patRoon handbook

Rick Helmus

2020-12-04

Contents

1 Introduction

Nowadays there are various software tools available to process data from non-target analysis (NTA) experiments. Individual tools such as ProteoWizard, XCMS, OpenMS, MetFrag and mass spectrometry vendor tools are often combined to perform a complete data processing workflow. During this workflow, raw data files may undergo pre-treatment (e.g. conversion), chromatographic and mass spectral data are combined to extract so called *features* (or 'peaks') and finally annotation is performed to elucidate chemical identities. The aim of `patRoon` is to harmonize the many available tools in order to provide a consistent user interface without the need to know all the details of each individual software tool and remove the need for tedious conversion of data when multiple tools are used. The name is derived from a Dutch word that means *pattern* and may also be an acronym for *hyPhenated mAss specTROmetry nOn-target aNalysis*. The workflow of non-target analysis is typically highly dependent on several factors such as the analytical instrumentation used and requirements of the study. For this reason, `patRoon` does not enforce a certain workflow. Instead, most workflow steps are optional, are highly configurable and algorithms can easily be mixed or even combined. Furthermore, `patRoon` supplies a straightforward interface to easily inspect, select, visualize and report all data that is generated during the workflow.

The documentation of `patRoon` consists of three parts:

1. A tutorial (accessible at here)
2. This handbook
3. The reference manual (accessible in `R` with `` ?`patRoon-package` `` or online here)

New users are highly recommended to start with the tutorial: this document provides an interactive introduction in performing a basic NTA processing workflow with `patRoon`. The handbook provides a more thorough overview of all concepts, functionalities and provides instructions and many examples on working with `patRoon`. Finally, the reference manual provides all the gritty details for all functionalities, and is meant if you want to know more details or need a quick reminder how a function should be used.

2 Installation

`patRoon` depends on various other software tools to perform the non-target analysis workflow steps and to implement various other functionality. Most of these dependencies are automatically installed when you install the `patRoon` R package, however, some may need to be manually installed and/or configured.

> ***NOTE*** It is highly recommended to perform installation steps in a 'clean' `R` session to avoid errors when installing or upgrading packages. As such it is recommended to close all open (R Studio) sessions and open a plain R console to perform the installation.

2.1 Automatic installation (Windows only)

An installation script is provided that automatically installs and configures all dependencies and finally installs `patRoon` itself. At this moment, this script only works with Microsoft Windows. You don't have to install anything else to use it, simply open `R` and execute these commands:

```r
source("https://raw.githubusercontent.com/rickhelmus/patRoon/master/install_patRoon.R")
installPatRoon()
```

A simple text based wizard will start and asks you what to install and how to do it. You can re-run this installer at any time, for instance, if something went wrong or you want to install additional dependencies.

2.2 Docker image (experimental)

Docker images are provided to easily install a reproducible environment with `R`, `patRoon` and nearly all of its dependencies. This section assumes you have a basic understanding of Docker and have it installed. If not, please refer to the many guides available on the Internet. The Docker images of `patRoon` were originally only used for automated testing, however, since these contain a complete working environment of `patRoon` they are also suitable for using the software. They come with all external dependencies (except ProteoWizard), `R` dependencies and `MetFrag` libraries. Furthermore, as of recently the Docker image now also contains RStudio server, which makes using `patRoon` even easier. This feature is still experimental and may change over time. If you find bugs or suggestion please file a bug report!

Below are some example shell commands on how to run the image.

```
# run an interactive R console session
docker run --rm -it patroonorg/patroonrs

# run a linux shell, from which R can be launched
docker run --rm -it patroonorg/patroonrs bash

# run rstudio server, accessible from localhost:8787
# login with rstudio/yourpasswordhere
docker run --rm -p 8787:8787 -u 0 -e PASSWORD=yourpasswordhere patroonorg/patroonrs /init

# same as above, but mount a local directory (~/myvolume) as local volume so it can be
↪  used for persistent storage
# please ensure that ~/myvolume exists!
docker run --rm -p 8787:8787 -u 0 -e PASSWORD=yourpasswordhere -v
↪  ~/myvolume:/home/rstudio/myvolume patroonorg/patroonrs /init
```

Note that the first two commands run as the default user `rstudio`, while the last two as `root`. The last commands launch RStudio server. You can access it by browsing to `localhost:8787` and logging with user `rstudio` and the password defined by the `PASSWORD` variable from the command (`yourpasswordhere` in the above example). The last command also links a local volume in order to obtain persistence of files in the container's home directory. The Docker image is based on the excellent work from the rocker project. For more information on RStudio related options see their documentation for the RStudio image.

2.3 Manual installation

The manual installation is for users who don't use Windows or Docker, prefer to do a manual installation or simply want to know what happens behind the scenes. The manual installation consists of three phases:

1. Installing some prerequisite `R` packages
2. Install and configure (non-`R`) dependencies
3. Install `patRoon`

2.3.1 R prerequisites

When installing `patRoon` Windows users have the option to install from a customized (miniCRAN) repository (`patRoonDeps`). This repository provides a central repository for `patRoon` and all its R packages. An advantage is that installation will be faster and you will not need to install Rtools. Note that you will need to have the latest `R` version installed in order to use this repository.

When you decide to use the `patRoonDeps` repository you can simply *skip* this step. **Otherwise** (i.e. you will use regular repositories instead), execute the following:

```
install.packages(c("BiocManager", "remotes"))
BiocManager::install("CAMERA")

# needed for RAMClustR for componentization
remotes::install_github("cbroeckl/RAMClustR")

# only needed for Bruker DataAnalysis integration
install.packages("RDCOMClient", repos = "http://www.omegahat.net/R")

# only when using the R interface (not recommended by default)
remotes::install_github("c-ruttkies/MetFragR/metfRag")
```

Note that the latter three commands concern installation of *optional* packages. If you are unsure then you probably don't need them.

2.3.2 Other dependencies

Depending on which functionality is used, the following external dependencies may need to be installed:

Software	Remarks
Java JDK	**Mandatory** for e.g. plotting structures and using MetFrag.
Rtools	Necessary on Window and when `patRoon` is *not* installed from `patRoonDeps`.
ProteoWizard	Needed for automatic data-pretreatment (e.g. data file conversion and centroiding, Bruker users may use DataAnalysis integration instead).
OpenMS	Recommended. Used for e.g. finding and grouping features.
MetFrag CL	Recommended. Used for annotation with MetFrag.
MetFrag CompTox DB	Database files necessary for usage of the CompTox database with MetFrag. Note that a recent version of MetFrag (>=2.4.5) is required. Note that the lists with additions for smoking metadata and wastewater metadata are also supported.
MetFrag PubChemLite DB	Database files needed to use PubChemLite with MetFrag (currently tested with tier0 and tier1 November 2019 versions).
SIRIUS	For formula and/or compound annotation.
OpenBabel	Used in some cases for suspect screening (e.g. to calculate molecular masses for suspects with only InChI information). Otherwise optional.
pngquant	Used to reduce size of HTML reports, definitely optional.

After installation you may need to configure the file path to ProteoWizard, OpenMS, SIRIUS, MetFrag, the MetFrag CompTox DB and/or pngquant (normally ProteoWizard and OpenMS should be automatically found). To configure their file locations you should set some global package options with the `options()` R function, for instance:

```
options(patRoon.path.pwiz = "C:/ProteoWizard") # location of ProteoWizard installation
↪  folder
options(patRoon.path.SIRIUS = "C:/sirius-win64-3.5.1") # location where SIRIUS was
↪  extracted
options(patRoon.path.OpenMS = "/usr/local/bin") # directory with the OpenMS binaries
options(patRoon.path.pngquant = "~/pngquant") # directory containing pngquant binary
options(patRoon.path.MetFragCL = "~/MetFrag2.4.5-CL.jar") # full location to the jar file
options(patRoon.path.MetFragCompTox = "C:/CompTox_17March2019_SelectMetaData.csv") # full
↪  location to desired CompTox CSV file
options(patRoon.path.MetFragPubChemLite = "~/PubChemLite_14Jan2020_tier0.csv") # full
↪  location to desired PubChemLite CSV file
options(patRoon.path.obabel = "C:/Program Files/OpenBabel-3.0.0") # directory with
↪  OpenBabel binaries
```

These commands have to be executed everytime you start a new R session (e.g. as part of your script). However, it is probably easier to add them to your `~/.Rprofile` file so that they are executed automatically when you start R. If you don't have this file yet you can simply create it yourself (for more information see e.g. this SO answer).

2.3.3 patRoon installation

Finally, it is time to install `patRoon` itself. As mentioned before, Windows users (who have the latest `R` version) can install `patRoon` and all its package dependencies from the `patRoonDeps` repository:

```
# install from patRoonDeps (only Windows with latest R version)
install.packages("patRoon", repos = "https://rickhelmus.github.io/patRoonDeps/", type =
↪  "binary")

# optional, data for tutorial
install.packages("patRoonData", repos = "https://rickhelmus.github.io/patRoonDeps/", type
↪  = "binary")
```

Otherwise, installation occurs directly from GitHub:

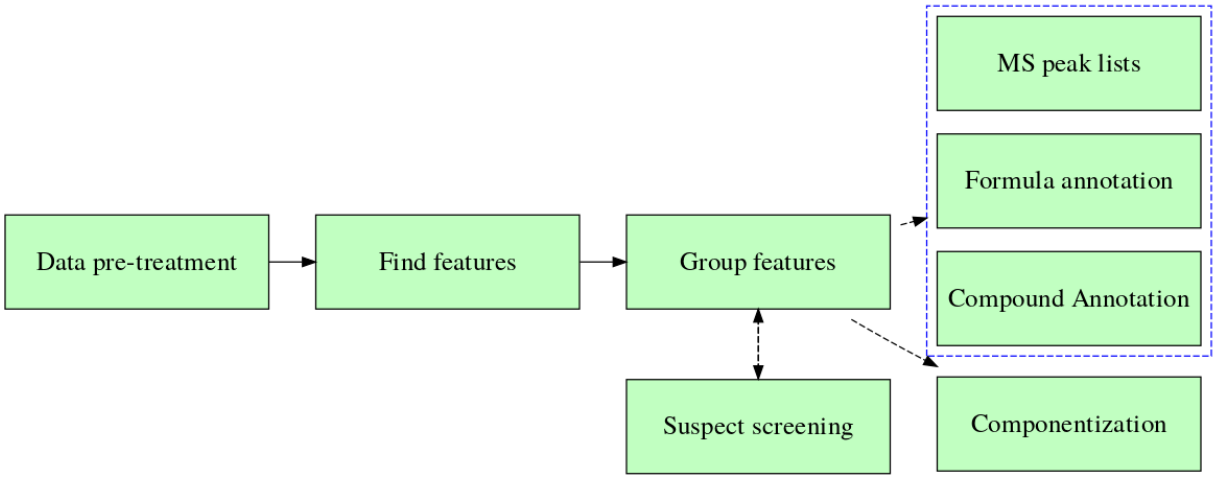
```
remotes::install_github("rickhelmus/patRoon")

# optional, data for tutorial
remotes::install_github("rickhelmus/patRoonData") # example data used by tutorial
```

Afterwards, you can run the `verifyDependencies()` function to see if `patRoon` can find all its dependencies (you may need to restart R beforehand)

```
patRoon::verifyDependencies()
```

3 Workflow concepts

In a non-target workflow both chromatographic and mass spectral data is automatically processed in order to provide a comprehensive chemical characterization of your samples. While the exact workflow is typically dependent on the type of study, it generally involves of the following steps:



Note that `patRoon` supports flexible composition of workflows. In the scheme above you can recognize optional steps by a *dashed line*. The inclusion of each step is only necessary if a further steps depends on its data. For instance, annotation and componentization do not depend on each other and can therefore be executed in any order or simply be omitted. A brief descripton of all steps is given below.

During **data pre-treatment** raw MS data is prepared for further analysis. A common need for this step is to convert the data to an open format so that other tools are able to process it. Other pre-treatment steps may involve re-calibration of m/z data or performing advanced filtering operations.

The next step is to extract **features** from the data. While different terminologies are used, a feature in `patRoon` refers to a single chromatographic peak in an extracted ion chromatogram for a single m/z value (within a defined tolerance). Hence, a feature contains both chromatographic data (e.g. retention time and peak height) and mass spectral data (e.g. the accurate m/z). Note that with mass spectrometry multiple m/z values may be detected for a single compound as a result of adduct formation, natural isotopes and/or in-source fragments. Some algorithms may try to combine these different masses in a single feature. However, in `patRoon` we generally assume this is not the case (and may optionally be done afterwards during the componentization step described below). Features are sometimes simply referred to as 'peaks'.

Features are found per analysis. Hence, in order to compare a feature across analyses, the next step is to group them. This step is essential as it finds equal features even if their retention time or m/z values slightly differ due to analytical variability. The resulting **feature groups** are crucial input for subsequent workflow steps. Prior to grouping, *retention time alignment* between analyses may be performed to improve grouping of features, especially when processing multiple analysis batches at once. Outside `patRoon` feature groups may also be defined as *profiles*, *aligned* or *grouped features* or *buckets*.

Depending on the study type, **suspect screening** is then performed to limit the features that should be considered for further processing. As its name suggests, with suspect screening only those features which are suspected to be present are considered for further processing. These suspects are retrieved from a suspect list which contains the m/z and (optionally) retention times for each suspect. Typical suspect lists may be composed from databases with known pollutants or from predicted transformation products. Note that for a 'full' non-target analysis no suspect screening is performed, hence, this step is simply omitted and all features are to be considered.

After features have been collected the next step typically involves **annotation**. During this step MS and

MS/MS data are collected in so called **MS peak lists**, which are then used as input for formula and compound annotation. Formula annotation involves automatic calculation of possible formulae for each feature based on its *m/z*, isotopic pattern and MS/MS fragments, whereas compound annotation (or identification) involves the assignment of actual chemical structures to each feature. Note that during formula and compound annotation typically multiple candidates are assigned to a single feature. To assist interpretation of this data each candidate is therefore ranked on characteristics such as isotopic fit, number of explained MS/MS fragments and metadata from an online database such as number of scientific references or presence in common suspect lists.

Besides annotation, another step to perform after extraction of features is **componentization**. A **component** is defined as a collection of multiple feature groups that are somehow related to each other. Typical exmples are features that belong to the same chemical compound (i.e. with different *m/z* values but equal retention time), such as adducts, isotopes and in-source fragments. Other examples are homologues series and features that display a similar intensity trend across samples.

To summarize:

- **Data-pretreatment** involves preparing raw MS data for further processing (e.g. conversion to an open format)
- **Features** describe chromatographic and *m/z* information (or 'peaks') in all analyses.
- A **feature group** consists of equal features across analyses.
- With **suspect screening** only features that are considered to be on a suspect list are considered further in the workflow.
- **MS peak lists** Summarizes all MS and MS/MS data that will be used for subsequent annotation.
- During **formula** and **compound annotation** candidate formulae/structures will be assigned and ranked for each feature.
- **Componentization** involves consolidating different feature groups that have a relationship to each other in to a single component.

The next chapters will discuss how to generate this data and process it.

4 Generating workflow data

Each step in the non-target workflow is performed by a function that performs the heavy lifting of a workflow step behind the scenes and finally return the results. An important goal of `patRoon` is to support multiple algorithms for each workflow step, hence, when such a function is called you have to specify which algorithm you want to use. The available algorithms and their characteristics will be discussed in the next sections. An overview of all functions involved in generating workflow data is shown in the table below.

Workflow step	Function	Output S4 class
Data pre-treatment	`convertMSFiles()`, `recalibrarateDAFiles()`	-
Finding features	`findFeatures()`	`features`
Grouping features	`groupFeatures()`	`featureGroups`
Suspect screening	`screenSuspects()` + `groupFeaturesScreening()`	`featureGroups`
MS peak lists	`generateMSPeakLists()`	`MSPeakLists`
Formula annotation	`generateFormulas()`	`formulas`
Compound annotation	`generateCompounds()`	`compounds`
Componentization	`generateComponents()`	`components`

All of these functions store their output in objects derived from so called S4 classes. Knowing the details about the S4 class system of `R` is generally not important when using `patRoon` (and well written resources

are available if you want to know more). In brief, usage of this class system allows a general data format that is used irrespective of the algorithm that was used to generate the data. For instance, when features have been found by OpenMS or XCMS they both return the same data format.

Another advantage of the S4 class system is the usage of so called *generic functions*. To put simply: a generic function performs a certain task for different types of data objects. A good example is the `plotSpectrum()` function which plots an (annotated) spectrum from data of MS peak lists or from formula or compound annotation:

```
# mslists, formulas, compounds contain results for MS peak lists and
# formula/compound annotations, respectively.

plotSpectrum(mslists, ...) # plot raw MS spectrum
plotSpectrum(formulas, ...) # plot annotated spectrum from formula annotation data
plotSpectrum(compounds, ...) # likewise but for compound annotation.
```

The next sections will further detail on how to actually perform the non-target workflow steps to generate data.

4.1 Preparations

4.1.1 Data pre-treatment

Prior to performing the actual non-target data processing workflow some preparations often need to be made. Often data has to be pre-treated, for instance, by converting it to an open format that is usable for subsequent workflow steps or to perform mass re-calibration. Some common functions are listed below.

Task	Function	Algorithms	Supported file formats
Conversion	`convertMSFiles()`	OpenMS, ProteoWizard, DataAnalysis	All common (algorithm dependent)
Advanced (e.g. spectral filtering)	`convertMSFiles()`	ProteoWizard	All common
Mass re-calibration	`recalibrarateDAFiles()`	DataAnalysis	Bruker

The `convertMSFiles()` function supports conversion between many different file formats typically used in non-target analysis. Furthermore, other pre-treatment steps are available (e.g. centroiding, filtering) when the ProteoWizard algorithm is used. For an overview of these functionalities see the MsConvert documentation. Some examples:

```
# Converts a single mzXML file to mzML format
convertMSFiles("standard-1.mzXML", to = "mzML", algorithm = "openms")

# Converts all Thermo files with ProteoWizard (the default) in the analyses/raw
# directory and stores the mzML files in analyses/raw. Afterwards, only MS1
# spectra are retained.
```

```
convertMSFiles("analyses/raw", "analyses/mzml", from = "thermo",
               centroid = "vendor", filters = "msLevel 1")
```

> ***NOTE*** Most algorithms further down the workflow require the *mzML* or *mzXML* file format and additionally require that mass peaks have been centroided. When using the ProteoWizard algorithm (the default), centroiding by vendor algorithms is generally recommended (i.e. by setting `centroid="vendor"` as shown in the above example).

