

patRoon handbook

Rick Helmus

2026-04-08

Contents

1	Introduction	3
2	Installation	3
2.1	patRoon Bundle	4
2.2	Docker image	5
2.3	Regular R installation	5
2.4	Managing legacy installations	10
2.5	Developers: devtools / pkgload	11
3	Workflow concepts	11
4	Generating workflow data	12
4.1	Introduction	12
4.2	Data processing projects	13
4.3	Sample analyses	15
4.4	Example data	19
4.5	Features	21
4.6	Componentization	27
4.7	Incorporating adduct and isotopic data	30
4.8	Annotation	32
5	Processing workflow data	43
5.1	Inspecting results	43
5.2	Filtering	49
5.3	Subsetting	55
5.4	Deleting data	57
5.5	Interactively explore and review data	59
5.6	Updating feature group data	61

5.7	Unique and overlapping features	61
5.8	MS similarity	62
5.9	Visualization	64
5.10	Use and modify sample analysis metadata	78
5.11	Reporting	82
6	Sets workflows	85
6.1	Initiating a sets workflow	86
6.2	Generating sets workflow data	88
6.3	Selecting adducts to improve grouping	89
6.4	Processing data	91
6.5	Advanced	93
7	Transformation product screening	96
7.1	Obtaining transformation product data	96
7.2	Linking parent and transformation product features	103
7.3	Example workflows	107
8	Ion mobility spectrometry (IMS-HRMS) workflows	111
8.1	Introduction	111
8.2	Performing IMS workflows	113
8.3	Processing data	123
8.4	Example workflow	126
9	Advanced usage	128
9.1	Adducts	128
9.2	Default numeric limits and tolerances	129
9.3	Feature intensity normalization	130
9.4	Feature parameter optimization	133
9.5	Chromatographic peak qualities	138
9.6	Exporting and converting feature data	139
9.7	Algorithm consensus	140
9.8	Background removal in MS/MS data	141
9.9	MS libraries	142
9.10	Compound clustering	145
9.11	Feature regression analysis	146
9.12	Predicting toxicities and concentrations (MS2Tox and MS2Quant integration)	148
9.13	Fold changes	154
9.14	Caching	156
9.15	Parallelization	157

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 **patRoön** 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, **patRoön** 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, **patRoön** supplies a straightforward interface to easily inspect, select, visualize and report all data that is generated during the workflow.

The documentation of **patRoön** consists of three parts:

1. A tutorial (accessible at [here](#))
2. This handbook
3. The reference manual (accessible in R with `?`patRoön-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 **patRoön**. The handbook provides a more thorough overview of all concepts, functionalities and provides instructions and many examples on working with **patRoön**. 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

This chapter outlines several strategies to install **patRoön** and its dependencies. These include other R packages and software tools external to R. The following strategies can largely automate this process, and will be discussed in the next sections:

1. The **patRoön** bundle, which contains all dependencies (including R), and is therefore very easy to setup (currently *Windows only*).
2. Reproducible Docker images.
3. Regular installations that integrate with the currently installed R environment.

The first strategy is recommended if you are using Windows and are new to R, or quickly want to try out the latest **patRoön** snapshot. Docker images are specifically for users who wish to run isolated containers and ensure high reproducibility. Finally, people already running R will most likely prefer the third strategy. Each strategy is discussed separately in the next sections.

2.1 patRoön Bundle

The **patRoön** bundle contains an almost full **patRoön** installation, including R, all R package dependencies and external software dependencies such as Java JDK, MetFrag and various compound libraries etc. Currently, only ProteoWizard and patRoönDataIMS are not included in the bundle due to licensing and size issues, respectively. See the regular installation section for further details on how to install these.

The bundles are automatically generated and tested, and can be obtained from the release page on GitHub for released versions of **patRoön** and the latest pre-release on GitHub for the latest snapshot.

After downloading the bundle, simply extract the .zip file. Then, a classic R terminal can be launched by executing `R/bin/x64/Rgui.exe` inside the directory where the bundle was extracted. However, it is probably more convenient to use it from RStudio:

Start RStudio -> Tools menu -> Global options -> General tab -> R version -> Change

Then, set the R version by selecting `Rterm.exe` from the `R/bin/x64` directory in the bundle (see screenshot below) and restart RStudio.



