DataFrame to be passed as value.
- spectraNames, spectraNames<-: gets or sets the spectra names.
- spectraVariables: returns a character vector with the available spectra variables (columns, fields or attributes) available in object.
- 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.

Data subsetting, filtering and merging

Subsetting and filtering of Spectra objects can be performed with the below listed methods.

- [: subsets the spectra keeping only selected elements (i). The method **always** returns a Spectra object.
- 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.
- 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).
- `filterMzRange`: filters the object keeping only peaks in each spectrum that are within the provided m/z range.
- `filterMzValues`: filters the object keeping only peaks in each spectrum that match the provided m/z value(s) considering also the absolute `tolerance` and m/z-relative `ppm` (`tolerance` and `ppm` can be either of length 1 or equal to the length of `mz` to define a different tolerance for each m/z).
- `filterPolarity`: filters the object keeping only spectra matching the provided polarity. Returns the filtered `Spectra` (with spectra in their original order).
- `filterPrecursorMz`: 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*.
- `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).
- `filterRt`: retains spectra of MS level `msLevel` with retention times (in seconds) within (`>=`) `rt[1]` and (`<=`) `rt[2]`. Returns the filtered `Spectra` (with spectra in their original order).
- `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.
- `selectSpectraVariables`: reduces the information within the object to the selected spectra variables: all data for variables not specified will be dropped. For mandatory columns (such as *msLevel*, *rtime* ...) only the values will be dropped, while additional (user defined) spectra variables will be completely removed. Returns the filtered `Spectra`.
- `split`: splits the `Spectra` object based on parameter `f` into a `list` of `Spectra` objects.

Several `Spectra` objects can be concatenated into a single object with the `c` function. Concatenation will fail if the processing queue of any of the `Spectra` objects is not empty or if different backends are used in the `Spectra` objects. The spectra variables of the resulting `Spectra` object is the union of the spectra variables of the individual `Spectra` objects.

Data manipulation and analysis methods

Many data manipulation operations, such as those listed in this section, are not applied immediately to the spectra, but added to a *lazy processing/manipulation queue*. Operations stored in this queue are applied on-the-fly to spectra data each time it is accessed. This lazy execution guarantees the same functionality for `Spectra` objects with any backend, i.e. backends supporting to save changes

to spectrum data (MsBackendDataFrame() or MsBackendHdf5Peaks()) as well as read-only backends (such as the MsBackendMzR()). Note that for the former it is possible to apply the processing queue and write the modified peak data back to the data storage with the applyProcessing function.

- 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 .... Examples are provided in the package vignette.
- 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).
- bin: aggregates individual spectra into discrete (m/z) bins. All intensity values for peaks falling into the same bin are summed.
- combineSpectra: combine sets of spectra into a single spectrum per set. For each spectrum group (set), spectra variables from the first spectrum are used and the peak matrices are combined using the function specified with FUN, which defaults to combinePeaks(). The sets of spectra can be specified with parameter f. In addition it is possible to define, with parameter p if and how to split the input data for parallel processing. This defaults to p = x$dataStorage and hence a per-file parallel processing is applied for Spectra with file-based backends (such as the MsBackendMzR()). Prior combination of the spectra all processings queued in the lazy evaluation queue are applied. Be aware that calling combineSpectra on a Spectra object with certain backends that allow modifications might **overwrite** the original data. This does not happen with a MsBackendDataFrame backend, but with a MsBackendHdf5Peaks backend the m/z and intensity values in the original hdf5 file(s) will be overwritten. The function returns a Spectra of length equal to the unique levels of f.
- compareSpectra: compare each spectrum in x with each spectrum in y using the function provided with FUN (defaults to 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). 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. Additional parameters to functions FUN and MAPFUN can be passed with .... The function returns a matrix with the results of FUN for each comparison, number of rows equal to length(x) and number of columns equal length(y) (i.e. element in row 2 and column 3 is the result from the comparison of x[2] with y[3]). If SIMPLIFY = TRUE the matrix is *simplified* to a numeric if length of x or y is one.
- filterIntensity: filters each spectrum keeping only peaks 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).