When Bruker MS data is used it can be automatically re-calibrated to improve its mass accuracy. Often this is preceeded by calling the `setDAMethod()` function to set a DataAnalysis method to all files in order to configure automatic re-calibration. The `recalibrarateDAFiles()` function performs the actual re-calibration. The `getDACalibrationError()` function can be used at anytime to request the current calibration error of each analysis. An example of these functions is shown below.

```
# anaInfo is a data.frame with information on analyses (see next section)
setDAMethod(anaInfo, "path/to/DAMethod.m") # configure Bruker files with given method
↪  that has automatic calibration setup
recalibrarateDAFiles(anaInfo) # trigger re-calibration for each analysis
getDACalibrationError(anaInfo) # get calibration error for each analysis (NOTE: also
↪  shown when previous function is finished)
```

4.1.2 Analysis information

The final bits of preparation is constructing the information for the analyses that need to be processed. In `patRoon` this is referred to as the *analysis information* and often stored in a variable `anaInfo` (of course you are free to choose a different name!). The analysis information should be a `data.frame` with the following columns:

- **path**: the directory path of the file containing the analysis data
- **analysis**: the name of the analysis. This should be the file name *without* file extension.
- **group**: to which *replicate group* the analysis belongs. All analysis which are replicates of each other get the same name.
- **blank**: which replicate group should be used for blank subtraction.
- **conc** (optional, advanced) A numeric value describing the concentration or any other value for which the intensity in this sample may correlate, for instance, dilution factor, sampling time etc. This column is only required when you want to obtain quantitative information (e.g. concentrations) using the `as.data.table()` method function (see `?featureGroups` for more information).

The `generateAnalysisInfo()` function can be used to (semi-)automatically generate a suitable `data.frame` that contains all the required information for a set of analysis. For, instance, the following line was used in the tutorial:

```
# Take example data from patRoonData package (triplicate solvent blank + triplicate
↪  standard)
generateAnalysisInfo(paths = patRoonData::exampleDataPath(),
                     groups = c(rep("solvent", 3), rep("standard", 3)),
                     blanks = "solvent")
```

```
#>                                              path   analysis    group   blank
#> 1 /usr/local/lib/R/site-library/patRoonData/extdata  solvent-1  solvent solvent
```

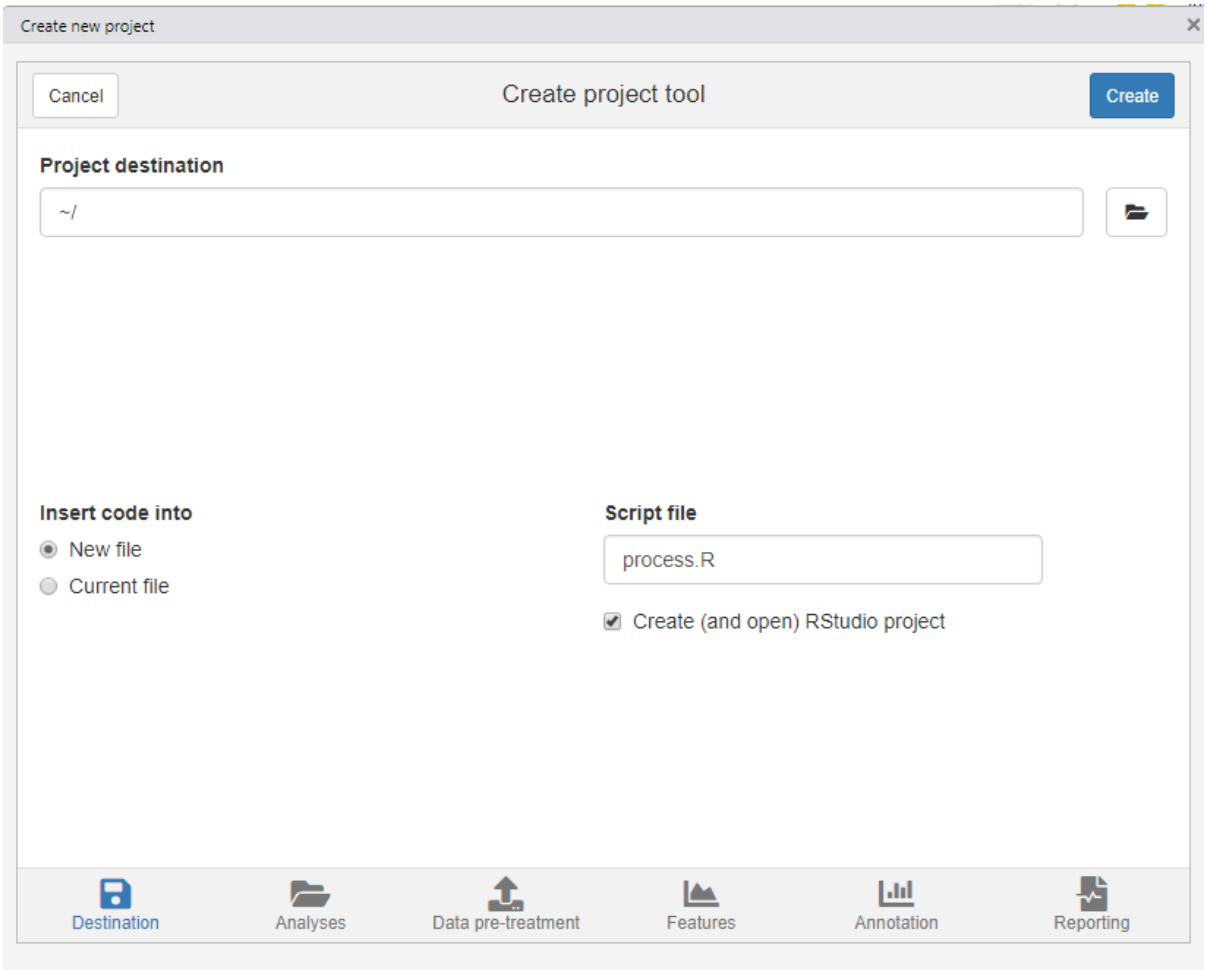
```
#> 2 /usr/local/lib/R/site-library/patRoonData/extdata  solvent-2  solvent solvent
#> 3 /usr/local/lib/R/site-library/patRoonData/extdata  solvent-3  solvent solvent
#> 4 /usr/local/lib/R/site-library/patRoonData/extdata standard-1 standard solvent
#> 5 /usr/local/lib/R/site-library/patRoonData/extdata standard-2 standard solvent
#> 6 /usr/local/lib/R/site-library/patRoonData/extdata standard-3 standard solvent
```

Alternatively, the `newProject()` function discussed in the next section can be used to interactively construct this information.

4.1.3 Automatic project generation with newProject()

The previous sections already highlighted some steps that have to be performed prior to starting a new non-target analysis workflow: data pre-treatment and gathering information on the analysis. Most of the times you will put this and other `R` code a script file so you can re-call what you have done before (i.e. reproducible research).

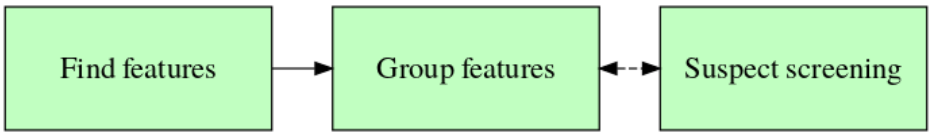
The `newProject()` function can be used to setup a new project. When you run this function it will launch a small tool (see screenshot below) where you can select your analyses and configure the various workflow steps which you want to execute (e.g. data pre-treatment, finding features, annotation etc). After setting everything up the function will generate a template script which can easily be edited afterwards. In addition, you have the option to create a new RStudio project, which is advantegeous as it neatly seperates your data processing work from the rest.



NOTE At the moment `newProject()` *only* works with (recent) versions of RStudio.

4.2 Features

Collecting features from the analyses consists of finding all features, grouping them across analyses (optionally after retention time alignment) and finally, if desired suspect screening:



4.2.1 Finding and grouping features

Several algorithms are available for finding features. These are listed in the table below alongside their usage and general remarks.

Algorithm	Usage	Remarks
OpenMS	`findFeatures(algorithm = "openms", ...)`	Uses the FeatureFinderMetabo algorithm
XCMS	`findFeatures(algorithm = "xcms", ...)`	Uses `xcms::xcmsSet()` function
XCMS (import)	`importFeatures(algorithm = "xcms", ...)`	Imports an existing `xcmsSet` object.
XCMS3	`findFeatures(algorithm = "xcms3", ...)`	Uses `xcms::findChromPeaks()` from the new XCMS3 interface
XCMS3 (import)	`importFeatures(algorithm = "xcms3", ...)`	Imports an existing `XCMSnExp` object.
enviPick	`findFeatures(algorithm = "envipick", ...)`	Uses `enviPick::enviPickwrap()`
DataAnalysis	`findFeatures(algorithm = "bruker", ...)`	Uses Find Molecular Features from DataAnalysis (Bruker only)

Most often the performance of these algorithms heavily depend on the data and parameter settings that are used. Since obtaining a good feature dataset is crucial for the rest of the workflow, it is highly recommended to experiment with different settings (this process can also be automated, see the feature optimization section for more details). Some common parameters to look at are listed in the table below. However, there are many more (advanced) parameters that can be set, please see the reference documentation for these (e.g. `?findFeatures`).

Algorithm	Common parameters
OpenMS	`noiseThrInt`, `chromSNR`, `chromFWHM`, `mzPPM`, `minFWHM`, `maxFWHM` (see `?findFeatures`)

Algorithm	Common parameters
XCMS / XCMS3	`peakwidth`, `mzdiff`, `prefilter`, `noise` (assuming default `centWave` algorithm, see `?findPeaks.centWave` / `?CentWaveParam`)
enviPick	`dmzgap`, `dmzdens`, `drtgap`, `drtsmall`, `drtdens`, `drtfill`, `drttotal`, `minpeak`, `minint`, `maxint` (see `?enviPickwrap`)
DataAnalysis	See *Find -> Parameters... -> Molecular Features* in DataAnalysis.

NOTE DataAnalysis feature settings have to be configured in DataAnalysis prior to calling `findFeatures()`.

Similarly, for grouping features across analyses several algorithms are supported.

Algorithm	Usage	Remarks
OpenMS	`groupFeatures(algorithm = "openms", ...)`	Uses the FeatureLinkerUnlabeled algorithm (and MapAlignerPoseClustering for retention alignment)
XCMS	`groupFeatures(algorithm = "xcms", ...)`	Uses `xcms::group()` `xcms::retcor()` functions
XCMS (import)	`importFeatureGroupsXCMS(...)`	Imports an existing `xcmsSet` object.
XCMS3	`groupFeatures(algorithm = "xcms3", ...)`	Uses `xcms::groupChromPeaks()` and `xcms::adjustRtime()` functions
XCMS3 (import)	`importFeatureGroupsXCMS3(...)`	Imports an existing `XCMSnExp` object.
ProfileAnalysis	`importFeatureGroups(algorithm = "brukerpa", ...)`	Import `.csv` file exported from Bruker ProfileAnalysis
TASQ	`importFeatureGroups(algorithm = "brukertasq", ...)`	Imports a *Global result table* (exported to Excel file and then saved as `.csv` file)

Just like finding features, each algorithm has their own set of parameters. Often the defaults are a good start but it is recommended to have look at them. See `?groupFeatures` for more details.

When using the XCMS algorithms both the 'classical' interface and latest `XCMS3` interfaces are supported. Currently, both interfaces are mostly the same regarding functionalities and implementation. However, since future developments of XCMS are primarily focused the latter this interface is recommended.

Some examples of finding and grouping features are shown below.

```
# The anaInfo variable contains analysis information, see the previous section

# Finding features
fListOMS <- findFeatures(anaInfo, "openms") # OpenMS, with default settings
fListOMS2 <- findFeatures(anaInfo, "openms", noiseThrInt = 500, chromSNR = 10) # OpenMS,
↪  adjusted minimum intensity and S/N
fListXCMS <- findFeatures(anaInfo, "xcms", ppm = 10) # XCMS
fListXCMSImp <- importFeatures(anaInfo, "xcms", xset) # import XCMS xcmsSet object
fListXCMS3 <- findFeatures(anaInfo, "xcms3", CentWaveParam(peakwidth = c(5, 15))) # XCMS3
fListEP <- findFeatures(anaInfo, "envipick", minint = 1E3) # enviPick
```

```
# Grouping features
fGroupsOMS <- groupFeatures(fListOMS, "openms") # OpenMS grouping, default settings
fGroupsOMS2 <- groupFeatures(fListOMS2, "openms", rtalign = FALSE) # OpenMS grouping, no
↪  RT alignment
fGroupsOMS3 <- groupFeatures(fListXCMS, "openms", maxGroupRT = 6) # group XCMS features
↪  with OpenMS, adjusted grouping parameter
# group enviPick features with XCMS3, disable minFraction
fGroupsXCMS <- groupFeatures(fListEP, "xcms3",
                             xcms::PeakDensityParam(sampleGroups = analInfo$group,
                                                    minFraction = 0))
```

4.2.2 Suspect screening

After features have been grouped suspect screening may be performed to eliminate any feature groups with *m/z* and (optionally) retention properties not present in a given suspect list. A typical workflow looks like this:

```
suspects <- data.frame(name = c("susp1", "susp2", "susp3"),
                       mz = c(100.1034, 250.0456, 300.1234),
                       stringsAsFactors = FALSE)
scr <- screenSuspects(fGroups, suspects)
fGroupsSusp <- groupFeaturesScreening(fGroups, scr)
```

This will perform the following steps:

1. Create a suspect list. A suspect list is a `data.frame` with mandatory `name` and `mz` columns. Optionally, it may also contain an `rt` column for retention times (in seconds). In reality loading such a table is easiest from a *.csv* file with e.g. `read.csv()`.
2. Screen a feature groups object for the suspects. The return value will be a `data.frame` with results.
3. Use the screening results to filter the original feature groups and store the results in `fGroupsSusp`.

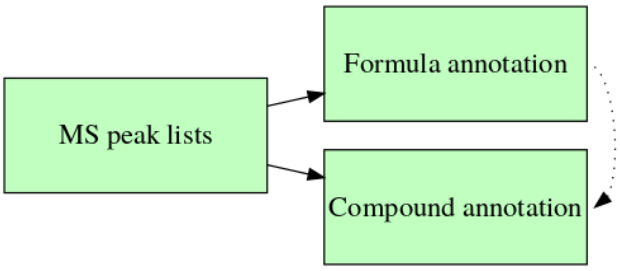
Alternatively, the `mz` column of the input suspect list can also be substituted by a `neutralMass`, `SMILES`, `InChI` or `formula` column, so that ion masses don't need to be calculated in advance. This is especially advantageous for e.g. NORMAN suspect lists where such information is readily available. A requirement is that the adduct species should be specified for automatic ion mass calculation. This can either be done with an extra `adduct` column in the suspect list, or by using the `adduct` function argument to `screenSuspects`, e.g.:

```
suspects <- data.frame(name = c("susp1", "susp2", "susp3"),
                       SMILES = c("C1=CC=C(C=C1)C(=O)O",
                       ↪  "C1=CC=C2C(=C1)C=CC3=CC=CC=C3N2C(=O)N", "C1=CC2=NNN=C2C=C1"),
                       stringsAsFactors = FALSE)
scr <- screenSuspects(fGroups, suspects, adduct = "[M+H]+")
fGroupsSusp <- groupFeaturesScreening(fGroups, scr)
```

Note that in some cases you may need to install OpenBabel (e.g. when only InChI data is available for mass calculation).

4.3 Annotation

The annotation consists of collecting MS peak lists and then formula and/or compound annotation:



Note that compound annotation is normally not dependent upon formula annotation. However, formula data can be used to improve ranking of candidates afterwards by the `addFormulaScoring()` function, which will be discussed later in this section.

4.3.1 MS peak lists

Algorithm	Usage	Remarks
mzR	`generateMSPeakLists(algorithm = "mzr", ...)`	Uses mzR for spectra retrieval. Recommended default.
DataAnalysis	`generateMSPeakLists(algorithm = "bruker", ...)`	Loads data after automatically generating MS and MS/MS spectra in DataAnalysis
DataAnalysis FMF	`generateMSPeakLists(algorithm = "brukerfmf", ...)`	Uses spectra from the *find molecular features* algorithm.

The recommended default algorithm is `mzr`: this algorithm is generally faster and is not limited to a vendor data format as it will read the open `mzML` and `mzXML` file formats. On the other hand, when DataAnalysis is used with Bruker data the spectra can be automatically background subtracted and there is no need for file conversion. Note that the `brukerfmf` algorithm only works when `findFeatures()` was called with the `bruker` algorithm.

When `generateMSPeakists()` is called it will

1. Find all MS and MS/MS spectra that 'belong' to a feature. For MS spectra this means that all spectra close to the retention time of a feature will be collected. In addition, for MS/MS normally only spectra will be considered that have a precursor mass close to that of the feature (however, this can be disabled for data that was recorded with data independent acquisition (DIA, MSˆE, bbCID, ...)).
2. Average all MS and MS/MS spectra to produce peak lists for each feature.
3. Average all peak lists for features within the same group.

Data from either (2) or (3) is used for subsequent annotation steps. Formula calculation can use either (as a trade-off between possibly more accurate results by outlier removal *vs* speed), whereas compound annotation will always use data from (3) since annotating single features (as opposed to their groups) would take a very long time.

There are several common function arguments to `generateMSPeakLists()` that can be used to optimize its behaviour:

Argument	Algorithm(s)	Remarks
`maxMSRtWindow`	`mzr`, `bruker`	Maximum time window +/- the feature retention time (in seconds) to collect spectra for averaging. Higher values may significantly increase processing times.
`precursorMzWindow`	`mzr`	Maximum precursor *m/z* search window to find MS/MS spectra. Set to `NULL` to disable (i.e. for DIA experiments).
`topMost`	`mzr`	Only retain feature data for no more than this amount analyses with highest intensity. For instance, a value of *1* will only keep peak lists for the feature with highest intensity in a feature group.
`bgsubtr`	`bruker`	Perform background subtraction (if the spectra type supports this, e.g. MS and bbCID)
`minMSIntensity`, `minMSMSIntensity`	`bruker`, `brukerfmf`	Minimum MS and MS/MS intensity. Note that DataAnalysis reports many zero intensity peaks so a value of at least *1* is recommended.
`MSMSType`	`bruker`	The type of spectra that should be used for MSMS: `"BBCID"` for bbCID experiments, otherwise `"MSMS"` (the default).

In addition, several parameters can be set that affect spectral averaging. These parameters are passed as a `list` to the `avgFeatParams` (`mzr` algorithm only) and `avgFGroupParams` arguments, which affect averaging of feature and feature group data, respectively. Some typical parameters include:

- `clusterMzWindow`: Maximum *m/z* window used to cluster mass peaks when averaging. The better the MS resolution, the lower this value should be.
- `topMost`: Retain no more than this amount of most intense mass peaks. Useful to filter out 'noisy' peaks.
- `minIntensityPre` / `minIntensityPost`: Mass peaks below this intensity will be removed before/after averaging.