2.1.1 Updating the bundle

To update the bundle run either of the following functions:

```
patRoönInst::sync(allDeps = TRUE) # synchronize all packages related to patRoön to the  
↪ currently tested versions  
patRoönInst::update() # update all R packages related to patRoön
```

Both functions will update **patRoön** and related packages to their latest versions. However, they differ on handling their dependencies.

In general, it is recommended to synchronize the package dependencies in the bundle, since this ensures that versions were tested with **patRoön**. If you installed any other packages and also want to update these, then *first* do so with regular mechanisms (e.g. `update.packages()`, `BiocManager::install()`) and *then* synchronize **patRoön** to ensure that all packages are with tested versions.

However, if you prefer to install the latest version of all dependencies, then running `patRoön::update()` might be more appropriate. In this case, it is still recommended to first update any ‘regular’ R packages as described above, as `patRoönInst::update()` may install some dependencies with a specific version in case other versions are known to not work.

More details on using **patRoönInst** to manage installations are discussed later.

2.1.2 Details

This section describes details on the contents and the configuration of the **patRoön** bundle, and is mainly intended for readers who want to know more details or perform customizations.

The **patRoön** bundle consists of the following:

- A complete installation of R.

- An open java development kit (JDK) from Adoptium
- **patRoön** and its mandatory and optional R packages dependencies, synchronized from **patRoönDeps** (discussed later).
- Most external dependencies *via* **patRoönExt** (also discussed later)

The R Windows installers are extracted with innoextract to obtain a ‘portable’ installation. The **Renviron.site** and **Rprofile.site** files are then generated to ensure that the bundled JDK will be used, R packages will be loaded and installed from the bundle and various other configurations are applied to ensure that the bundle will not conflict with a regular R installation.

The bundles are automatically generated, and the relevant script can be found [here](#).

2.2 Docker image

Docker images are provided to easily install a reproducible environment with R, **patRoön** 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 **patRoön** were originally only used for automated testing, however, since these contain a complete working environment of **patRoön** they are also suitable for using the software. They come with all external dependencies (except ProteoWizard), R dependencies and **MetFrag** libraries. Furthermore, the Docker image also contains RStudio server, which makes using **patRoön** even easier.

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

```
# run an interactive R console session
docker run --rm -it uva-hva.gitlab.host:4567/r.helmus/patroon/patroonrs

# run a linux shell, from which R can be launched
docker run --rm -it uva-hva.gitlab.host:4567/r.helmus/patroon/patroonrs bash

# run rstudio server, accessible from localhost:8787
# login with rstudio/yourpasswordhere
docker run --rm -p 8787:8787 -u 0 -e PASSWORD=yourpasswordhere
→ uva-hva.gitlab.host:4567/r.helmus/patroon/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 uva-hva.gitlab.host:4567/r.helmus/patroon/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 in 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 Regular R installation

A ‘regular’ installation involves installing **patRoön** and its dependencies using the local installation of R. This section outlines available tools to do this mostly automatically using the auxiliary **patRoönInst** and **patRoönExt** R packages, as well as instructions to perform the complete installation manually.

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.

2.3.1 Automatic installation

The `patRooinst` auxiliary package simplifies the installation process. This package automatically installs all R package dependencies, including those unavailable from regular repositories such as CRAN and Bioconductor. Furthermore, `patRooinst` installs `patRooinstExt`, an R package that bundles most common dependencies external to the R environment (e.g. MetFrag, OpenMS etc).

NOTE ProteoWizard needs to be installed manually due to license restrictions. See the manual installation section for further details.

The first step is to install `patRooinst`:

```
install.packages("patRooinst", repos = c('https://rickhelmus.r-universe.dev',
  ↪ 'https://cloud.r-project.org'))

# or alternatively, from GitHub
install.packages("remotes") # run this in case the remotes (or devtools) package is not
  ↪ yet installed
remotes::install_github("rickhelmus/patRooinst")
```

Then to perform an installation or update:

```
patRooinst::install() # install patRooinst and any missing dependencies
patRooinst::update() # update patRooinst and its dependencies
```

The installation can be customized in various ways. Firstly, the repositories used to download R packages can be customized through the `origin` argument. The following options are currently available:

- **patRooinstDeps**: contains `patRooinst` and its dependencies (including *their* dependencies) with versions that were tested against the latest `patRooinst` version. This repository is used for the `patRooinst` bundle, and only available for Windows systems.
- **r-universe**: contains a snapshot of the latest version of `patRooinst` and its direct dependencies.
- **“regular”**: in this case packages will be sourced directly from CRAN/Bioconductor or GitHub. This means that suitable build tools (e.g. Rtools on Windows) need to be available during installation.

The default on Windows systems is `patRooinstDeps`, and `r-universe` otherwise. Note that both repositories only provide packages for recent R versions.

Other installation customizations include which packages will be installed (or updated), and installing all packages to an isolated R library. Some examples:

```
# install from r-universe
patRooinst::install(origin = "runiverse")
# only install patRooinst, without optional dependencies and directly from GitHub
patRooinst::install(origin = "regular", pkgs = "patRooinst")
# full installation, except two selected packages
patRooinst::install(ignorePkgs = c("nontarget", "MetaClean"))
```

```
# full installation, but exclude 'big' optional dependencies such as example data
↳ (patRoonDelete)
patRoonDelete::install(ignorePkgs = "big")
# install everything to an isolated R library (use .libPaths() to use it)
patRoonDelete::install(lib.loc = "~/patRoonDelete-lib")
```

Besides installing and updating packages, it is also possible to *synchronize* them with the selected repository using the `sync()` function. This is mostly the same as `update()`, but can also downgrade packages to ensure their versions exactly match that of the repository. This is currently only supported for the `patRoonDeleteDeps` repository. Furthermore, as synchronization may involve downgrading it is intended for environments that are primarily used for `patRoonDelete`, such as the bundle and isolated R libraries. Synchronization can be performed for all or only direct dependencies:

```
patRoonDeleteInst::sync(allDeps = TRUE) # synchronize all dependencies
patRoonDeleteInst::sync(allDeps = FALSE) # synchronize only direct dependencies
```

More options are available to customize the installation, see the reference manual (`?patRoonDeleteInst::install`) for more details.

2.3.2 Manual installation

A manual installation starts with installing external dependencies, followed by R dependencies and `patRoonDelete` itself.

2.3.2.1 External (non-R) dependencies `patRoonDelete` interfaces with various software tools that are external to R. A complete overview is given in the table below

Dependency	Remarks
Java JDK	Mandatory for e.g. plotting structures and using MetFrag. Highly recommend Used by e.g. suspect screening to automatically validate and calculate chemical properties such as InChIs and formulae. While optional, highly recommended.
OpenBabel	
Rtools	
ProteoWizard	May be necessary on Window when installing <code>patRoonDelete</code> and its R dependencies (discussed later). Needed for automatic data-pretreatment (e.g. data file conversion and centroiding, Bruker users may use DataAnalysis integration instead).
TIMSCONVERT	Recommended to convert Bruker timsTOF data. Needs a Python installation and the <code>reticulate</code> R package. The <code>installTIMSCONVERT()</code> function can be used to automatically install TIMSCONVERT in a virtual Python environment. For manual installation, see the installation instructions (“Manual installation”, the GUI does not provide the necessary files).
OpenMS	Recommended. Used for e.g. finding and grouping features.
TDF-SDK	Install for directly loading Bruker timsTOF data <i>via</i> the OpenTIMS backend.
MetFrag CL	Recommended. Used for annotation with MetFrag.

Dependency	Remarks
MetFrag CompTox DB	Database files necessary for usage of the CompTox database with MetFrag. Note that a recent version of MetFrag ($\geq 2.4.5$) is required. Note that the lists with additions for smoking metadata and wastewater metadata are also supported.
MetFrag PubChemLite DB	Database file needed to use PubChemLite with MetFrag. For IMS workflows, see below.
MetFrag PubChemLite+CCSbase DB	Database file needed to use the PubChemLite+CCSbase database with MetFrag. This database is recommended for IMS workflows, as it contains CCS values. It will also work for regular HRMS workflows, and is therefore provided by patRoanExt .
MetFrag PubChem OECD PFAS DB SIRIUS	Database file to use the OECD PFAS database with MetFrag. For obtaining feature data and formula and/or compound annotation.
BioTransformer	For prediction of transformation products. See the BioTransformer page for installation details. If you have trouble compiling the jar file you can download it from here.
C3SDB (CCSbase)	Used for CCS prediction in IMS workflows. Requires a Python 3.12 installation and the reticulate R package. The <code>installC3SDB()</code> function will automatically install C3SDB in a Python virtual environment.
SAFD	For finding features with SAFD. Please follow all the installation on the SAFD webpage.

Most of these dependencies are optional and only needed if their algorithms are used during the workflow.

2.3.2.1.1 Installation via patRoanExt The **patRoanExt** auxiliary package automatizes the installation of most common external dependencies. For installation, just run:

```
install.packages("remotes") # run this if remotes (or devtools) is not already installed
remotes::install_github("rickhelmus/patRoanExt")
```

NOTE Make sure you have an active internet connection since several files will be downloaded during the installation of **patRoanExt**.

Note that when you do an automated **patRoan** installation this package is automatically installed. See the project page for more details, including ways to customize which software tools will be installed.

NOTE Currently, **patRoanExt** does not install **ProteoWizard** due to license restrictions, and some tools, such as **OpenMS** and **OpenBabel**, are only installed on Windows systems. See the next section to install any missing tools manually.

2.3.2.1.2 Manually installing and configuring external tools Download the tools manually from the linked sources shown in the table above, and subsequently install (or extract) them. You may need to configure their file paths afterwards (**OpenMS**, **OpenBabel** and **ProteoWizard** are often found automatically). To configure the file locations you should set some global package options with the `options()` R function, for instance:


```

options(patRoos.path.pwiz = "C:/ProteoWizard") # location of ProteoWizard installation
↳ folder
options(patRoos.path.SIRIUS = "C:/sirius-win64-3.5.1") # directory with the SIRIUS
↳ binaries
options(patRoos.path.OpenMS = "/usr/local/bin") # directory with the OpenMS binaries
options(patRoos.path.MetFragCL = "~/MetFragCommandLine-2.4.8.jar") # full location to the
↳ jar file
options(patRoos.path.MetFragCompTox = "C:/CompTox_17March2019_SelectMetaData.csv") # full
↳ location to desired CompTox CSV file
options(patRoos.path.MetFragPubChemLite = "~/PubChemLite_exposomics_20220429.csv") # full
↳ location to desired PubChemLite CSV file
options(patRoos.path.MetFragPubChemLite = "~/PubChem_OECDPFAS_largerPFASparts_20220324")
↳ # full location to PFAS DB (NOTE: configured like PubChemLite)
options(patRoos.path.BioTransformer = "~/biotransformer/biotransformer-3.0.0.jar")
options(patRoos.path.obabel = "C:/Program Files/OpenBabel-3.0.0") # directory with
↳ OpenBabel binaries
options(patRoos.path.TDFSdk = "C:/timsdata/win64/timsdata.dll") # path to TDF-SDK library
↳ (.dll or .so)

```

These commands have to be executed every time 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).

NOTE The tools that are configured through the `options()` described above will *override* any tools that were *also* installed through `patRoosExt`. Hence, this mechanism can be used to use specific versions not available through `patRoosExt`. However, this also means that you need to ensure that options are unset when you prefer that tools are used through `patRoosExt`.

2.3.2.2 Installing patRoos and its R dependencies The table below lists all the R packages that are involved in the installation of `patRoos`.

Note that only the `CAMERA` installation is mandatory, the rest involves installation of *optional* packages. If you are unsure which you need then you can always install the packages at a later stage.

The last three columns of the table provide hints on the availability from the `patRoosDeps`, `r-universe` and original regular sources (the sources were discussed previously). Note that you may need to install `remotes`, `BiocManager` and `Rtools` if packages are installed from their regular source. Some examples are shown below:

```

# Install patRoos (and its mandatory dependencies) from patRoosDeps
install.packages("patRoos", repos = "https://rickhelmus.github.io/patRoosDeps", type =
↳ "binary")

# Install KPIC2 from r-universe
install.packages("KPIC", repos = c('https://rickhelmus.r-universe.dev',
↳ 'https://cloud.r-project.org'))

# Install the mandatory CAMERA package (will be installed automatically if using
↳ patRoosDeps/r-universe)
install.packages("BiocManager") # execute this if 'BiocManager' is not yet installed
BiocManager::install("CAMERA")