- spectrapply: apply a given function to each spectrum in a Spectra object. The Spectra is splitted into individual spectra and on each of them (i.e. Spectra of length 1) the function FUN is applied. Additional parameters to FUN can be passed with the ... argument. Parameter BPPARAM allows to enable parallel processing, which however makes only sense if FUN is computational intense. spectrapply returns a list (same length than object) with the result from FUN. See examples for more details. Note that the result and its order depends on the factor f used for splitting object with split, i.e. no re-ordering or unsplit is performed on the result.
- smooth: smooth 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().
- 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).

Author(s)

Sebastian Gibb, Johannes Rainer

Examples

```
## 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

## Get the number of spectra
length(data)

## Get the number of peaks per spectrum
lengths(data)

## Create a Spectra from mzML files and use the `MsBackendMzR` on-disk
## backend.
sciex_file <- dir(system.file("sciex", package = "msdata"),
    full.names = TRUE)
sciex <- Spectra(sciex_file, backend = MsBackendMzR())
sciex

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

## 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))

## ---- ACCESSING AND ADDING DATA ----

## Get the MS level for each spectrum.
msLevel(data)

## 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).
data$msLevel

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

```
## Determine whether one of the spectra has a specific m/z value
containsMz(data, mz = 120.4)

## Accessing spectra variables works for all backends:
intensity(sciex)
intensity(sciex_im)

## Get the m/z for the first spectrum.
mz(data)[[1]]

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

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

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

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

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

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

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

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

## Drop spectra variable data and/or columns.
res <- selectSpectraVariables(data, c("mz", "intensity"))

## This removed the additional columns "spectrum_id" and deleted all values
## for all spectra variables, except "mz" and "intensity".
spectraData(res)

## Compared to the data before selectSpectraVariables.
spectraData(data)


## ---- SUBSETTING, FILTERING AND COMBINING

## Subset to all MS2 spectra.
```

```
data[msLevel(data) == 2]

## Same with the filterMsLevel function
filterMsLevel(data, 2)

## Below we combine the `data` and `sciex_im` objects into a single one.
data_comb <- c(data, sciex_im)

## The combined Spectra contains a union of all spectra variables:
head(data_comb$spectrum_id)
head(data_comb$rtime)
head(data_comb$dataStorage)
head(data_comb$dataOrigin)

## Filter a Spectra for a target precursor m/z with a tolerance of 10ppm
spd$precursorMz <- c(323.4, 543.2302)
data_filt <- Spectra(spd)
filterPrecursorMz(data_filt, mz = 543.23 + ppm(c(-543.23, 543.23), 10))

## Filter a Spectra keeping only peaks matching certain m/z values
sps_sub <- filterMzValues(data, mz = c(103, 104), tolerance = 0.3)
mz(sps_sub)

## Filter a Spectra keeping only peaks within a m/z range
sps_sub <- filterMzRange(data, mz = c(100, 300))
mz(sps_sub)

## Remove empty spectra variables
sciex_noNA <- dropNaSpectraVariables(sciex)

## Available spectra variables before and after dropNaSpectraVariables
spectraVariables(sciex)
spectraVariables(sciex_noNA)


## ---- DATA MANIPULATIONS AND OTHER OPERATIONS ----

## Set the data to be centroided
centroided(data) <- TRUE

## Replace peak intensities below 40 with 3.
res <- replaceIntensitiesBelow(data, threshold = 40, value = 3)
res

## Get the intensities of the first and second spectrum.
intensity(res)[[1]]
intensity(res)[[2]]

## Remove all peaks with an intensity below 40.
res <- filterIntensity(res, intensity = c(40, Inf))

## Get the intensities of the first and second spectrum.
intensity(res)[[1]]
intensity(res)[[2]]

## Lengths of spectra is now different
lengths(mz(res))
```

```
lengths(mz(data))

## 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
}
res2 <- filterIntensity(data, intensity = keep_peaks)
intensity(res2)[[1L]]
intensity(data)[[1L]]

## 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:
res2 <- filterIntensity(data, intensity = keep_peaks, prop = 2)
intensity(res2)[[1L]]
intensity(data)[[1L]]