See `?generateMSPeakLists` for all possible parameters.

A suitable list object to set averaging parameters can be obtained with the `getDefAvgPListParams()` function.

```
# lower default clustering window, other settings remain default
avgPListParams <- getDefAvgPListParams(clusterMzWindow = 0.001)

# Apply to both feature and feature group averaging
plists <- generateMSPeakLists(fGroups, "mzr", avgFeatParams = avgPListParams,
↪  avgFGroupParams = avgPListParams)
```

4.3.2 Formulae

Formulae can be automatically calculated for all features using the `generateFormulas()` function. The following algorithms are currently supported:

Algorithm	Usage	Remarks
GenForm	`generateFormulas(algorithm = "genform", ...)`	Bundled with `patRoon`. Reasonable default.
SIRIUS	`generateFormulas(algorithm = "sirius", ...)`	Requires MS/MS data.
DataAnalysis	`generateFormulas(algorithm = "bruker", ...)`	Requires FMF features (i.e. `findFeatures(algorithm = "bruker", ...)`). MS peak lists are not needed. Uses *SmartFormula* algorithms.

Calculation with GenForm is often a good default. It is fast and basic rules can be applied to filter out obvious non-existing formulae. A possible drawback of GenForm, however, is that may become slow when many candidates are calculated, for instance, due to a relative high feature *m/z* (e.g. >600) or loose elemental restricitions. More thorough calculation is performed with SIRIUS: this algorithm often yields fewer and often more plausible results. However, SIRIUS requires MS/MS data (hence features without will not have results) and formula prediction may not work well for compounds that structurally deviate from the training sets used by SIRIUS. Calculation with DataAnalysis is only possible when features are obtained with DataAnalysis as well. An advantage is that analysis files do not have to be converted and no MS peak generation is necessary, however, compared to other algorithms calculation is often relative slow.

There are two methods for formula assignment:

1. Formulae are first calculated for each individual feature within a feature group. These results are then pooled, outliers are removed and remaining formulae are assigned to the feature group (i.e. `calculateFeatures = TRUE`).
2. Formulae are directly calculated for each feature group by using group averaged peak lists (see previous section) (i.e. `calculateFeatures = FALSE`).

The first method is more thorough and the possibility to remove outliers may sometimes result in better formula assignment. However, the second method is much faster and generally recommended for large number of analyses.

By default, formulae are either calculated by *only* MS/MS data (SIRIUS) or with both MS *and* MS/MS data (GenForm/Bruker). The latter also allows formula calculation when no MS/MS data is present. Furthermore, with Bruker algorithms, data from both MS and MS/MS formula data can be combined to allow inclusion of candidates that would otherwise be excluded by e.g. poor MS/MS data. However, a disadvantage is that formulae needs to be calculated twice. The `MSMode` argument (listed below) can be used to customize this behaviour.

An overview of common parameters that are typically set to customize formula calculation is listed below.

Argument	Algorithm(s)	Remarks
relMzDev	`genform`, `sirius`	The maximum relative *m/z* deviation for a formula to be considered (in *ppm*).
elements	`genform`, `sirius`	Which elements to consider. By default `"CHNOP"`. Try to limit possible elements as much as possible.
calculateFeatures	`genform`, `sirius`	Whether formulae should be calculated first for all features (see discussion above) (always `TRUE` with DataAnalysis).
featThreshold	All	Minimum relative amount (*0-1*) amongst all features within a feature group that a formula candidate should be present (e.g. *1* means that a candidate is only considered if it was assigned to all features).
adduct	`genform`, `sirius`	The adduct to consider for calculation (e.g. `"[M+H]+"`, `"[M-H]-"`, more details in the adduct section).

Argument	Algorithm(s)	Remarks
MSMode	`genform`, `bruker`	Whether formulae should be generated only from MS data (`"ms"`), MS/MS data (`"msms"`) or both (`"both"`). The latter is default, see discussion above.
profile	`sirius`	Instrument profile, e.g. `"qtof"`, `"orbitrap"`, `"fticr"`.

Some typical examples:

```
formulasGF <- generateFormulas(fGroups, "genform", mslists) # GenForm, default settings
formulasGF2 <- generateFormulas(fGroups, "genform", mslists, calculateFeatures = FALSE) #
↪  direct feature group assignment (faster)
formulasSIR <- generateFormulas(fGroups, "sirius", mslists, elements = "CHNOPSClBr") #
↪  SIRIUS, common elements for pollutant
formulasSIR2 <- generateFormulas(fGroups, "sirius", adduct = "[M-H]-") # SIRIUS, negative
↪  ionization
formulasBr <- generateFormulas(fGroups, "bruker", MSMode = "MSMS") # Only consider MSMS
↪  data (SmartFormula3D)
```

4.3.3 Compounds

An important step in a typical non-target workflow is structural identification for features of interest. Afterall, this information may finally reveal *what* a feature is. The first step is to find all possible structures in a database that may be assigned to the feature (based on e.g. monoisotopic mass or formula). These candidates are then scored to rank likely candidates, for instance, on correspondence with in-silico or library MS/MS spectra and environmental relevance.

Structure assignment in `patRoon` is performed automatically for all feature groups with the `generateCompounds()` function. Currently, this function supports two algorithms:

Algorithm	Usage	Remarks
MetFrag	`generateCompounds(algorithm = "metfrag", ...)`	Supports many databases (including custom) and scorings for candidate ranking.
SIRIUS with CSI:FingerID	`generateCompounds(algorithm = "sirius", ...)`	Incorporates prior comprohensive formula calculations.

Compound annotation is often a relative time and resource intensive procedure. For this reason, features are not annotated individually, but instead a feature group as a whole is annotated, which generally saves significant amounts of computational requirements. Nevertheless, it is not uncommon that this is the most time consuming step in the workflow. For this reason, prioritization of features is highly important, even more so to avoid 'abusing' servers when an online database is used for compound retrieval.

Selecting the right database is important for proper candidate assignment. Afterall, if the 'right' chemical compound is not present in the used database, it is impossible to assign the correct structure. Luckily, however, several large databases such as PubChem and ChemSpider are openly available which contain tens of millions of compounds. On the other hand, these databases may also lead to many unlikely candidates and therefore more specialized (or custom databases) may be preferred. Which database will be used is dictated by the `database` argument to `generateCompounds()`, currently the following options exist:

Database	Algorithm(s)	Remarks
`pubchem`	`"metfrag"`, `"sirius"`	PubChem is currently the largest compound database and is used by default.
`chemspider`	`"metfrag"`	ChemSpider is another large database. Requires security token from here (see next section).
`comptox`	`"metfrag"`	The EPA CompTox contains many compounds and scorings relevant to environmental studies. Needs manual download (see next section).
`pubchemlite`	`"metfrag"`	A specialized subset of the PubChem database. Needs manual download (see next section).
`for-ident`	`"metfrag"`	The FOR-IDENT (STOFF-IDENT) database for water related substances.
`kegg`	`"metfrag"`, `"sirius"`	The KEGG database for biological compounds
`hmdb`	`"metfrag"`, `"sirius"`	The HMDB contains many human metabolites.
`bio`	`"sirius"`	Selects all supports biological databases.
`csv`, `psv`, `sdf`	`"metfrag"`	Custom database (see next section). CSV example.

4.3.3.1 Configuring MetFrag databases and scoring Some extra configuration may be necessary when using certain databases with MetFrag. In order to use the ChemSpider database a security token should be requested and set with the `chemSpiderToken` argument to `generateCompounds()`. The CompTox and PubChemLite databases need to be manually downloaded from CompTox (or variations with smoking or wastewater metadata) and PUbChemLite. The file location of this and other local databases (`csv`, `psv`, `sdf`) needs to be manually configured, see the examples below and/or `?generateCompounds` for more information on how to do this.

```
# PubChem: the default
compsMF <- generateCompounds(fGroups, mslists, "metfrag", adduct = "[M+H]+")

# ChemSpider: needs security token
compsMF2 <- generateCompounds(fGroups, mslists, "metfrag", database = "chemspider",
                              chemSpiderToken = "MY_TOKEN_HERE", adduct = "[M+H]+")

# CompTox: set global option to database path
options(patRoon.path.MetFragCompTox = "~/CompTox_17March2019_SelectMetaData.csv")
compsMF3 <- generateCompounds(fGroups, mslists, "metfrag", database = "comptox", adduct =
↪  "[M+H]+")

# CompTox: set database location without global option
compsMF4 <- generateCompounds(fGroups, mslists, "metfrag", database = "comptox", adduct =
↪  "[M+H]+",
                              extraOpts = list(LocalDatabasePath =
                              ↪  "~/CompTox_17March2019_SelectMetaData.csv"))
```

```
# Same, but for custom database
compsMF5 <- generateCompounds(fGroups, mslists, "metfrag", database = "csv", adduct =
↪  "[M+H]+",
                              extraOpts = list(LocalDatabasePath = "~/mydb.csv"))
```

An example of a custom *.csv* database can be found here.

With MetFrag compound databases are not only used to retrieve candidate structures but are also used to obtain metadata for further ranking. Each database has its own scorings, a table with currently supported scorings can be obtained with the `compoundScorings()` function (some columns omitted):

name	metfrag	database	default
score	Score		TRUE
fragScore	FragmenterScore		TRUE
metFusionScore	OfflineMetFusionScore		TRUE
individualMoNAScore	OfflineIndividualMoNAScore		FALSE
numberPatents	PubChemNumberPatents	pubchem	TRUE
numberPatents	Patent_Count	pubchemlite	TRUE
pubMedReferences	PubChemNumberPubMedReferences	pubchem	TRUE
pubMedReferences	ChemSpiderNumberPubMedReferences	chemspider	TRUE
pubMedReferences	NUMBER_OF_PUBMED_ARTICLES	comptox	TRUE
pubMedReferences	PubMed_Count	pubchemlite	TRUE
extReferenceCount	ChemSpiderNumberExternalReferences	chemspider	TRUE
dataSourceCount	ChemSpiderDataSourceCount	chemspider	TRUE
referenceCount	ChemSpiderReferenceCount	chemspider	TRUE
RSCCount	ChemSpiderRSCCount	chemspider	TRUE
formulaScore			FALSE
smartsInclusionScore	SmartsSubstructureInclusionScore		FALSE
smartsExclusionScore	SmartsSubstructureExclusionScore		FALSE
suspectListScore	SuspectListScore		FALSE
retentionTimeScore	RetentionTimeScore		FALSE
CPDATCount	CPDAT_COUNT	comptox	TRUE
TOXCASTActive	TOXCAST_PERCENT_ACTIVE	comptox	TRUE
dataSources	DATA_SOURCES	comptox	TRUE
pubChemDataSources	PUBCHEM_DATA_SOURCES	comptox	TRUE
EXPOCASTPredExpo	EXPOCAST_MEDIAN_EXPOSURE_PREDICTION_MG/KG-BW/DAY	comptox	TRUE
ECOTOX	ECOTOX	comptox	TRUE
NORMANSUSDAT	NORMANSUSDAT	comptox	TRUE
MASSBANKEU	MASSBANKEU	comptox	TRUE
TOX21SL	TOX21SL	comptox	TRUE
TOXCAST	TOXCAST	comptox	TRUE
KEMIMARKET	KEMIMARKET	comptox	TRUE
MZCLOUD	MZCLOUD	comptox	TRUE
pubMedNeuro	PubMedNeuro	comptox	TRUE
CIGARETTES	CIGARETTES	comptox	TRUE
INDOORCT16	INDOORCT16	comptox	TRUE
SRM2585DUST	SRM2585DUST	comptox	TRUE
SLTCHEMDB	SLTCHEMDB	comptox	TRUE
THSMOKE	THSMOKE	comptox	TRUE
ITNANTIBIOTIC	ITNANTIBIOTIC	comptox	TRUE
STOFFIDENT	STOFFIDENT	comptox	TRUE
KEMIMARKET_EXPO	KEMIMARKET_EXPO	comptox	TRUE
KEMIMARKET_HAZ	KEMIMARKET_HAZ	comptox	TRUE
REACH2017	REACH2017	comptox	TRUE
KEMIWW_WDUIndex	KEMIWW_WDUIndex	comptox	TRUE
KEMIWW_StpSE	KEMIWW_StpSE	comptox	TRUE
KEMIWW_SEHitsOverDL	KEMIWW_SEHitsOverDL	comptox	TRUE
ZINC15PHARMA	ZINC15PHARMA	comptox	TRUE
PFASMASTER	PFASMASTER	comptox	TRUE
peakFingerprintScore	AutomatedPeakFingerprintAnnotationScore		FALSE
lossFingerprintScore	AutomatedLossFingerprintAnnotationScore		FALSE

(continued)

name	metfrag	database	default
agroChemInfo	AgroChemInfo	pubchemlite	FALSE
bioPathway	BioPathway	pubchemlite	FALSE
drugMedicInfo	DrugMedicInfo	pubchemlite	FALSE
foodRelated	FoodRelated	pubchemlite	FALSE
pharmacoInfo	PharmacoInfo	pubchemlite	FALSE
safetyInfo	SafetyInfo	pubchemlite	FALSE
toxicityInfo	ToxicityInfo	pubchemlite	FALSE
knownUse	KnownUse	pubchemlite	FALSE
disorderDisease	DisorderDisease	pubchemlite	FALSE
identification	Identification	pubchemlite	FALSE
annoTypeCount	FPSum	pubchemlite	TRUE
annoTypeCount	AnnoTypeCount	pubchemlite	TRUE

The first two columns contain the generic and original MetFrag naming schemes for each scoring type. While both naming schemes can be used, the generic is often shorter and harmonized with other algorithms (e.g. SIRIUS). The *database* column specifies for which databases a particular scoring is available (empty if not database specific). Most scorings are selected by default (as specified by the *default* column), however, this behaviour can be customized by using the `scoreTypes` argument:

```
# Only in-silico and PubChem number of patents scorings
compsMF1 <- generateCompounds(fGroups, mslists, "metfrag", adduct = "[M+H]+",
                              scoreTypes = c("fragScore" "numberPatents"))

# Custom scoring in custom database
compsMF2 <- generateCompounds(fGroups, mslists, "metfrag", adduct = "[M+H]+",
                              database = "csv", adduct = "[M+H]+",
                              extraOpts = list(LocalDatabasePath = "~/mydb.csv"),
                              scoreTypes = c("fragScore", "myScore", "myScore2"))
```

By default ranking is performed with equal weight (i.e. *1*) for all scorings. This can be changed by the `scoreWeights` argument, which should be a `vector` containing the weights for all scorings following the order of `scoreTypes`, for instance:

```
compsMF <- generateCompounds(fGroups, mslists, "metfrag", adduct = "[M+H]+",
                             scoreTypes = c("fragScore" "numberPatents"),
                             scoreWeights = c(1, 2))
```

Sometimes thousands or more structural candidates are found when annotating a feature group. In this situation processing all these candidates will too involving (especially when external databases are used). To avoid this a default cut-off is set: when the number of candidates exceed a certain amount the search will be aborted and no results will be reported for that feature group. The maximum number of candidates can be set with the `maxCandidatesToStop` argument. The default value is relative conservative, especially for local databases it may be useful to increase this number.

4.3.3.2 Timeout and error handling

The use of online databases has the drawback that an error may occur, for instance, as a result of a connection error. Furthermore, MetFrag typically returns an error when too many candidates are found (as set by the `maxCandidatesToStop` argument). By default processing is restarted if an error has occurred (configured by the `errorRetries` argument). Similarly, the `timeoutRetries` and `timeout` arguments can be used to avoid being 'stuck' on obtaining results, for instance, due to an unstable internet connection.

If no compounds could be assigned due to an error a warning will be issued. In this case it is best to see what went wrong by manually checking the log files, which by default are stored in the *log/metfrag* folder.

4.3.3.3 Formula scoring

Ranking of candidate structures may further be improved by incorporating formula information by using the `addFormulaScoring()` function:

```
comps <- addFormulaScoring(coms, formulas, updateScore = TRUE)
```

Here, corresponding formula and explained fragments will be used to calculate a *formulaScore* for each candidate. Note that SIRIUS candidates are already based on calculated formulae, hence, running this function on SIRIUS results is less sensable unless scoring from another formula calculation algorithm is desired.

4.3.3.4 Further options and parameters

There are *many* more options and parameters that affect compound annotation. For a full overview please have a look at the reference manual (e.g. by running `?generateCompounds`).

4.4 Componentization

In `patRoon` *componentization* refers to grouping related feature groups together in components. Currently there are three different methodologies to generate components:

- Similarity on chromatographic elution profiles: feature groups with similar chromatographic behaviour which are assuming to be the same chemical compound (e.g. adducts or isotopologues).
- Homologous series: features with increasing m/z and retention time.
- Intensity profiles: features that follow a similar intensity profile in the analyses.

The following algorithms are currently supported:

Algorithm	Usage	Remarks
CAMERA	`generateComponents(algorithm = "camera", ...)`	Clusters feature groups with similar chromatographic elution profiles and annotate by known chemical rules (adducts, isotopologues, in-source fragments).
RAMClustR	`generateComponents(algorithm = "ramclustr", ...)`	As above.
nontarget	`generateComponents(algorithm = "nontarget", ...)`	Uses the nontarget R package to perform unsupervised homologous series detection.
Intensity clustering	`generateComponents(algorithm = "intclust", ...)`	Groups features with similar intensity profiles across analyses by hierarchical clustering.

NOTE Componentization is a complex process and currently still in a relative young development phase. As such, its functionality and interface are planned to be further improved and results obtained now should always be manually checked, for instance, by using the reporting functions.

4.4.1 Features with similar chromatographic behaviour

Isotopes, adducts and in-source fragments typically result in detection of multiple mass peaks by the mass spectrometer for a single chemical compound. While some feature finding algorithms already try to collapse (some of) these in to a single feature, this process is often incomplete (if performed at all) and it is not uncommon that multiple features will describe the same compound. To overcome this complexity the algorithms from CAMERA and RAMClustR can be used to group features that undergo highly similar chromatographic behaviour but have different m/z values. Basic chemical rules are then applied to the resulting components to annotate adducts, in-source fragments and isotopologues, which may be highly useful for general identification purposes.

Some common function arguments to `generateComponents()` are listed below. Note that careful tuning for some of these is required to obtain useful results. In our experience the current default settings may significantly 'over-cluster' features that (clearly visibly) do not belong to each other. For this reason, you are advised to optimize and verify the various parameters supported by both algorithms. For a complete listing all arguments see the reference manual (e.g. `?generateComponents`).

Argument	Algorithm	Remarks
`ionization`	`"camera"`, `"ramclustr"`	Ionization mode: `"positive"` or `"negative"`
`minSize`	`"camera"`, `"ramclustr"`	Minimum component size. Smaller components will be removed.
`relMinReplicates`	`"camera"`, `"ramclustr"`	See below.
`st`, `sr`, `maxt`, `hmax`	`"ramclustr"`	Common parameters to influence clustering of RAMClustR. See `?ramclustR` for details.
`extraOpts`	`"camera"`	A list with extra argument passed to the `annotate()` function from CAMERA.
`extraOptsRC`, `extraOptsFM`	`"ramclustr"`	A list with extra arguments passed to the `ramclustR()` and `do.findmain()` functions from RAMClustR.

Note that both algorithms were primarily designed for datasets where features are generally present in the majority of the analyses (as is relatively common in metabolomics). For environmental analyses, however, this is often not the case. As a result, it may happen that not all features from the feature groups within a component share their presence in the same analyses. In reality, this situation would be fairly unusual, and it is likely that such features actually do not belong to the same component. An extra filter option was added to improve such scenarios: after componentization all features are checked to have a minimal presence across all analyses within the component. This is configured by the `relMinReplicates` argument of `generateComponents()`, which specifies the relative number of replicate groups in which a feature should be present. For instance, when this value is *0.5* (the default), a feature must be present in at least half of all replicate groups present in the component.

Some example usage is shown below.

```
# Use CAMERA with defaults
componCAM <- generateComponents(fGroups, "camera", ionization = "positive")

# CAMERA with customized settings
componCAM2 <- generateComponents(fGroups, "camera", ionization = "positive",
                                 extraOpts = list(mzabs = 0.001, sigma = 5))

# Use RAMClustR with customized parameters
```

```
componRC <- generateComponents(fGroups, "ramclustr", ionization = "positive", hmax = 0.4,
                               extraOptsRC = list(cor.method = "spearman"),
                               extraOptsFM = list(ppm.error = 5))
```

4.4.2 Homologues series

Homologues series can be automatically detected by interfacing with the nontarget R package. Components are made from feature groups that show increasing m/z and retention time values. Series are first detected within each replicate group. Afterwards, series from all replicates are linked in case (partial) overlap occurs and this overlap consists of the *same* feature groups (see figure below). Linked series are then finally merged if this will not cause any conflicts with other series: such a conflict typically occurs when two series are not only linked to each other.

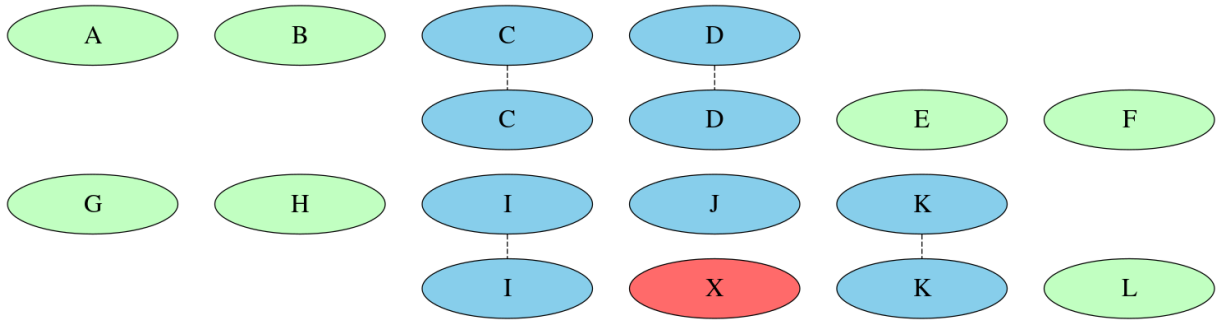


Figure 1: **Linking of homologues series** top: partial overlap and will be linked; bottom: no linkage due to different feature in overlapping series.

The series that are linked can be interactively explored with the `plotGraph()` function (discussed here).

Common function arguments to `generateComponents()` are listed below.

Argument	Remarks
`ionization`	Ionization mode: `"positive"` or `"negative"`
`rtRange`, `mzRange`	Retention and m/z increment range. Retention times can be negative to allow series with increasing m/z values and decreasing retention times.
`elements`	Vector with elements to consider.
`rtDev`, `absMzDev`	Maximum retention time and m/z deviation.
`extraOpts`	List with extra arguments passed to the `homol.search()` function.

```
# default settings
componNT <- generateComponents(fGroups, "nontarget", ionization = "positive")

# customized settings
componNT2 <- generateComponents(fGroups, "nontarget", ionization = "positive",
                                elements = c("C", "H"), rtRange = c(-60, 60))
```

4.4.3 Intensity clustering

Whereas previous componentization methods utilized chemical properties to relate features, intensity clustering uses a statistical approach. This methodology is especially useful to find features that show similar trends in the analysed samples. Intensities for all features are first normalized and thereafter hierarchical clustering is performed to find features that show similar intensity profiles across analyses. Components are then formed from automatically assigned clusters (using the dynamicTreeCut R package, however, assignment can be changed afterwards).

Some common arguments to `generateComponents()` are listed below. It is recommended to test various settings (especially for `method`) to optimize the clustering results.

Argument	Default	Remarks
`method`	`"complete"`	Clustering method. See `?hclust`
`metric`	`"euclidean"`	Metric used to calculate the distance matrix. See `?daisy`.
`normFunc`	`max`	Function used to normalize data. Feature intensities within a feature group are divided by the result of when this function is called with their intensity values.
`average`	`TRUE`	Whether intensities of replicates should first be averaged.
`maxTreeHeight`, `deepSplit`, `minModuleSize`	`1`, `TRUE`, `1`	Used for dynamic cluster assignment. See `?cutreeDynamicTree`.

The resulting components are stored in an object from the `componentsIntClust` S4 class. Several methods are defined that can be used on such objects to re-assign clusters, perform plotting operations and so on. Below are some examples. For plotting see the relevant visualization section. More info can be found in the reference manual (e.g. `?componentsIntClust`).

```
# generate components with default settings
componInt <- generateComponents(fGroups, "intclust")

# manually re-assign clusters
componInt <- treeCut(componInt, k = 10)

# automatic re-assignment of clusters (adjusted max tree height)
componInt <- treeCutDynamic(componInt, maxTreeHeight = 0.7)
```

5 Processing workflow data

The previous chapter mainly discussed how to create workflow data. This chapter will discuss how to *use* the data.

5.1 Inspecting results

Several generic functions exist that can be used to inspect data that is stored in a particular object (e.g. features, compounds etc):

Generic	Classes	Remarks
`length()`	All	Returns the length of the object (e.g. number of features, compounds etc)
`algorithm()`	All	Returns the name of the algorithm used to generate the object.
`groupNames()`	All	Returns all the unique identitifiers (or names) of the feature groups for which this object contains results.
`names()`	`featureGroups`, `components`	Returns names of the feature groups (similar to `groupNames()`) or components
`show()`	All	Prints general information.
`"[["` / `"$"` operators	All	Extract general information, see below.
`as.data.table()` / `as.data.frame()`	All	Convert data to a `data.table` or `data.frame`, see below.
`analysisInfo()`, `analyses()`, `replicateGroups()`	`features`, `featureGroups`	Returns the analysis information, analyses or replicate groups for which this object contains data.
`groupInfo()`	`featureGroups`	Returns feature group information (m/z and retention time values).
`componentInfo()`	`components`	Returns information for all components.
`annotatedPeakList()`	`formulas`, `compounds`	Returns a table with annotated mass peaks (see below).

The common `R` extraction operators `"[["`, `"$"` can be used to obtain data for a particular feature groups, analysis etc:

```
# Feature table (only first columns for readability)
fList[["standard-1"]][, 1:6]
```

```
#>                          ID       ret        mz       area    retmin    retmax
#>   1: f_1325871792469526966  19.00698  78.99684  65793.391  13.01202  29.99700
#>   2:  f_9981922862380190160  20.00598  79.02098 125354.100  13.01202  29.99700
#>   3:  f_9389205064611426963   8.01600  79.02101 188045.900   3.03000  12.01302
#>   4: f_1711658517027003487 2 112.34220  79.02103  71168.984 106.55400 147.51600
#>   5:  f_7616893087408325613 111.64380  84.95950  54875.129 105.15180 599.87400
#>  ---
#> 433: f_1780674719801578262 5  11.01402 295.22387   9514.476   9.01602  13.01202
#> 434:  f_2722754295659840987  13.01202 296.92038  37753.414  10.01502  15.01098
#> 435: f_1366215907591431984 9  12.01302 297.16751  17366.125   9.01602  15.01098
#> 436:   f_783800915643457865  11.01402 297.20368  13995.990   9.01602  14.01102
#> 437:  f_4595933616757439994  12.01302 299.12581  13630.378   9.01602  15.01098
```

```
# Feature group intensities
fGroups$M120_R328_81
```

```
#> [1] 55936 61668 59624
```

```
fGroups[[1, "M120_R328_81"]] # only first analysis
```

```
#> [1] 55936
```

```
# obtains list MS/MS peak list  (feature group averaged datas)
mslists[["M120_R328_81"]]$MSMS
```

```
#>           mz intensity precursor
#> 1:  92.04936  9426.556     FALSE
#> 2: 118.08606  2167.444     FALSE
#> 3: 120.05549 43949.222      TRUE
#> 4: 121.05850  3397.556     FALSE
```

```
# get all formula candidates for a feature group
formulas[["M120_R328_81"]][, 1:7]
```

```
#>      analysis neutral_formula formula formula_mz    error dbe  isoScore
#> 1: standard-1          C6H5N3  C6H6N3   120.0556 1.266667   6 0.9905167
#> 2: standard-1          C6H5N3  C6H6N3   120.0556 1.266667   6 0.9905167
```

```
# get all compound candidates for a feature group
compounds[["M120_R328_81"]][, 1:4]
```

```
#>      explainedPeaks    score neutralMass                     SMILES
#>   1:              1 3.322371    119.0483          C1=CC2=NNN=C2C=C1
#>   2:              1 1.915812    119.0483   C1=CC=C(C=C1)N=[N+]=[N-]
#>   3:              1 1.644548    119.0483        C1=CC=C2C(=C1)N2N=N
#>   4:              1 1.522632    119.0483 C1=CC(=N)C(=[N+]=[N-])C=C1
#>   5:              1 1.490700    119.0483   C1=CC=C2C(=C1)[N-]N[N-]2
#>  ---
#>  96:              1 1.056517    119.0483   C=C1CC(=C=C1)N=[N+]=[N-]
#>  97:              1 1.055418    119.0483          C1=CN=CN2C1=NC=C2
#>  98:              1 1.054233    119.0483   C=C1C=CC(=C1)N=[N+]=[N-]
#>  99:              1 1.053306    119.0483    C#CC1(CC=C1)N=[N+]=[N-]
#> 100:              1 1.053152    119.0487     C(C[PH2+]N=[N+]=[N-])N
```

```
# get a table with information of a component
components[["CMP7"]][, 1:6]
```

```
#>            group       rt       mz isotopes adnr adduct_rule
#> 1: M254_R321_484 320.2038 254.0596         1           1
#> 2: M276_R321_550 320.6412 276.0415         1           6
```

A more sophisticated way to obtain data from a workflow object is to use `as.data.table()` or `as.data.frame()`. These functions will convert *all* information within the object to a table (`data.table` or `data.frame`) and allow various options to add extra information. An advantage is that this common data format can be used with many other functions within `R`. The output is in a tidy format.

NOTE If you are not familiar with `data.table` and want to know more see data.table. Briefly, this is a more efficient and largely compatible alternative to the regular `data.frame`.

NOTE The `as.data.frame()` methods defined in `patRoon` simply convert the results from `as.data.table()`, hence, both functions are equal in their usage and are defined for the same object classes.

Some typical examples are shown below.

```
# obtain table with all features (only first columns for readability)
as.data.table(fList)[, 1:6]
```

```
#>         analysis                      ID       ret        mz      area     retmin
#>    1:  solvent-1  f_5078967719092585026  19.10298  78.99678  73929.27  13.108980
#>    2:  solvent-1 f_17989775753929033662  14.10798  84.95948 941557.00   3.922998
#>    3:  solvent-1  f_5189479535116000426 112.97220  84.95949  47558.80 105.301800
#>    4:  solvent-1   f_547181819488669422  13.10898  88.95255  34552.67   7.114020
#>    5:  solvent-1  f_5549475336976468432   2.52900  88.96823  14710.10   1.732002
#>   ---
#> 2440: standard-3  f_8730621282619171553  12.14898 294.93888 350965.20   9.151020
#> 2441: standard-3   f_990042466826947408  11.14998 295.18863  10705.59   9.151020
#> 2442: standard-3  f_2478487158895637249  13.14798 296.92039  42195.86  10.150980
#> 2443: standard-3  f_9461375865393630022  11.14998 297.16737  15177.58   9.151020
#> 2444: standard-3 f_11158263824054210883  11.14998 297.20377  16678.13   9.151020
```

```
# Returns group info and intensity values for each feature group
as.data.table(fGroups, average = TRUE) # average intensities for replicates
```

```
#>             group      ret       mz standard
#>  1:  M120_R328_81 327.9955 120.0555 59076.00
#>  2: M134_R399_118 398.6365 134.0712 78472.00
#>  3: M135_R261_123 260.5933 135.1014 16325.33
#>  4: M137_R303_127 302.7823 137.0708 43810.67
#>  5: M146_R185_153 185.2398 146.0599 23716.00
#> ---
#> 23: M237_R510_445 510.3882 237.1026 60214.67
#> 24: M242_R461_459 460.6473 242.2848 78576.00
#> 25: M254_R321_484 320.7174 254.0598 33440.00
#> 26: M276_R321_550 320.9508 276.0416 10876.00
#> 27: M279_R284_557 283.9626 279.0915 42941.33
```

```
# Returns all peak lists for each feature group
as.data.table(mslists)
```

```
#>              group type        mz intensity precursor
#>   1:  M120_R328_81   MS  84.95945  2526.560     FALSE
#>   2:  M120_R328_81   MS  87.08025  1148.153     FALSE
#>   3:  M120_R328_81   MS  88.96826  3891.785     FALSE
#>   4:  M120_R328_81   MS  93.00016  1235.728     FALSE
#>   5:  M120_R328_81   MS  97.96859  1412.052     FALSE
```

```
#>  ---
#> 223: M146_R185_153 MSMS  91.05407  1621.285     FALSE
#> 224: M146_R185_153 MSMS 117.05709  1473.430     FALSE
#> 225: M146_R185_153 MSMS 118.06485  4570.679     FALSE
#> 226: M146_R185_153 MSMS 146.05993 30936.412      TRUE
#> 227: M146_R185_153 MSMS 147.06315  4029.333     FALSE
```

```
# Returns all formula candidates for each feature group with scoring
# information, neutral loss etc
as.data.table(formulas)[, 1:6]
```

```
#>             group   analysis neutral_formula formula formula_mz     error
#>  1:  M120_R328_81 standard-1          C6H5N3  C6H6N3   120.0556 1.2666667
#>  2:  M120_R328_81 standard-1          C6H5N3  C6H6N3   120.0556 1.2666667
#>  3: M134_R399_118 standard-1          C7H7N3  C7H8N3   134.0713 0.6666667
#>  4: M134_R399_118 standard-1          C7H7N3  C7H8N3   134.0713 0.6666667
#>  5: M134_R399_118 standard-1          C7H7N3  C7H8N3   134.0713 0.6666667
#> ---
#> 13: M137_R303_127 standard-1         C7H8N2O C7H9N2O   137.0709 1.3333333
#> 14: M146_R185_153 standard-1          C9H7NO  C9H8NO   146.0600 1.0666667
#> 15: M146_R185_153 standard-1          C9H7NO  C9H8NO   146.0600 1.0666667
#> 16: M146_R185_153 standard-1          C9H7NO  C9H8NO   146.0600 1.0666667
#> 17: M146_R185_153 standard-1          C9H7NO  C9H8NO   146.0600 1.0666667
```

```
# Returns all compound candidates for each feature group with scoring and other metadata
as.data.table(compounds)[, 1:4]
```

```
#>              group explainedPeaks    score neutralMass
#>   1:  M120_R328_81              1 3.322371    119.0483
#>   2:  M120_R328_81              1 1.915812    119.0483
#>   3:  M120_R328_81              1 1.644548    119.0483
#>   4:  M120_R328_81              1 1.522632    119.0483
#>   5:  M120_R328_81              1 1.490700    119.0483
#>  ---
#> 496: M146_R185_153              3 1.532748    145.0528
#> 497: M146_R185_153              3 1.531075    145.0528
#> 498: M146_R185_153              3 1.524293    145.0528
#> 499: M146_R185_153              3 1.524285    145.0528
#> 500: M146_R185_153              3 1.524285    145.0528
```

```
# Returns table with all components (including feature group info, annotations etc)
as.data.table(components)[, 1:6]
```

```
#>     name  cmp_ret cmp_retsd neutral_mass   analysis size
#>  1: CMP1 560.3138 0.3277624         <NA> standard-2    2
#>  2: CMP1 560.3138 0.3277624         <NA> standard-2    2
#>  3: CMP2 328.4513 0.6446699         <NA> standard-3    2
#>  4: CMP2 328.4513 0.6446699         <NA> standard-3    2
#>  5: CMP3 424.2700 0.0000000    182.07092 standard-1    2
#>  6: CMP3 424.2700 0.0000000    182.07092 standard-1    2
#>  7: CMP4 593.5129 0.3296681         <NA> standard-1    2
```

```
#>  8: CMP4 593.5129 0.3296681         <NA> standard-1    2
#>  9: CMP5 301.6178 1.0152257         <NA> standard-3    3
#> 10: CMP5 301.6178 1.0152257         <NA> standard-3    3
#> 11: CMP5 301.6178 1.0152257         <NA> standard-3    3
#> 12: CMP6 215.1077 0.4942076     107.0977 standard-3    2
#> 13: CMP6 215.1077 0.4942076     107.0977 standard-3    2
#> 14: CMP7 320.8341 0.1649877    253.05231 standard-2    2
#> 15: CMP7 320.8341 0.1649877    253.05231 standard-2    2
```

Finally, the `annotatedPeakList()` function is useful to inspect annotation results for a formula or compound candidate:

```
# formula annotations for for a formula candidate of feature group M120_R328_81
annotatedPeakList(formulas, precursor = "C6H6N3", groupName = "M120_R328_81",
                  MSPeakLists = mslists)
```

```
#>          mz intensity precursor formula neutral_loss
#> 1:  92.04936  9426.556     FALSE   C6H6N           N2
#> 2: 118.08606  2167.444     FALSE    <NA>         <NA>
#> 3: 120.05549 43949.222      TRUE  C6H6N3
#> 4: 121.05850  3397.556     FALSE    <NA>         <NA>
```

```
# compound annotation for first candidate of feature group M120_R328_81
annotatedPeakList(compounds, index = 1, groupName = "M120_R328_81",
                  MSPeakLists = mslists)
```

```
#>          mz intensity precursor formula neutral_loss score
#> 1:  92.04936  9426.556     FALSE   C6H6N           N2  1022
#> 2: 118.08606  2167.444     FALSE    <NA>         <NA>    NA
#> 3: 120.05549 43949.222      TRUE    <NA>         <NA>    NA
#> 4: 121.05850  3397.556     FALSE    <NA>         <NA>    NA
```

More advanced examples for these functions are shown below.

```
# Feature table, can also be accessed by numeric index
fList[[1]]
mslists[["standard-1", "M120_R328_81"]] # feature data (instead of feature group
↪  averaged)
formulas[[1, "M120_R328_81"]] # feature data (if available, i.e. calculateFeatures=TRUE)
components[["CMP1", 1]] # only for first feature group in component

as.data.frame(fList) # classic data.frame format, works for all objects
as.data.table(fGroups) # return non-averaged intensities (default)
as.data.table(fGroups, features = TRUE) # include feature information
as.data.table(mslists, averaged = FALSE) # peak lists each feature
as.data.table(mslists, fGroups = fGroups) # add feature group information

as.data.table(formulas, countElements = c("C", "H")) # include C/H counts (e.g. for van
↪  Krevelen plots)
# report only top precursor and fragment formula. This yields in one row per feature
↪  group.
as.data.table(formulas, maxFormulas = 1, maxFragFormulas = 1)
```

```
# add various information for organic matter characterization (common elemental
# counts/ratios, classifications etc)
as.data.table(formulas, OM = TRUE)

as.data.table(compounds, fGroups = fGroups) # add feature group informaion
as.data.table(compounds, fragments = TRUE) # include information of all annotated
↪  fragments

annotatedPeakList(formulas, precursor = "C6H6N3", groupName = "M120_R328_81",
                  MSPeakLists = mslists, onlyAnnotated = TRUE) # only include annotated
                  ↪  peaks
annotatedPeakList(compounds, index = 1, groupName = "M120_R328_81",
                  MSPeakLists = mslists, formulas = formulas) # include formula
                  ↪  annotations
```

5.2 Filtering

During a non-target workflow it is not uncommon that some kind of data-cleanup is necessary. Datasets are often highly complex, which makes separating data of interest from the rest highly important. Furthermore, general cleanup typically improves the quality of the dataset, for instance by removing low scoring annotation results or features that are unlikely to be 'correct' (e.g. noise or present in blanks). For this reason `patRoon` supports *many* different filters that easily clean data produced during the workflow in a highly customizable way.