## Since data manipulation operations are by default not directly applied to
## the data but only added to the internal lazy evaluation queue, it is also
## possible to remove these data manipulations with the `reset` function:
res_rest <- reset(res)
res_rest
lengths(mz(res_rest))
lengths(mz(res))
lengths(mz(data))

## `reset` after a `applyProcessing` can not restore the data, because the
## data in the backend was changed. Similarly, `reset` after any filter
## operations can not restore data for a `Spectra` with a
## `MsBackendDataFrame`.
res_2 <- applyProcessing(res)
res_rest <- reset(res_2)
lengths(mz(res))
lengths(mz(res_rest))


## Compare spectra: comparing spectra 2 and 3 against spectra 10:20 using
## the normalized dotproduct method.
res <- compareSpectra(sciex_im[2:3], sciex_im[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

## To use a simple Pearson correlation instead we can define a function
## that takes the two peak matrices and calculates the correlation for
## their second columns (containing the intensity values).
correlateSpectra <- function(x, y, use = "pairwise.complete.obs", ...) {
    cor(x[, 2], y[, 2], use = use, ...)
}
res <- compareSpectra(sciex_im[2:3], sciex_im[10:20],
    FUN = correlateSpectra)
res

## 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(sciex_im[2:3], sciex_im[10:20], ppm = 10, type = "inner",
    FUN = function(x, y, ...) nrow(x))

## 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 `$` operator or the
## corresponding method.
res <- spectrapply(sciex_im[1:20], FUN = function(x) mean(x$intensity[[1]]))
head(res)

## It is however important to note that dedicated methods to access the
## data (such as `intensity`) are much more efficient than using `lapply`:
res <- lapply(intensity(sciex_im[1:20]), mean)
head(res)


## ---- 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)
```

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

Description

combinePeaks 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 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.

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

```
combinePeaks(
  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 <- combinePeaks(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 <- combinePeaks(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 <- combinePeaks(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
combinePeaks(list(p1, p2, p3), peaks = "intersect")

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

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

## Report the minimal m/z and intensity
```

```
combinePeaks(list(p1, p2, p3), peaks = "intersect", tolerance = 0.1,
    intensityFun = min, mzFun = min)
```

---

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.

Usage

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

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`: `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.
`...`	option parameters.

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

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")
```

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:

- `MsBackendDataFrame`: stores all data in memory using 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'
backendInitialize(object, ...)

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

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

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

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

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

## 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)
```

```
## 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'
filterPrecursorMz(object, mz = numeric())

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

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

## 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 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'
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'
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

MsBackendDataFrame()

## S4 method for signature 'MsBackendDataFrame'
```

```
backendInitialize(object, data, ...)

MsBackendHdf5Peaks()