All major workflow objects (e.g. `featureGroups`, `compounds`, `components` etc.) support filtering operations by the `filter()` generic. This function takes the object to be filtered as first argument and any remaining arguments describe the desired filter options. The `filter()` generic function then returns the modified object back. Some examples are shown below.

```
# remove low intensity (<500) features
features <- filter(features, absMinIntensity = 500)

# remove features with intensities lower than 5 times the blank
fGroups <- filter(fGroups, blankThreshold = 5)

# only retain compounds with >1 explained MS/MS peaks
compounds <- filter(compounds, minExplainedPeaks = 1)
```

The following sections will provide a more detailed overview of available data filters.

> ***NOTE*** Some other `R` packages (notably `dplyr`) also provide a `filter()` generic function. To use the `filter()` function from different packages you need explicitly specify which one to use in your script. This can be done by prefixing it with the package name, e.g. `patRoon::filter(...)`, `dplyr::filter(...)` etc.

5.2.1 Features

There are many filters available for feature data:

Filter	Classes	Remarks
`absMinIntensity`, `relMinIntensity`	`features`, `featureGroups`	Minimum intensity
`preAbsMinIntensity`, `preRelMinIntensity`	`featureGroups`	Minimum intensity prior to other filtering (see below)
`retentionRange`, `mzRange`, `mzDefectRange`, `chromWidthRange`	`features`, `featureGroups`	Filter by feature properties
`absMinAnalyses`, `relMinAnalyses`	`featureGroups`	Minimum feature abundance in all analyses
`absMinReplicates`, `relMinReplicates`	`featureGroups`	Minimum feature abundance in different replicates
`absMinFeatures`, `relMinFeatures`	`featureGroups`	Only keep analyses with at least this amount of features
`absMinReplicateAbundance`, `relMinReplicateAbundance`	`featureGroups`	Minimum feature abundance in a replicate group
`maxReplicateIntRSD`	`featureGroups`	Maximum relative standard deviation of feature intensities in a replicate group.
`blankThreshold`	`featureGroups`	Minimum intensity factor above blank intensity
`rGroups`	`featureGroups`	Only keep (features of) these replicate groups

Application of filters to feature data is important for (environmental) non-target analysis. Especially blank and replicate filters (i.e. `blankThreshold` and `absMinReplicateAbundance`/`relMinReplicateAbundance`) are important filters and are highly recommended to always apply for cleaning up your dataset.

All filters are available for feature group data, whereas only a subset is available for feature objects. The main reason is that other filters need grouping of features between analyses. Regardless, in `patRoon` filtering feature data is less important, and typically only needed when the number of features are extremely large and direct grouping is undesired.

From the table above you can notice that many filters concern both *absolute* and *relative* data (i.e. as prefixed with `abs` and `rel`). When a relative filter is used the value is scaled between *0* and *1*. For instance:

```
# remove features not present in at least half of the analyses within a replicate group
fGroups <- filter(fGroups, relMinReplicateAbundance = 0.5)
```

An advantage of relative filters is that you will not have to worry about the data size involved. For instance, in the above example the filter always takes half of the number of analyses within a replicate group, even when replicate groups have different number of analyses.

Note that multiple filters can be specified at once. Especially for feature group data the order of filtering may impact the final results, this is explained further in the reference manual (i.e. `` ?`feature-filtering` ``).

Some examples are shown below.

```
# filter features prior to grouping: remove any features eluting before first 2 minutes
fList <- filter(fList, retentionRange = c(120, Inf))

# common filters for feature groups
fGroups <- filter(fGroups,
                  absMinIntensity = 500, # remove features <500 intensity
                  relMinReplicateAbundance = 1, # features should be in all analysis of
                  ↪  replicate groups
```

```
                  maxReplicateIntRSD = 0.75, # remove features with intensity RSD in
                  ↪  replicates >75%
                  blankThreshold = 5, # remove features <5x intensity of (average) blank
                  ↪  intensity
                  removeBlanks = TRUE) # remove blank analyses from object afterwards

# filter by feature properties
fGroups <- filter(mzDefectRange = c(0.8, 0.9),
                  chromWidthRange = c(6, 120))

# remove features not present in at least 3 analyses
fGroups <- filter(fGroups, absMinAnalyses = 3)

# remove features not present in at least 20% of all replicate groups
fGroups <- filter(fGroups, relMinReplicates = 0.2)

# only keep data present in replicate groups "repl1" and "repl2"
# all other features and analyses will be removed
fGroups <- filter(fGroups, rGroups = c("repl1", "repl2"))
```

5.2.2 Annotation

There are various filters available for handling annotation data:

Filter	Classes	Remarks
`absMSIntThr`, `absMSMSIntThr`, `relMSIntThr`, `relMSMSIntThr`	`MSPeakLists`	Minimum intensity of mass peaks
`topMSPeaks`, `topMSMSPeaks`	`MSPeakLists`	Only keep most intense mass peaks
`withMSMS`	`MSPeakLists`	Only keep results with MS/MS data
`minExplainedPeaks`	`formulas` / `compounds`	Minimum number of annotated mass peaks
`elements`, `fragElements`, `lossElements`	`formulas`, `compounds`	Restrain elemental composition
`topMost`	`formulas`, `compounds`	Only keep highest ranked candidates
`minScore`, `minFragScore`, `minFormulaScore`	`compounds`	Minimum compound scorings
`scoreLimits`	`formulas`, `compounds`	Minimum/Maximum scorings
`OM`	`formulas`	Only keep candidates with likely elemental composition found in organic matter

Several intensity related filters are available to clean-up MS peak list data. For instance, the `topMSPeaks`/`topMSMSPeaks` filters provide a simple way to remove noisy data by only retaining a defined number of most intense mass peaks. Note that none of these filters will remove the mass peak of the feature from its MS peak list.

The filters applicable to formula and compound annotation generally concern minimal scoring or chemical properties. The former is useful to remove unlikely candidates, whereas the second is useful to focus on certain study specific chemical properties (e.g. known neutral losses).

Common examples are shown below.

```
# intensity filtering
mslists <- filter(mslists,
                  absMSIntThr = 500, # minimum MS mass peak intensity of 500
                  relMSMSIntThr = 0.1) # minimum MS/MS mass peak intensity of 10%

# only retain 10 most intens mass peaks
# (feature mass is always retained)
mslists <- filter(mslists, topMSPeaks = 10)

# only keep formulae with 1-10 sulphur or phosphorus elements
formulas <- filter(formulas, elements = c("S1-10", "P1-10"))

# only keep candidates with MS/MS fragments that contain 1-10 carbons and 0-2 oxygens
formulas <- filter(formulas, fragElements = "C1-10O0-2")

# only keep candidates with CO2 neutral loss
formulas <- filter(formulas, lossElements = "CO2")

# only keep the 15 highest ranked candidates with at least 1 annotated MS/MS peak
compounds <- filter(compounds, minExplainedPeaks = 1, topMost = 15)

# minimum in-silico score
compounds <- filter(compounds, minFragScore = 10)

# candidate should be referenced in at least 1 patent
# (only works if database lists number of patents, e.g. PubChem)
compounds <- filter(compounds,
                    scoreLimits = list(numberPatents = c(1, Inf))
```

5.2.3 Components

Finally several filters are available for components:

Filter	Remarks
`size`	Minimum component size
`adducts`, `isotopes`	Filter features by adduct/istopes annotation
`rtIncrement`, `mzIncrement`	Filter homologs by retention/mz increment range

Note that these filters are only applied if the components contain the data the filter works on. For instance, filtering by adducts will *not* affect components obtained from homologous series.

As before, some typical examples are shown below.

```
# only keep components with at least 4 features
componInt <- filter(componInt, minSize = 4)

# remove all features from components are not annotated as an adduct
componRC <- filter(componRC, adducts = TRUE)

# only keep protonated and sodium adducts
componRC <- filter(componRC, adducts = c("[M+H]+", "[M+Na]+"))
```

```r
# remove all features not recognized as isotopes
componRC <- filter(componRC, isotopes = FALSE)

# only keep monoisotopic mass
componRC <- filter(componRC, isotopes = 0)

# min/max rt/mz increments for homologs
componNT <- filter(componNT, rtIncrement = c(10, 30),
                   mzIncrement = c(16, 50))
```

> ***NOTE*** As mentioned before, components are still in a relative young development phase and results should always be verified!

5.2.4 Negation

All filters support *negation*: if enabled all specified filters will be executed in an opposite manner. Negation may not be so commonly used, but allows greater flexibility which is sometimes needed for advanced filtering steps. Furthermore, it is also useful to specifically isolate the data that otherwise would have been removed. Some examples are shown below.

```r
# keep all features/analyses _not_ present from replicate groups "repl1" and "repl2"
fGroups <- filter(fGroups, rGroups = c("repl1", "repl2"), negate = TRUE)

# only retain features with a mass defect outside 0.8-0.9
fGroups <- filter(mzDefectRange = c(0.8, 0.9), negate = TRUE)

# remove candidates with CO2 neutral loss
formulas <- filter(formulas, lossElements = "CO2", negate = TRUE)

# select 15 worst ranked candidates
compounds <- filter(compounds, topMost = 15, negate = TRUE)

# only keep components with <5 features
componInt <- filter(componInt, minSize = 5, negate = TRUE)
```

5.3 Subsetting

The previous section discussed the `filter()` generic function to perform various data cleaning operations. A more generic way to select data is by *subsetting*: here you can manually specify which parts of an object should be retained. Subsetting is supported for all workflow objects and is performed by the R subset operator (`"["`). This operator either subsets by one or two arguments, which are referred to as the `i` and `j` arguments.

Class	Argument `i`	Argument `j`	Remarks
`features`	analyses		
`featureGroups`	analyses	feature groups	
`MSPeakLists`	analyses	feature groups	peak lists for feature groups will be re-averaged when subset on analyses (by default)
`formulas`	feature groups		

Class	Argument `i`	Argument `j`	Remarks
`compounds`	feature groups		
`components`	components	feature groups	

For objects that support two-dimensional subsetting (e.g. `featureGroups`, `MSPeakLists`), either the `i` or `j` argument is optional. Furthermore, unlike subsetting a `data.frame`, the position of `i` and `j` does not change when only one argument is specified:

```
df[1, 1] # subset data.frame by first row/column
df[1] # subset by first column
df[1, ] # subset by first row

fGroups[1, 1] # subset by first analysis/feature group
fGroups[, 1] # subset by first feature group (i.e. column)
fGroups[1] # subset by first analysis (i.e. row)
```

The subset operator allows three types of input:

- A logical vector: elements are selected if corresponding values are `TRUE`.
- A numeric vector: select elements by numeric index.
- A character vector: select elements by their name.

When a logical vector is used as input it will be re-cycled if necessary. For instance, the following will select by the first, third, fifth, etc. analysis.

```
fGroups[c(TRUE, FALSE)]
```

In order to select by a `character` you will need to know the names for each element. These can, for instance, be obtained by the `groupNames()` (feature group names), `analyses()` (analysis names) and `names()` (names for components or feature groups for `featureGroups` objects) generic functions.

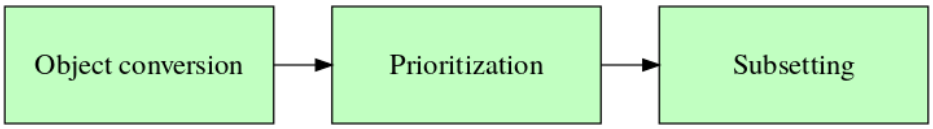
Some more examples of common subsetting operations are shown below.

```
# select first three analyses
fList[1:3]

# select first three analyses and first 500 feature groups
fGroups[1:3, 1:500]

# select all feature groups from first component
fGroupsNT <- fGroups[, componNT[[1]]$group]

# only keep feature groups with formula annotation results
fGroupsForms <- fGroups[, groupNames(formulas)]

# only keep feature groups with either formula or compound annotation results
fGroupsAnn <- fGroups[, union(groupNames(formulas), groupNames(compounds))]

# select first 15 components
components[1:15]
```

```
# select by name
components[c("CMP1", "CMP5")]

# only retain feature groups in components for which compound annotations are
# available
components[, groupNames(compounds)]
```

5.3.1 Prioritization workflow

An important use case of subsetting is prioritization of data. For instance, after statistical analysis only certain feature groups are deemed relevant for the rest of the workflow. A common prioritization workflow is illustrated below:



During the first step the workflow object is converted to a suitable format, most often using the `as.data.frame()` function. The converted data is then used as input for the prioritization strategy. Finally, these results are then used to select the data of interest in the original object.

A very simplified example of such a process is shown below.

```
featTab <- as.data.frame(fGroups, average = TRUE)

# prioritization: sort by (averaged) intensity of the "sample" replicate group
# (from high to low) and then obtain the feature group identifiers of the top 5.
featTab <- featTab[order(featTab$standard, decreasing = TRUE), ]
groupsOfInterest <- featTab$group[1:5]

# subset the original data
fGroups <- fGroups[, groupsOfInterest]

# fGroups now only contains the feature groups for which intensity values in the
# "sample" replicate group were in the top 5
```

5.4 Unique and overlapping features

Often an analysis batch is composed of different sample groups, such as different treatments, influent/effluent etc. In such scenarios it may be highly interesting to evaluate uniqueness or overlap between these samples. Furthermore, extracting overlapping or unique features is a simple but effective prioritization strategy.

The `overlap()` and `unique()` functions can be used to extract overlapping and unique features between replicate groups, respectively. Both functions return a subset of the given `featureGroups` object. An overview of their arguments is given below.

Argument	Function(s)	Remarks
`which`	`unique()`, `overlap()`	The replicate groups to compare.
`relativeTo`	`unique()`	Only return unique features compared to these replicate groups (`NULL` for all). Replicate groups in `which` are ignored.
`outer`	`unique()`	If `TRUE` then only return features which are *also* unique among the compared replicates groups.

Argument	Function(s)	Remarks
`exclusive`	`overlap`	Only keep features that *only* overlap between the compared replicate groups.

Some examples:

```
# only keep features uniquely present in replicate group "repl1"
fGroupsUn1 <- unique(fGroups, which = "repl1")
# only keep features in repl1/repl2 which are not in repl3
fGroupsUn2 <- unique(fGroups, which = c("repl1", "repl2"),
                     relativeTo = "repl3")
# only keep features that are only present in repl1 OR repl2
fGroupsUn3 <- unique(fGroups, which = c("repl1", "repl2"),
                     outer = TRUE)

# only keep features overlapping in repl1/repl2
fGroupsOv1 <- overlap(fGroups, which = c("repl1", "repl2"))
# only keep features overlapping in repl1/repl2 AND are not present in any other
# replicate group
fGroupsOv2 <- overlap(fGroups, which = c("repl1", "repl2"),
                      exclusive = TRUE)
```

In addition, several plotting functions are discussed in the next section that visualize overlap and uniqueness of features.

5.5 Visualization

5.5.1 Features and annatation data

Several generic functions are available to visualize feature and annotation data:

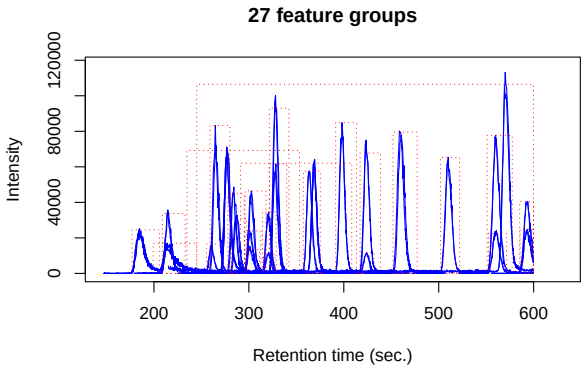
Generic	Classes	Remarks
`plot()`	`featureGroups`, `featureGroupsComparison`	Scatter plot for retention and m/z values
`plotInt()`	`featureGroups`	Intensity profiles across analyses
`plotChroms()`	`featureGroups`, `components`	Plot extracted ion chromatograms (EICs)
`plotSpectrum()`	`MSPeakLists`, `formulas`, `compounds`, `components`	Plots (annotated) spectra
`plotStructures()`	`compounds`	Draws candidate structures
`plotScores()`	`formulas`, `compounds`	Barplot for candidate scoring
`plotGraph()`	`componentsNT`	Draws interactive graphs of linked homologous series

The most common plotting functions are `plotChroms()`, which plots chromatographic data for features, and `plotSpectrum()`, which will plot (annotated) spectra. An overview of their most important function arguments are shown below.

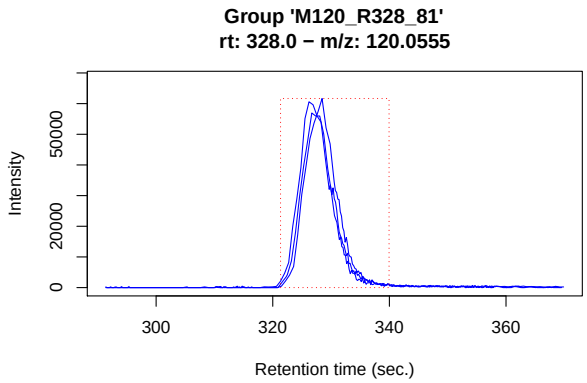
Argument	Generic	Remarks
`rtWindow`	`plotChroms()`	Extra time (in s) +/- retention limits of plotted features (useful to zoom out)
`mzWindow`	`plotChroms()`	m/z width of EICs (in Da)
`retMin`	`plotChroms()`	If `TRUE` plot retention times in minutes
`topMost`	`plotChroms()`	Only draw this amount of highest intensity features in each group.
`showPeakArea`, `showFGroupRect`	`plotChroms()`	Fill peak areas / draw rectangles around feature groups?
`title`	`plotChroms()`, `plotSpectrum()`	Override plot title
`colourBy`	`plotChroms()`	Colour individual feature groups (`"fGroups"`) or replicate groups (`"rGroups"`). By default nothing is coloured (`"none"`)
`showLegend`	`plotChroms()`	Display a legend? (only if `colourBy!="none"`)
`onlyPresent`	`plotChroms()`	Only plot EICs for analyses where a feature was detected? Setting to `FALSE` is useful to inspect if a feature was 'missed'.
`xlim`, `ylim`	`plotChroms()`, `plotSpectrum()`	Override x/y axis ranges, i.e. to manually set plotting range.
`groupName`, `analysis`, `precursor`, `index`	`plotSpectrum()`	What to plot. See examples below.
`MSLevel`	`plotSpectrum()`	Whether to plot an MS or MS/MS spectrum (only `MSPeakLists`)
`formulas`	`plotSpectrum()`	Whether `formula` annotation should be added (only `compounds`)
`plotStruct`	`plotSpectrum()`	Whether the structure should be added to the plot (only `compounds`)

Note that we can use subsetting to select which feature data we want to plot, e.g.

```
plotChroms(fGroups[1:2]) # only plot EICs from first and second analyses.
```

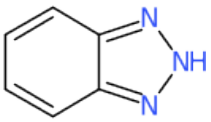


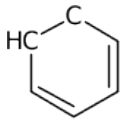

```
plotChroms(fGroups[, 1]) # only plot all features of first group
```



The `plotStructure()` function will draw a chemical structure for a compound candidate. In addition, this function can draw the maximum common substructure (MCS) of multiple candidates in order to assess common structural features.

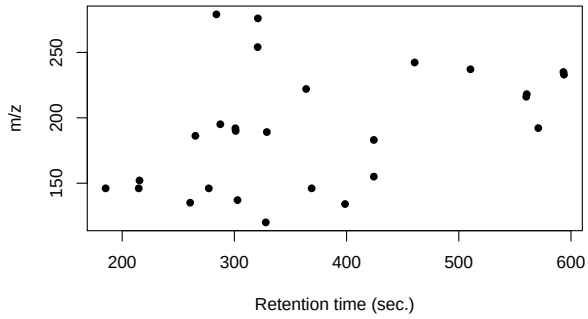
```
# structure for first candidate
plotStructure(compounds, index = 1, groupName = "M120_R328_81")
# MCS for first three candidates
plotStructure(compounds, index = 1:3, groupName = "M120_R328_81")
```



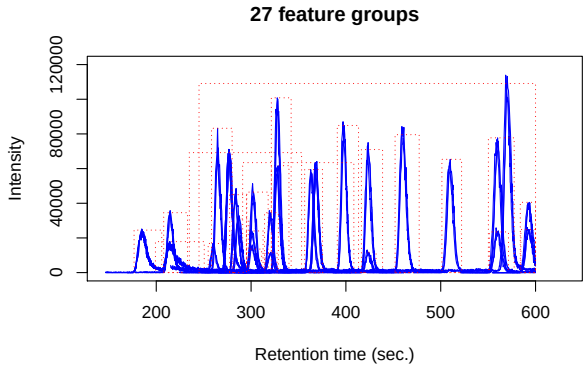


Some other common and less common plotting operations are shown below.

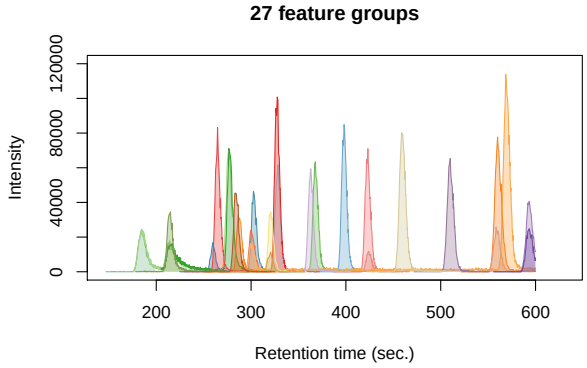
```
plot(fGroups) # simple scatter plot of retention and m/z values
```



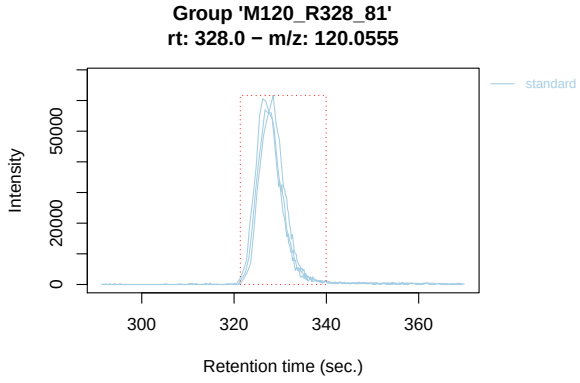

```
plotChroms(fGroups) # plot EICs for all features
```



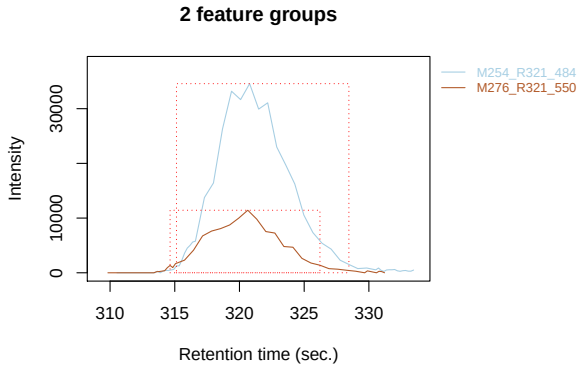

```
# get overview of all feature groups
plotChroms(fGroups,
           colourBy = "fGroup", # unique colour for each group
           topMost = 1, # only most intense feature in each group
           showPeakArea = TRUE, # show integrated areas
           showFGroupRect = FALSE,
           showLegend = FALSE) # no legend (too busy for many feature groups)
```



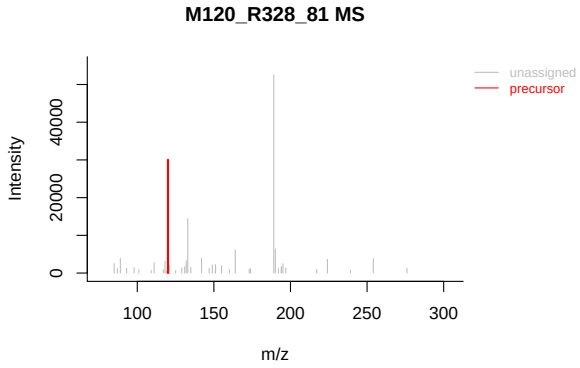

```
plotChroms(fGroups[, 1], # only plot all features of first group
           colourBy = "rGroup") # and mark them individually per replicate group
```



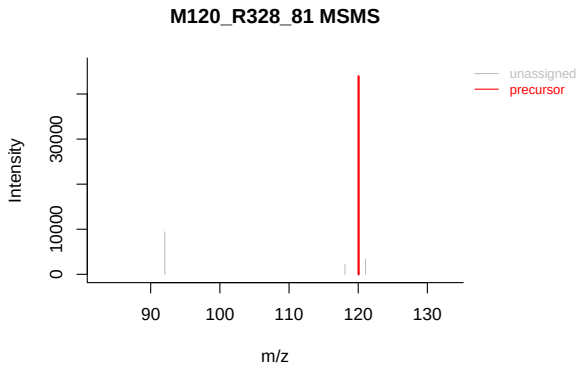
```
plotChroms(components, index = 7, fGroups = fGroups) # EICs from a component
```



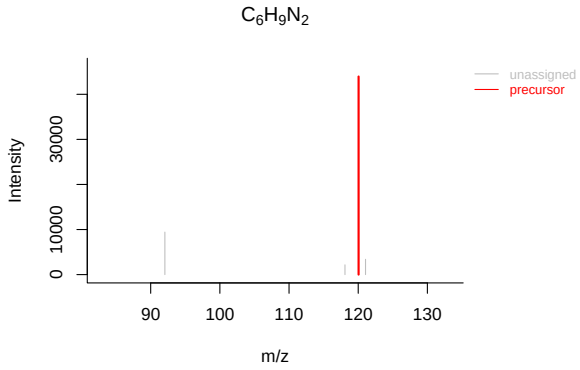

```
plotSpectrum(mslists, "M120_R328_81") # non-annotated MS spectrum
```



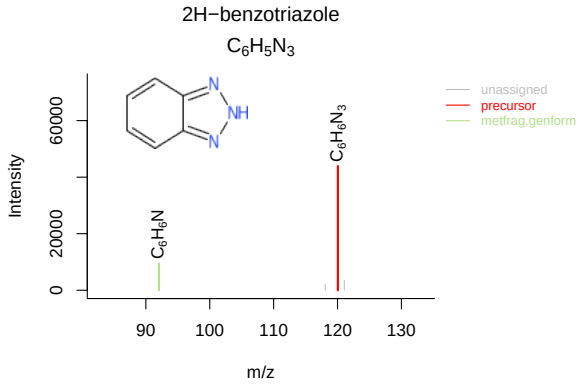

```
plotSpectrum(mslists, "M120_R328_81", MSLevel = 2) # non-annotated MS/MS spectrum
```



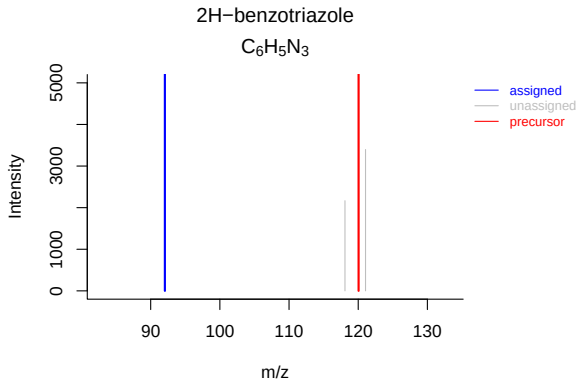

```
# formula annotated spectrum
plotSpectrum(formulas, precursor = "C6H9N2", groupName = "M120_R328_81",
             MSPeakLists = mslists)
```



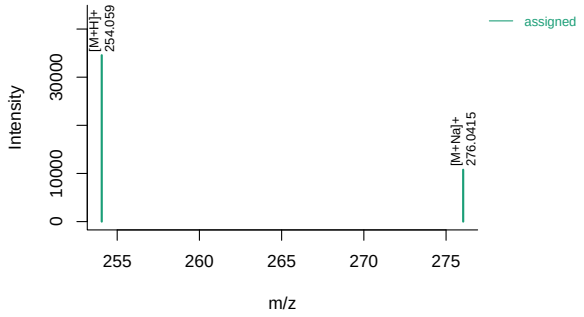

```
# compound annotated spectrum, with added formula annotations
plotSpectrum(compounds, index = 1, groupName = "M120_R328_81", MSPeakLists = mslists,
             formulas = formulas)
```




```
# custom intensity range (e.g. to zoom in)
plotSpectrum(compounds, index = 1, groupName = "M120_R328_81", MSPeakLists = mslists,
             ylim = c(0, 5000), plotStruct = FALSE)
```




```
plotSpectrum(components, index = 7) # component spectrum
```




```
# Inspect homologous series
plotGraph(componNT)
```

5.5.2 Comparing data

There are three functions that can be used to visualize overlap and uniqueness between data:

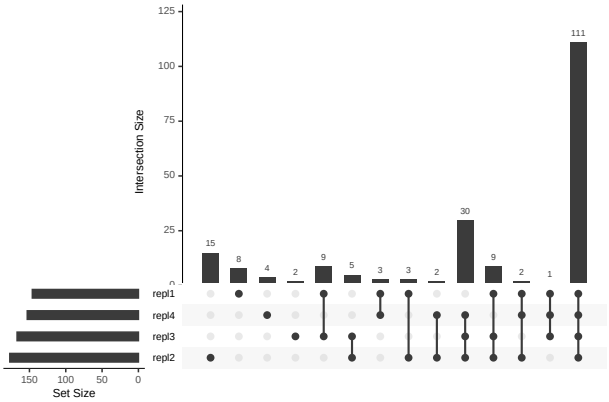
Generic	Classes
`plotVenn`	`featureGroups`, `featureGroupsComparison`, `formulas`, `compounds`
`plotUpSet`	`featureGroups`, `featureGroupsComparison`, `formulas`, `compounds`
`plotChord`	`featureGroups`, `featureGroupsComparison`

The most simple comparison plot is a Venn diagram (i.e. `plotVenn()`). This function is especially useful for two or three-way comparisons. More complex comparisons are better visualized with UpSet diagrams (i.e. `plotUpSet()`). Finally, chord diagrams (i.e. `plotChord()`) provide visually pleasing diagrams to assess overlap between data.

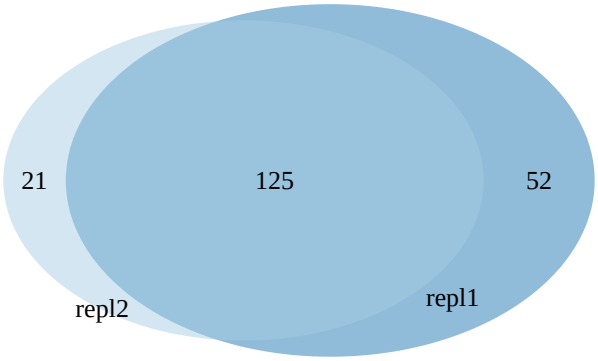
These functions can either be used to compare feature data or different objects of the same type. The former is typically used to compare overlap or uniqueness between features in different replicate groups, whereas comparison between objects is useful to visualize differences in algorithmic output. Besides visualization, note that both operations can also be performed to modify or combine objects (see unique and overlapping features and algorithm consensus).

As usual, some examples are shown below.

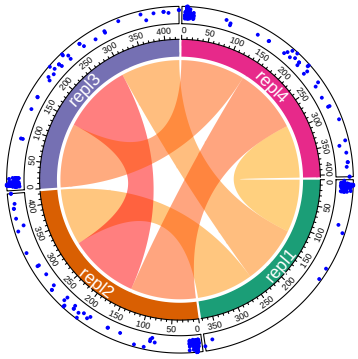
```
plotUpSet(fGroups) # compare replicate groups
```



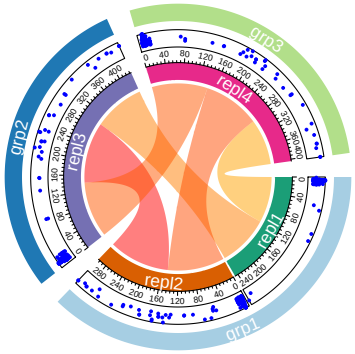

```
plotVenn(fGroups, which = c("repl1", "repl2")) # compare some replicate groups
```



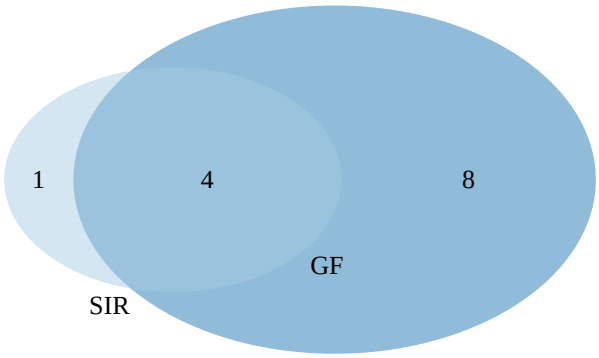

```
plotChord(fGroups, average = TRUE) # overlap between replicate groups
```




```
# compare with custom made groups
plotChord(fGroups, average = TRUE,
          outer = c(repl1 = "grp1", repl2 = "grp1", repl3 = "grp2", repl4 = "grp3"))
```




```
# compare GenForm and SIRIUS results
plotVenn(formsGF, formsSIR,
         labels = c("GF", "SIR")) # manual labeling
```

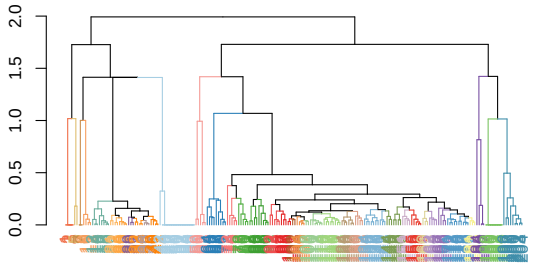


5.5.3 Hierarchical clustering results

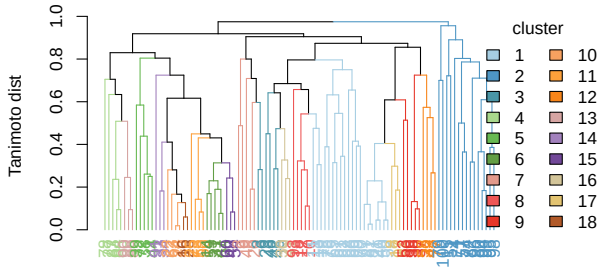
In `patRoon` hierarchical clustering is used to generate components based on their intensity profiles (see intensity clustering) and to cluster candidate compounds with similar chemical structure (see compound clustering). The functions below can be used to visualize their results.