MsBackendMzR()
```

Arguments

object	Object extending MsBackend.
...	Additional arguments.
value	replacement value for <- methods. See individual method description or expected data type.
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 occurr 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 filterPrecursorMz: numeric(2) with the lower and upper m/z boundary.
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.
acquisitionNum	for filterPrecursorScan: integer with the acquisition number of the spectra to which the object should be subsetted.
rt	for filterRt: numeric(2) defining the retention time range to be used to subset/filter object.
msLevel.	same as msLevel above.
x	Object extending MsBackend.
use.names	For lengths: whether spectrum names should be used.
spectraVariables	For selectSpectraVariables: character with the names of the spectra variables to which the backend should be subsetted.
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.
f	factor defining the grouping to split x. See split().
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).
i	For [: integer, logical or character to subset the object.

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.

Value

See documentation of respective function.

Backend functions

New backend classes **must** extend the base MsBackend class and **have** to implement the following methods:

- [: subset the backend. Only subsetting by element (*row*/i) is allowed
- $, $<-: 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).
- peaksData returns a list with the spectras' peak data. The length of the list is equal to the number of spectra in object. Each element of the list is a matrix with columns "mz" and "intensity". For an empty spectrum, a matrix with 0 rows and two columns (named mz and intensity) is returned.
- 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). This method has to ensure to set the spectra variable dataStorage correctly.
- backendMerge: merges (combines) MsBackend objects into a single instance. All objects to be merged have to be of the same type (e.g. MsBackendDataFrame()).
- 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 a dataStorage of NA_character_ is not supported.
- 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.
- 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.
- 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.

- 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.
- 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.
- filterPrecursorScan: retains parent (e.g. MS1) and children scans (e.g. MS2) of acquisition number acquisitionNum. 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]. Implementation of this method is optional since a default implementation for MsBackend is available.
- intensity: gets the intensity values from the spectra. Returns a 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 `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:::.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.
- `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.
- `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 `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<- replaces the peak data (m/z and intensity values) of the backend. This method expects a `list` of `matrix` objects with columns `"mz"` and `"intensity"` that has the same length as the number of spectra in the backend. Note that just writeable backends support this method.
- `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.
- 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".
- spectraNames: returns a character vector with the names of the spectra in object.
- 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.
- 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.

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.

MsBackendDataFrame, in-memory MS data backend

The MsBackendDataFrame objects keep all MS data in memory.

New objects can be created with the MsBackendDataFrame() function. The backend can be subsequently initialized with the backendInitialize method, taking a DataFrame with the MS data as parameter. 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": NumericList() of numeric vectors representing the m/z values for each spectrum.
- "intensity": NumericList() of numeric vectors representing the intensity values for each spectrum.

Additional columns are allowed too.

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.

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.

Implementation notes

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

The MsBackend defines the following slots:

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

Author(s)

Johannes Rainer, Sebastian Gibb, Laurent Gatto

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)
```

`ProcessingStep` *Processing step*

Description

Class containing the function and arguments to be applied in a lazy-execution framework.

Objects of this class are created using the `ProcessingStep()` function. The processing step is executed with the `executeProcessingStep()` function.

Usage

```
ProcessingStep(FUN = character(), ARGS = list())

executeProcessingStep(object, ...)
```

Arguments

`FUN`	`function` or `character` representing a function name.
`ARGS`	`list` of arguments to be passed along to `FUN`.
`object`	`ProcessingStep` object.
`...`	optional additional arguments to be passed along.

Details

This object contains all relevant information of a data analysis processing step, i.e. the function and all of its arguments to be applied to the data. This object is mainly used to record possible processing steps of a `Spectra()` object.

Value

The `ProcessingStep` function returns and object of type `ProcessingStep`.

Author(s)

Johannes Rainer

Examples

```
## Create a simple processing step object
ps <- ProcessingStep(sum)

executeProcessingStep(ps, 1:10)
```

---

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 `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 = character(),
  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 = character(),
  labelCex = 1,
  labelSrt = 0,
  labelAdj = NULL,
  labelPos = NULL,
  labelOffset = 0.5,
  labelCol = "#00000080",
  axes = TRUE,
  frame.plot = axes,
  ...
)

plotSpectraMirror(
  x,
  y,
  xlab = "m/z",
  ylab = "intensity",
  type = "h",
  xlim = numeric(),
  ylim = numeric(),
  main = character(),
  col = "#00000080",
  labels = character(),
  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. Can be a character with length equal to the number of peaks, 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 common() for more details).
tolerance	for plotSpectraMirror: absolute acceptable difference of m/z values for peaks to be considered matching (see 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

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) format(unlist(mz(z)), digits = 4))

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

## Add a custom annotation for each peak.
sp$label <- list(c("", "A", "B", "C", "D"),
    c("Frodo", "Bilbo", "Peregrin", "Samwise"))
## Plot each peak in a different color
plotSpectra(sp, labels = function(z) unlist(z$label),
```

```
    col = list(1:5, 1:4))

## Plot a single spectrum specifying the label.
plotSpectra(sp[2], labels = 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) {
        lbls <- format(mz(z)[[1L]], digits = 4)
        lbls[intensity(z)[[1L]] <= 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) 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) format(mz(z)[[1L]], digits = 3),
    labelSrt = -30, labelPos = 2, labelOffset = 0.2, tolerance = 0.1,
    matchCol = "#ff000080", matchLwd = 2)
grid()
```

Index