Generic	Classes	Remarks
`plot()`	`componentsIntClust`, `compoundsCluster`	Plots a dendrogram
`plotInt()`	`componentsIntClust`	Plots normalized intensity profiles in a cluster
`plotHeatMap()`	`componentsIntClust`	Plots an heatmap
`plotSilhouettes()`	`componentsIntClust`	Plot silhouette information to determine the cluster amount
`plotStructure()`	`compoundsCluster`	Plots the maximum common substructure (MCS) of a cluster

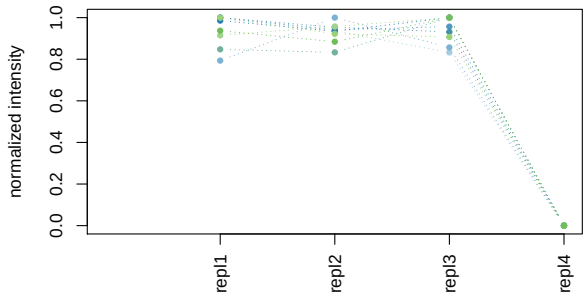
```
plot(componInt) # dendrogram
```



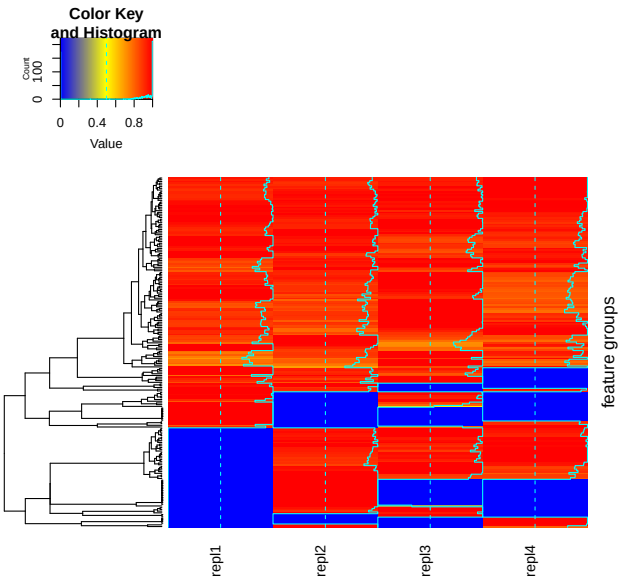

```
plot(compsClust, groupName = "M120_R328_81") # dendrogram for clustered compounds
```



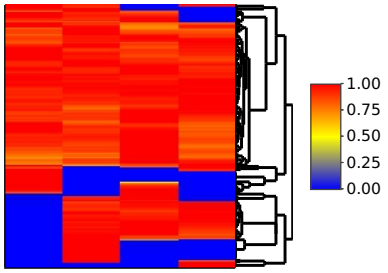

```
plotInt(componInt, index = 4) # intensities of 4th cluster
```



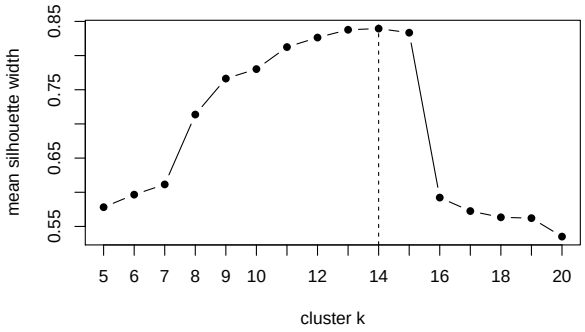

```
plotHeatMap(componInt) # plot heatmap
```




```
plotHeatMap(componInt, interactive = TRUE) # interactive heatmap (with zoom-in!)
```




```
plotSilhouettes(componInt, 5:20) # plot silhouettes (e.g. to obtain ideal cluster amount)
```



5.5.4 Interactive plotting of chromatography data

The `plotChroms()` function introduced before can be used to visualize chromatography data for one or more features. An interactive alternative is to call the `checkChromatograms()` function. This function will launch a GUI that allows you to browse through all features and inspect their EICs. Simply pass in the `featureGroups` object you want to inspect:

```
checkChromatograms(fGroups)
```

Note that this tool does not work well yet with large number of analyses/features. For this reason, it may be worthwile to launch it with subsets of your data, e.g.

```
checkChromatograms(fGroups[1:3, 1:250]) # only first 3 analyses and their first 500
↪  feature groups
```

Another purpose of the `checkChromatograms()` function is to remove 'bad' features (e.g. those which are probably not really features, but just noise). The workflow for this is:

1. Launch `checkChromatograms()` and remove any unwanted feature groups by disabling the *keep* checkbox.
2. Store the result of the `checkChromatograms()` function to a variable.
3. Use this variable to subset the original feature groups.

To do so:

```
keep <- checkChromatograms(fGroups)
fGroups <- fGroups[, keep]
```

Note that when you re-run `checkChromatograms()` you can restore the state of which feature groups should be kept/removed by passing the previous result to the function:

```
keep <- checkChromatograms(fGroups) # select feature groups

# continue at a later stage
keep <- checkChromatograms(fGroups, enabledFGroups = keep)
```

> ***NOTE*** Again, `checkChromatograms()` may be slow when processing larger datasets. The reporting functionalities provide a good alternative to quickly get an overview of all EIC data.

5.5.5 Generating EICs in DataAnalysis

If you have Bruker data and the DataAnalysis software installed, you can automatically add EIC data in a DataAnalysis session. The `addDAEIC()` will do this for a single *m/z* in one analysis, whereas the `addAllDAEICs()` function adds EICs for all features in a `featureGroups` object.

```
# add a single EIC with background subtraction
addDAEIC("mysample", "~/path/to/sample", mz = 120.1234, bgsubtr = TRUE)
# add TIC for MS/MS signal of precursor 120.1234 (value of mz is ignored for TICs)
addDAEIC("mysample", "~/path/to/sample", mz = 100, ctype = "TIC",
         mtype = "MSMS", fragpath = "120.1234", name = "MSMS 120")

addAllDAEICs(fGroups) # add EICs for all features
addAllDAEICs(fGroups[, 1:50]) # as usual, subsetting can be used for partial data
```

5.6 Reporting

The previous sections showed various functionalies to inspect and visualize results. An easy and automated way to do this automatically is by using the *reporting* functionality of `patRoon`. The following three reporting functions are available:

- `reportCSV()`: exports workflow data to comma-separated value (csv) files
- `reportPDF()`: generates simple reports by plotting workflow data in portable document files (PDFs)
- `reportHTML()`: generates interactive and easily explorable reports

There are many different arguments available to configure the reporting process. Some common arguments are listed below; for a complete listing see the reference manual (e.g. `?reporting`).

Argument	Functions	Remarks
`fGroups`, `formulas`, `compounds`, `formulas`, `components`, `compsCluster`	All	Objects to plot. Only `fGroups` is mandatory.
`MSPeakLists`	`reportPDF()`, `reportHTML()`	The `MSPeakLists` object that was used to generate annotation data. Only needs to be specified if `formulas` or `compounds` are reported.
`path`	All	Directory path where report files will be stored (`"report"` by default).
`formulasTopMost`, `compoundsTopMost`	`reportPDF()`, `reportHTML()`	Report no more than this amount of highest ranked candidates.
`EICOnlyPresent`	`reportPDF()`, `reportHTML()`	Only plot an EIC for an analysis if a feature was detected.
`selfContained`	`reportHTML()`	Outputs to a single and self contained `.html` file. Handy to share reports, but not recommended for large amounts of data.

Which data will be reported is fully configurable. The only workflow object that must be specified are the feature groups (i.e. with the `fGroups` argument), all other data (e.g. `compounds`, `components`) are optional. This means that reporting can be performed at every stage during the workflow, which, for instance, can be useful to quickly inspect results when testing out various settings to generate workflow data.

When formula or compound results are reported with `reportPDF()` or `reportHTML()` then only the top ranked candidates are considered. This limtation is often necessary as reporting many candidates will take considerable time. By default the top 5 for each feature group are reported, however, this number can be changed with the `formulasTopMost` and `compoundsTopMost` arguments.

Some typical examples:

```
reportHTML(fGroups) # simple interactive report with feature data
# generate PDFs with feature and compound annotation data
reportPDF(fGroups, compounds = compounds, MSPeakLists = mslists)
reportCSV(fGroups, path = "myReport") # change destination path

# generate report with all workflow types and increase maximum number of
# compound candidates to top 10
reportHTML(fGroups, formulas = formulas, compounds = compounds,
           components = components, MSPeakLists = mslists,
           compsCluster = compsClust,
           compoundsTopMost = 10)
```

6 Advanced usage

6.1 Adducts

When generating formulae and compound annotations and some other functionalities it is required to specify the adduct species. Behind the scenes, different algorithms typically use different formats. For instance, in order to specify a protonated species...

- `GenForm` either accepts `"M+H"` and `"+H"`
- `MetFrag` either accepts the numeric code `1` or `"[M+H]+"`
- `SIRIUS` accepts `"[M+H]+"`

In addition, most algorithms only accept a limited set of possible adducts, which do not necessarily all overlap with each other. The `GenFormAdducts()` and `MetFragAdducts()` functions list the possible adducts for `GenForm` and `MetFrag`, respectively.

In order to simplify the situation `patRoon` internally uses its own format and converts it automatically to the algorithm specific format when necessary. Furthermore, during conversion it checks if the specified adduct format is actually allowed by the algorithm. Adducts in `patRoon` are stored in the `adduct` S4 class. Objects from this class specify which elements are added and/or subtracted, the final charge and the number of molecules present in the adduct (e.g. *2* for a dimer).

```
adduct(add = "H") # [M+H]+
adduct(sub = "H", charge = -1) # [M-H]-
adduct(add = "K", sub = "H2", charge = -1) # [M+K-H2]-
adduct(add = "H3", charge = 3) # [M+H3]3+
adduct(add = "H", molMult = 2) # [2M+H]+
```

A more easy way to generate adduct objects is by using the `as.adduct()` function:

```
as.adduct("[M+H]+")
as.adduct("[M+H2]2+")
as.adduct("[2M+H]+")
as.adduct("[M-H]-")
as.adduct("+H", format = "genform")
as.adduct(1, isPositive = TRUE, format = "metfrag")
```

In fact, the `adduct` argument to workflow functions such as `generateFormulas()` and `generateCompounds()` is automatically converted to an `adduct` class with the `as.adduct()` function if necessary:

```
formulas <- generateFormulas(..., adduct = adduct(sub = "H", charge = -1))
formulas <- generateFormulas(..., adduct = "[M-H]-") # same as above
```

More details can be found in the reference manual (`?adduct` and `` ?`adduct-utils` ``).

6.2 Feature parameter optimization

Many different parameters exist that may affect the output quality of feature finding and grouping. To avoid time consuming manual experimentation, functionality is provided to largely automate the optimization process. The methodology, which uses design of experiments (DoE), is based on the excellent Isotopologue Parameter Optimization (IPO) `R` package. The functionality of this package is directly integrated in patRoon. Some functionality was added or changed, the most important being support for other feature finding and grouping algorithms besides XCMS and basic optimization support for qualitative parameters. Nevertheless, the core optimization algorithms are largely untouched.

This section will introduce the most important concepts and functionalities. Please see the reference manual for more information (e.g. `` ?`feature-optimization` ``).

6.2.1 Parameter sets

Before starting an optimization experiment we have to define *parameter sets*. These sets contain the parameters and (initial) numeric ranges that should be tested. A parameter set is defined as a regular `list`, and can be easily constructed with the `generateFeatureOptPSet()` and `generateFGroupsOptPSet()` functions (for feature finding and feature grouping, respectively).

```
pSet <- generateFeatureOptPSet("openms") # default test set for OpenMS
pSet <- generateFeatureOptPSet("openms", chromSNR = c(5, 10)) # add parameter
# of course manually making a list is also possible (e.g. if you don't want to test the
↪  default parameters)
pSet <- list(noiseThrInt = c(1000, 5000))
```

When optimizing with XCMS a few things have to be considered. First of all, when using the XCMS3 interface (i.e. `algorithm="xcms3"`) the underlying method that should be used for finding and grouping features and retention alignment should be set. In case these are not set default methods will be used.

```
pSet <- list(method = "centWave", ppm = c(2, 8))
pSet <- list(ppm = c(2, 8)) # same: centWave is default

# get defaults, but for different grouping/alignment methods
pSetFG <- generateFGroupsOptPSet("xcms3", groupMethod = "nearest", retAlignMethod =
↪  "peakgroups")
```

In addition, when optimizing feature grouping (both XCMS interfaces) we need to set the grouping and retention alignment parameters in two different nested lists: these are `groupArgs`/`retcorArgs` (`algorithm="xcms"`) and `groupParams`/`retAlignParams` (`algorithm="xcms3"`).

```
pSetFG <- list(groupParams = list(bw = c(20, 30))) # xcms3
pSetFG <- list(retcorArgs = list(gapInit = c(0, 7))) # xcms
```

When a parameter set has been defined it should be used as input for the `optimizeFeatureFinding()` or `optimizeFeatureGrouping()` functions.

```
ftOpt <- optimizeFeatureFinding(anaInfo, "openms", pSet)
fgOpt <- optimizeFeatureGrouping(fList, "openms", pSetFG) # fList is an existing features
↪  object
```

Similar to `findFeatures()`, the first argument to `optimizeFeatureFinding()` should be the analysis information. Note that it is not uncommon to perform the optimization with only a subset of (representative) analyses (i.e. to reduce processing time).

```
ftOpt <- optimizeFeatureFinding(anaInfo[1:2, ], "openms", pSet) # only use first two
↪  analyses
```

From the parameter set a design of experiment will be automatically created. Obviously, the more parameters are specified, the longer such an experiment will take. After an experiment has finished, the optimization algorithm will start a new experiment where numeric ranges for each parameter are increased or decreased in order to more accurately find optimum values. Hence, the numeric ranges specified in the parameter set are only *initial* ranges, and will be changed in subsequent experiments. After each experiment iteration the results will be evaluated and a new experiment will be started as long as better results were obtained during the last experiment (although there is a hard limit defined by the `maxIterations` argument).

For some parameters it is recommended or even necessary to set hard limits on the numeric ranges that are allowed to be tested. For instance, setting a minimum feature intensity threshold is highly recommended to avoid excessive processing time and potentially suboptimal results due to excessive amounts of resulting features. Configuring absolute parameter ranges is done by setting the `paramRanges` argument.

```
# set minimum intensity threshold (but no max)
ftOpt <- optimizeFeatureFinding(anaInfo, "openms",
                                list(noiseThrInt = c(1000, 5000), # initial testing range
                                paramRanges = list(noiseThrInt = c(500, Inf))) # never
                                ↪  test below 500
```

Depending on the used algorithm, several default absolute limits are imposed. These may be obtained with the `getDefFeaturesOptParamRanges()` and `getDefFGroupsOptParamRanges()` functions.

The common operation is to optimize numeric parameters. However, parameters that are not numeric (i.e. *qualitative*) need a different approach. In this case you will need to define multiple parameter sets, where each set defines a different qualitative value.

```
ftOpt <- optimizeFeatureFinding(anaInfo, "openms",
                                list(chromFWHM = c(4, 8), isotopeFilteringModel =
                                ↪  "metabolites (5% RMS)"),
                                list(chromFWHM = c(4, 8), isotopeFilteringModel =
                                ↪  "metabolites (2% RMS)"))
```

In the above example there are two parameter sets: both define the numeric `chromFWHM` parameter, whereas the qualitative `isotopeFilteringModel` parameter has a different value for each. Note that we had to specify the `chromFWHM` twice, this can be remediated by using the `templateParams` argument:

```
ftOpt <- optimizeFeatureFinding(anaInfo, "openms",
                                list(isotopeFilteringModel = "metabolites (5% RMS)"),
                                list(isotopeFilteringModel = "metabolites (2% RMS)"),
                                templateParams = list(chromFWHM = c(4, 8)))
```

As its name suggests, the `templateParams` argument serves as a template parameter set, and its values are essentially combined with each given parameter set. Note that current support for optimizing qualitative parameters is relatively basic and time consuming. This is because tests are essentially repeated for each parameter set (e.g. in the above example the `chromFWHM` parameter is optimized twice, each time with a different value for `isotopeFilteringModel`).

6.2.2 Processing optmization results

The results of an optimization process are stored in objects from the S4 `optimizationResult` class. Several methods are defined to inspect and visualize these results.

The `optimizedParameters()` function is used to inspect the best parameter settings. Similarly, the `optimizedObject()` function can be used to obtain the object that was created with these settings (i.e. a `features` or `featureGroups` object).

```
optimizedParameters(ftOpt) # best settings for whole experiment
```

```
#> $chromFWHM
#> [1] 3.5
#>
#> $mzPPM
#> [1] 13.5
#>
#> $minFWHM
#> [1] 1.5
#>
#> $maxFWHM
#> [1] 20
```

```
optimizedObject(ftOpt) # features object with best settings for whole experiment
```

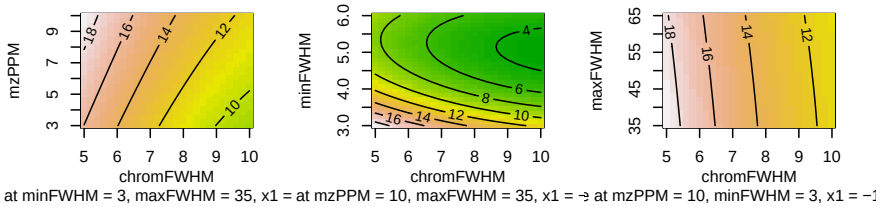
```
#> A featuresOpenMS object (derived from features -> workflowStep)
#> Object size (indication): 71.4 kB
#> Algorithm: openms
#> Total feature count: 409
#> Average feature count/analysis: 409
#> Analyses: solvent-1 (1 total)
#> Replicate groups: solvent (1 total)
#> Replicate groups used as blank: solvent (1 total)
```

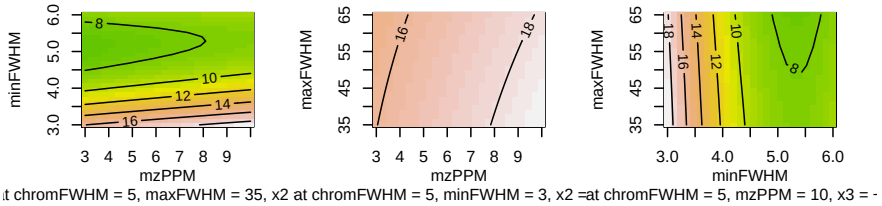
Results can also be obtained for specific parameter sets/iterations.

```
optimizedParameters(ftOpt, 1) # best settings for first parameter set
optimizedParameters(ftOpt, 1, 1) # best settings for first parameter set and experiment
↪  iteration
optimizedObject(ftOpt, 1) # features object with best settings for first parameter set
```

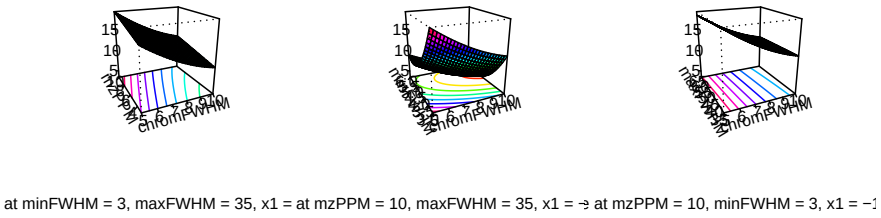
The `plot()` function can be used to visualize optimization results. This function will plot graphs for results of all tested parameter pairs. The graphs can be contour, image or perspective plots (as specified by the `type` argument).

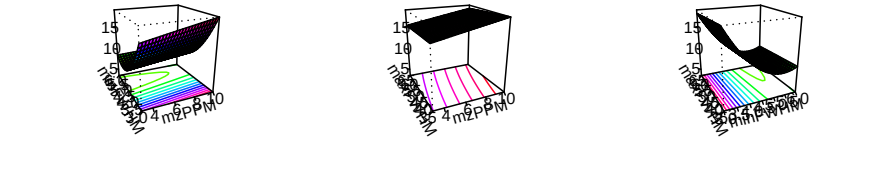
```
plot(ftOpt, paramSet = 1, DoEIteration = 1) # contour plots for first param
↪  set/experiment
```






```
plot(ftOpt, paramSet = 1, DoEIteration = 1, type = "persp") # pretty perspective plots
```





Please refer to the reference manual for more methods to inspect optimization results (e.g. `?optimizationResult`).

6.3 Exporting and converting feature data

The feature group data obtained during the workflow can be exported to various formats with the `export()` generic function. There are currently three formats supported: `"brukerpa"` (Bruker ProfileAnalysis), `"brukertasq"` (Bruker TASQ) and `"mzmine"` (mzMine). The former exports a 'bucket table' which can be loaded in ProfileAnalysis, the second and third export a target list that can be processed with TASQ and mzMine, respectively.

The `getXCMSSet()` function converts a `features` or `featureGroups` object to an `xcmsSet` object which can be used for further processing with xcms. Similarly, the `getXCMSnExp()` function can be used for conversion to an XCMS3 style `XCMSnExp` object.

Some examples for these functions are shown below.

```
export(fGroups, "brukertasq", out = "my_targets.csv")

# convert features to xcmsSet.
# NOTE: exportedData should only be FALSE when the analysis data files cannot be
# loaded by XCMS (e.g. when obtained with DataAnalysis)
xset <- getXCMSSet(fList, exportedData = TRUE)
xsetg <- getXCMSSet(fGroups, exportedData = TRUE) # get grouped xcmsSet

# using the new XCMS3 interface
# NOTE: for XCMS3 data currently always has to be exported
xdata <- getXCMSnExp(fList)
xdata <- getXCMSnExp(fGroups)
```

6.4 Algorithm consensus

With `patRoon` you have the option to choose between several algorithms for most workflow steps. Each algorithm is typically characterized by its efficiency, robustness, and may be optimized towards certain data properties. Comparing their output is therefore advantegeuous in order to design an optimum workflow. The `consensus()` generic function will compare different results from different algorithms and returns a *consensus*, which may be based on minimal overlap, uniqueness or simply a combination of all results from involved objects. The output from the `consensus()` function is of similar type as the input types and is therefore compatible to any 'regular' further data processing operations (e.g. input for other workflow steps or plotting). Note that a consensus can also be made from objects generated by the same algorithm, for instance, to compare or combine results obtained with different parameters (e.g. different databases used for compound annotation).

The `consensus()` generic is defined for most workflow objects. Some of its common function arguments are listed below.

Argument	Classes	Remarks
`obj`, `...`	All	Two or more objects (of the same type) that should be compared to generate the consensus.
`compThreshold`, `relAbundance`, `absAbundance`, `formThreshold`	`compounds`, `formulas`, `featureGroupsComparison`	The minimum overlap (relative/absolute) for a result (feature, candidate) to be kept.
`uniqueFrom`	`compounds`, `formulas`, `featureGroupsComparison`	Only keep *unique* results from specified objects.

Argument	Classes	Remarks
`uniqueOuter`	`compounds`, `formulas`, `featureGroupsComparison`	Should be combined with `uniqueFrom`. If `TRUE` then only results are kept which are *also* unique between the objects specified with `uniqueFrom`.

Note that current support for generating a consensus between `components` objects is very simplistic; here results are not compared, but the consensus simply consists a combination of all the components from each object.

Generating a consensus for feature groups involves first generating a `featureGroupsComparison` object. This step is necessary since (small) deviations between retention times and/or mass values reported by different feature finding/grouping algorithms complicates a direct comparison. The comparison objects are made by the `comparison()` function, and its results can be visualized by the plotting functions discussed in the previous chapter.

Some examples are shown below

```
compoundsCons <- consensus(compoundsMF, compoundsSIR) # combine MetFrag/SIRIUS results
compoundsCons <- consensus(compoundsMF, compoundsSIR,
                           compThreshold = 1) # only keep results that overlap

fGroupComp <- comparison(fGroupsXCMS, fGroupsOpenMS, fGroupsEnviPick,
                         groupAlgo = "openms")
plotVenn(fGroupComp) # visualize overlap/uniqueness
fGroupsCons <- consensus(fGroupComp,
                         uniqueFrom = 1:2) # only keep results unique in OpenMS+XCMS
fGroupsCons <- consensus(fGroupComp,
                         uniqueFrom = 1:2,
                         uniqueOuter = TRUE) # as above, but also exclude any overlap
                         ↪  between OpenMS/XCMS
```

6.5 Compound clustering

When large databases such as PubChem or ChemSpider are used for compound annotation, it is common to find *many* candidate structures for even a single feature. While choosing the right scoring settings can significantly improve their ranking, it is still very much possible that many candidates of potential interest remain. In this situation it might help to perform *compound clustering*. During this process, all candidates for a feature are clustered hierarchically on basis of similar chemical structure. From the resulting cluster the *maximum common substructure* (MCS) can be derived, which represents the largest possible substructure that 'fit' in all candidates. By visual inspection of the MCS it may be possible to identify likely common structural properties of a feature.

In order to perform compound clustering the `makeHCluster()` generic function should be used. This function heavily relies on chemical fingerprinting functionality provided by rcdk.

```
compounds <- generateCompounds(...) # get our compounds
compsClust <- makeHCluster(compounds)
```

This function accepts several arguments to fine tune the clustering process:

- `method`: the clustering method (e.g. `"complete"` (default), `"ward.D2"`), see `?hclust` for options
- `fpType`: finger printing type (`"extended"` by default), see `?get.fingerprint`

- `fpSimMethod`: similarity method for generating the distance method (`"tanimoto"` by default), see `?fp.sim.matrix`

For all arguments see the reference manual (`?makeHClust`).

The resulting object is of type `compoundsCluster`. Several methods are defined to modify and inspect these results:

```
# plot MCS of first cluster from candidates of M109_R116_61
plotStructure(compsClust, groupName = "M109_R116_61", 1)

# plot dendrogram
plot(compsClust, groupName = "M109_R116_61")

# re-assign clusters for a feature group
compsClust <- treeCut(compsClust, k = 5, groupName = "M109_R116_61")
# ditto, but automatic cluster determination
compsClust <- treeCutDynamic(compsClust, minModuleSize = 3, groupName = "M109_R116_61")
```

For a complete overview see the reference manual (`?compoundsCluster`).

6.6 Basic quantitative and regression analysis

While `patRoon` is currently primarily focused on qualitative analyses, some *basic* quantitative analysis can also be performed, for instance, to estimate concentrations of features. In fact, other types of data that may be useful for regression analysis can be set such as sample dilution factor or sampling time. The latter may, for instance, be used to isolate features with increasing or decreasing intensity. Regardless of what kind of regression analysis is performed, here we simply refer the values to be calculated as *concentrations*. In order to use this functionality, an extra column (`conc`) should be added to the analysis information, for instance:

```
# obtain analysis information as usual, but add some concentrations.
# The blanks are set to NA, whereas the standards are set to increasing levels.
anaInfo <- generateAnalysisInfo(paths = patRoonData::exampleDataPath(),
                                groups = c(rep("solvent", 3), rep("standard", 3)),
                                blanks = "solvent",
                                concs = c(NA, NA, NA, 1, 2, 3))
```

For analyses with known concentrations (e.g. standards) the `conc` column should be set; for all others the value should be set to `NA`.

The `as.data.table()` function (or `as.data.frame()`) can then be used to calculate regression data and estimate concentrations:

```
# use areas for quantitation and make sure that feature data is reported
# (otherwise no concentrations are calculated)
# (only relevant columns are shown for clarity)
as.data.table(fGroups, areas = TRUE, features = TRUE, regression = TRUE)
```

```
#>             group conc       RSQ intercept slope  conc_reg
#>  1:  M120_R328_81    1 0.4029224  55388.00  1844 0.2971800
#>  2:  M120_R328_81    2 0.4029224  55388.00  1844 3.4056399
#>  3:  M120_R328_81    3 0.4029224  55388.00  1844 2.2971800
```

```
#>  4: M134_R399_118    1 0.7969603  88396.00 -4962 0.7085852
#>  5: M134_R399_118    2 0.7969603  88396.00 -4962 2.5828295
#> ---
#> 77: M276_R321_550    2 0.9695239   9860.00   508 1.7952756
#> 78: M276_R321_550    3 0.9695239   9860.00   508 3.1023622
#> 79: M279_R284_557    1 0.6417469  47565.33 -2312 1.4313725
#> 80: M279_R284_557    2 0.6417469  47565.33 -2312 1.1372549
#> 81: M279_R284_557    3 0.6417469  47565.33 -2312 3.4313725
```

Calculated concentrations are stored in the `conc_reg` column, alongside while other regression data (i.e. `RSQ`, `intercept`, `slope` columns). To perform basic trend analysis the `RSQ` (i.e. R squared) can be used:

```
fGroupsTab <- as.data.table(fGroups, areas = TRUE, features = FALSE, regression = TRUE)
# subset fGroups with reasonable correlation
increasingFGroups <- fGroups[, fGroupsTab[RSQ >= 0.8, group]]
```

6.7 Caching

In `patRoon` lengthy processing operations such as finding features and generating annotation data is *cached*. This means that when you run such a calculation again (without changing any parameters), the data is simply loaded from the cache data instead of re-generating it. This in turn is very useful, for instance, if you have closed your R session and want to continue with data processing at a later stage.

The cache data is stored in a sqlite database file. This file is stored by default under the name `cache.sqlite` in the current working directory (for this reason it is very important to always restore your working directory!). However, the name and location can be changed by setting a global package option:

```
options(patRoon.cache.fileName = "~/myCacheFile.sqlite")
```

For instance, this might be useful if you want to use a shared cache file between projects.

After a while you may see that your cache file can get quite large. This is especially true when testing different parameters to optimize your workflow. Furthermore, you may want to clear the cache after you have updated `patRoon` and want to make sure that the latest code is used to generate the data. At any point you can simply remove the cache file. A more fine tuned approach which doesn't wipe all your cached data is by using the `clearCache()` function. With this function you can selectively remove parts of the cache file. The function has two arguments: `what`, which specifies what should be removed, and `file` which specifies the path to the cache file. The latter only needs to be specified if you want to manage a different cache file.

In order to figure what is in the cache you can run `clearCache()` without any arguments:

```
clearCache()
```

```
#> Please specify which cache you want to remove. Available are:
#> - EICData (3 rows)
#> - MSPeakListsAvg (4 rows)
#> - MSPeakListsMzR (104 rows)
#> - componentsCAMERA (1 rows)
#> - componentsNontarget (1 rows)
#> - componentsRC (1 rows)
#> - compoundsCluster (1 rows)
#> - featureGroupsOpenMS (3 rows)
```

```
#> - featuresOpenMS (55 rows)
#> - filterFGroups_blank (3 rows)
#> - filterFGroups_intensity (7 rows)
#> - filterFGroups_minReplicates (25 rows)
#> - filterFGroups_replicateAbundance (6 rows)
#> - filterFGroups_replicate_group (16 rows)
#> - filterFGroups_retention (2 rows)
#> - filterMSPeakLists (2 rows)
#> - formulasFGroupConsensus (2 rows)
#> - formulasGenForm (104 rows)
#> - formulasSIRIUS (6 rows)
#> - loadIntensities (55 rows)
#> - metfrag (30 rows)
#> - mzREIC (266 rows)
#> - reportPlots (527 rows)
#> - specData (6 rows)
#> - all (removes complete cache database)
```

Using this output you can re-run the function again, for instance:

```
clearCache("featuresOpenMS")
clearCache(c("featureGroupsOpenMS", "formulasGenForm")) # clear multiple
clearCache("OpenMS") # clear all with OpenMS in name (ie partial matched)
clearCache("all") # same as simply removing the file
```

6.8 Parallelization

patRoon relies on several external (command-line) tools to generate workflow data. Some of these tools are computationally heavy, and it may therefore take long before they finish processing large NTA datasets. In order to reduce computation times, these commands are executed in *parallel*. Running several commands simultaneously is especially advantageous on multi-core CPUs. The table below outlines the tools that are executed in parallel.

Tool	Used by	Notes
msConvert	convertMSFiles(algorithm="pwiz", ...)	
FileConverter	convertMSFiles(algorithm="openms", ...)	
FeatureFinderMetabo	generateFeatures(algorithm="openms", ...)	
GenForm	generateFormulas(agorithm="genform", ...)	
SIRIUS	generateFormulas(agorithm="sirius", ...), generateCompounds(agorithm="sirius", ...)	Only if splitBatches=TRUE
MetFrag	generateCompounds(agorithm="metfrag", ...)	
pngquant	reportHTML(...)	Only if optimizePng=TRUE

Two parallelization approaches are available: classic, which uses the processx R package to execute multiple

tools in parallel, and `future`, where so called "futures" are created by the future.apply `R` package. An overview of the characteristics of both parallelization methods is shown below.

`classic`	`future`
requires little or no configuration	configuration needed to setup
works with all tools	doesn't work with `pngquant` and slower with `GenForm`
only supports parallelization on the local computer	allows both local and cluster computing

Which method is used is controlled by the `patRoon.MP.method` package option. Note that `reportHTML()` will always use the classic method for `pngquant`.

6.8.1 Classic parallelization method

The classic method is the 'original' method implemented in `patRoon`, and is therefore well tested and optimized. It is easier to setup, works well with all tools, and is therefore the default method. It is enabled as follows:

```
options(patRoon.MP.method = "classic")
```

The number of parallel processes is configured through the `patRoon.MP.maxProcs` option. By default it is set to the number of available CPU cores, which results usually in the best performance. However, you may want to lower this, for instance, to keep your computer more responsive while processing or limit the RAM used by the data processing workflow.

```
options(patRoon.MP.maxProcs = 2) # do not execute more than two tools in parallel.
```

This will change the parallelization for the complete workflow. However, it may be desirable to change this for only a part the workflow. This is easily achieved by using the `withOpt()` function.

```
# do not execute more than two tools in parallel.
options(patRoon.MP.maxProcs = 2)

# ... but execute up to four GenForm processes
withOpt(MP.maxProcs = 4, {
    formulas <- generateFormulas(fGroups, "genform", ...)
})
```

The `withOpt` function will temporarily change the given option(s) while executing a given code block and restore it afterwards (it is very similar to the `with_options()` function from the withr `R` package). Furthermore, notice how `withOpt()` does not require you to prefix the option names with `patRoon.`.

6.8.2 Future parallelization method

The primary goal of the "future" method is to allow parallel processing on one or more external computers. Since it uses the future `R` package, many approaches are supported, such as local parallelization (similar to the `classic` method), cluster computing via multiple networked computers and more advanced HPC approaches such as `slurm` via the future.batchtools `R` package. This parallelization method can be activated as follows:

```r
options(patRoon.MP.method = "future")

# set a future plan

# example 1: start a local cluster with four nodes
future::plan("cluster", workers = 4)

# example 2: start a networked cluster with four nodes on PC with hostname "otherpc"
future::plan("cluster", workers = rep("otherpc", 4))
```

It is important to properly configure the right future plan. Please see the documentation of the respective packages (*e.g.* future and future.batchtools) for more details.

The `withOpt()` function introduced in the previous section can also be used to temporarily switch between parallelization approaches, for instance:

```r
# default to future parallelization
options(patRoon.MP.method = "future")
future::plan("cluster", workers = 4)

# ... do workflow

# do classic parallelization for GenForm
withOpt(MP.method = "classic", {
    formulas <- generateFormulas(fGroups, "genform", ...)
})

# .. do more workflow
```

By default, no progress bars are visible when using the future method (this may change in the future). The reason for this is that the progressr package, which is used to report progress, requires the user to configure *how* progress should be reported. While this is a bit of extra work, it allows many different ways to report progress. You can find more information on the progressr website. While this may change in the future, for now each function should be wrapped within a call to `progressr::with_progress()`, e.g.

```r
# setup parallelization etc

progressr::with_progress({
    compounds <- generateCompounds(fGroups, "metfrag", ...)
})
```

Some more important notes when using the `future` parallelization method:

- As highlighted in the table at the beginning of this section, `GenForm` currently performs less optimal with future processing compared to with the `classic` approach. Nevertheless, it may still be interesting to use the `future` method to move the computations to another system to free up resources on your local system.
- Behind the scenes the future.apply package is used to schedule the tools to be executed. The `patRoon.MP.futureSched` option sets the value for the `future.scheduling` argument to the `future_lapply()` function, and therefore allows you to tweak the scheduling.
- Make sure that `patRoon` and the tool to be executed (`MetFrag`, `SIRIUS` etc.) are exactly the *same* version on all computing hosts.

- Make sure that `patRoon` is properly configured on all hosts, *e.g.* set the `patRoon.path.XXX` options to ensure all tools can be found.
- For `MetFrag` annotation: if a local database such as `PubChemLite` is used, it must be present on each computing node as well. Furthermore, the local computer (even if not used for the computations) *also* must have this file present. Like the previous point, make sure that the `patRoon.path.XXX` options are set properly.
- If you encounter errors then it may be handy to switch to `future::plan("sequential")` and see if it works.
- In order to restart the nodes, for instance after re-configuring `patRoon`, updating `R` packages etc, simply re-execute `future::plan(...)`.
- Setting the `future.debug` package option to `TRUE` may give you more insight what is happening and may therefore be interesting for debugging e.g. problems.
- Take care to look for the log files (next section) if you encounter any errors.

6.8.3 Logging

Most tools that are executed in parallel will log their output to text files. These files may be highly useful to check, for instance, if an error occurred. By default, the logfiles are stored in the `log` directory placed in the current working directory. However, you can change this location by setting the `patRoon.MP.logPath` option. If you set this option to `FALSE` than no logging occurs.