# Install patRoosData from GitHub

```

package	comments	patRoondEps	r-universe	regular installation
CAMERA	Mandatory	no	no	<code>'BiocManager::install('CAMERA')'</code>
RDCOMClient	Only for windows	no	no	<code>'remotes::install_github('BSchamberger/RDCOMClient')'</code>
ff	Dependency of RAMClustR	no	no	<code>'install.packages('ff')'</code>
Rdisop	Dependency of InterpretMSSpectrum	no	no	<code>'BiocManager::install('Rdisop')'</code>
InterpretMSSpectrum	Dependency of RAMClustR	no	yes	<code>'install.packages('InterpretMSSpectrum')'</code>
RAMClustR		no	yes	<code>'remotes::install_github('cbroekl/RAMClustR@master')'</code>
enviPick		no	yes	<code>'remotes::install_github('blosloos/enviPick')'</code>
nontargetData	Dependency of nontarget	no	yes	<code>'remotes::install_github('blosloos/nontargetData')'</code>
nontarget		no	yes	<code>'remotes::install_github('blosloos/nontarget')'</code>
ropIs	Dependency of KPIC	no	no	<code>'BiocManager::install('ropIs')'</code>
KPIC		no	yes	<code>'remotes::install_github('rickhelmus/KPIC2')'</code>
cliqueMS		no	yes	<code>'remotes::install_github('rickhelmus/cliqueMS')'</code>
BiocStyle	Dependency of MetaClean	no	no	<code>'BiocManager::install('BiocStyle')'</code>
Rgraphviz	Dependency of MetaClean	no	no	<code>'BiocManager::install('Rgraphviz')'</code>
fastAdaboost	Dependency of MetaClean	no	yes	<code>'remotes::install_github('souravc83/fastAdaboost')'</code>
MetaClean		no	yes	<code>'remotes::install_github('KelseyChetnik/MetaClean')'</code>
MetaCleanData		no	no	<code>'remotes::install_github('KelseyChetnik/MetaCleanData')'</code>
splashR		no	yes	<code>'remotes::install_github('berlinguyinca/spectra-hash')'</code>
MS2Tox		no	no	<code>'remotes::install_github('rickhelmus/MS2Tox@model_fix')'</code>
MS2Quant		no	yes	<code>'remotes::install_github('kruevelab/MS2Quant@main')'</code>
reticulate		no	no	<code>'install.packages('reticulate')'</code>
ChemmineR	Dependency of fmcsR	no	no	<code>'BiocManager::install('ChemmineR')'</code>
fmcsR		no	no	<code>'BiocManager::install('fmcsR')'</code>
Rmstoolkitlib		no	yes	<code>'remotes::install_github('rickhelmus/Rmstoolkitlib@main')'</code>
patRoondData		no	no	<code>'remotes::install_github('rickhelmus/patRoondData')'</code>
patRoondDataIMS		no	no	<code>'remotes::install_github('rickhelmus/patRoondDataIMS@r')'</code>
patRoondExt		no	no	<code>'remotes::install_github('rickhelmus/patRoondExt')'</code>
patRoond		no	yes	<code>'remotes::install_github('rickhelmus/patRoond@master')'</code>

```
install.packages("remotes") # execute this if remotes (or devtools) is not yet installed
remotes::install_github("rickhelmus/patRoondData")
```

2.3.3 Verifying the installation

After the installation is completed, you may need to restart R. Afterwards, the `verifyDependencies()` function can be used to see if `patRoond` can find all its dependencies:

```
patRoond::verifyDependencies()
```

2.4 Managing legacy installations

Previous `patRoond` versions (<2.3) could be installed via an installation script. This script is now deprecated and replaced by the previously discussed installation methods. If you used this script in the past, and would like to update `patRoond`, it is important to first disable or fully remove the legacy installation. This is easily accomplished by the `patRoondInst` package that was discussed before:

```
patRoondInst::toggleLegacy(FALSE) # disable legacy installation
patRoondInst::removeLegacy() # remove all files part of the legacy installation
patRoondInst::removeLegacy(restoreRProfile = TRUE) # as above, and remove any automatic
↪ changes that were made in ~/.Rprofile
```

NOTE Restart R afterwards to ensure all changes are in effect.

For more details, please refer to the reference manual (`?patRoondInst::legacy`).

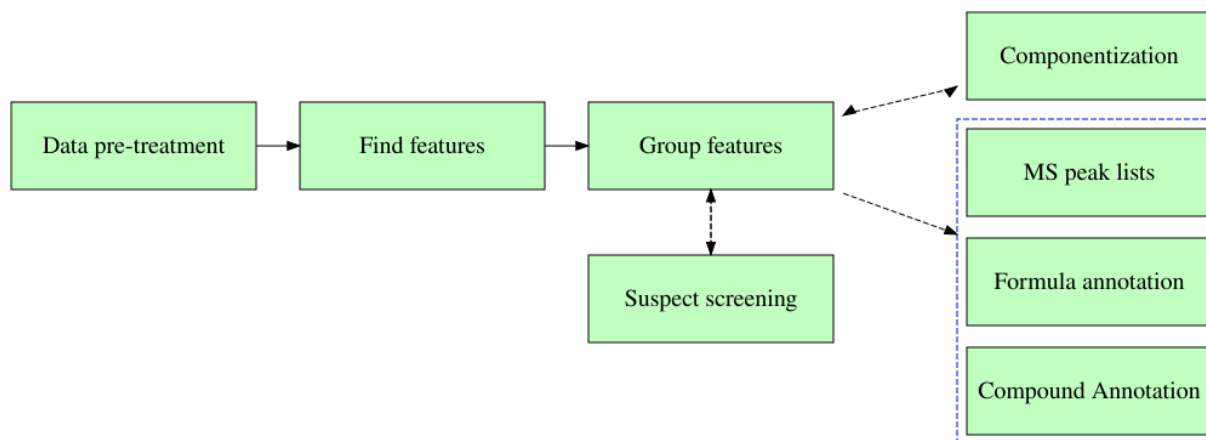
2.5 Developers: devtools / pkgload

If you are a developer and using the `load_all()` function from `devtools` or `pkgload` to load `patRoön`, it is highly recommended to disable debug compilation flags as this considerably degrades performance:

```
# disable debug flags via package options
options(pkg.build_extra_flags = FALSE)
# disable debug flags via environment variable, may be more reliable with e.g. testthat
Sys.setenv(PKG_BUILD_EXTRA_FLAGS = "false")
```

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 `patRoön` 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 description 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 `patRoön` 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 `patRoön` 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 **patRoön** 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.

The feature group data may then be subjected to **componentization**. A **component** is defined as a collection of multiple feature groups that are somehow related to each other. Typical examples 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 homologous series and features that display a similar intensity trend across samples. If adducts or isotopes were annotated during componentization then this data may be used to prioritize the feature groups.

The last step in the workflow commonly 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.

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.
- **Componentization** involves consolidating different feature groups that have a relationship to each other in to a single component.
- **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.

The next chapters will discuss how to generate this data and process it. Afterwards, several advanced topics are discussed such as combining positive and negative ionization data, screening for transformation products, the use of ion mobility data in workflows and other advanced functionality.

4 Generating workflow data

4.1 Introduction

4.1.1 Workflow functions

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 **patRoön** 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	<code>convertMSFiles()</code> , <code>recalibrateDAFiles()</code>	-
Finding features	<code>findFeatures()</code>	<code>features</code>
Grouping features	<code>groupFeatures()</code>	<code>featureGroups</code>
Suspect screening	<code>screenSuspects()</code>	<code>featureGroupsScreening</code>
Componentization	<code>generateComponents()</code>	<code>components</code>
MS peak lists	<code>generateMSPeakLists()</code>	<code>MSPeakLists</code>
Formula annotation	<code>generateFormulas()</code>	<code>formulas</code>
Compound annotation	<code>generateCompounds()</code>	<code>compounds</code>

4.1.2 Workflow output

The output of each workflow step is stored in objects derived from so called S4 classes. Knowing the details about the S4 class system of R is generally not important when using `patRoan` (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.
```

4.1.3 Overview of workflow functions and their output

Below is an overview of the common workflow function, their output and how they interact. Note that sets workflows, IMS workflows and other advanced functionality is not shown here.

The next sections in this chapter will further detail on how to actually perform the non-target workflow steps to generate data. The transformation product screening workflows are discussed in a separate chapter.

4.2 Data processing projects

While not strictly necessary, it is recommended to perform each non-target data processing analysis in a *project*. In `patRoan`, these projects typically contain one or more R script files, configuration files and tables with the analysis information.

To start a new project, or to simply explore the capabilities of `patRoan`, it is easiest to run the `newProject()` function within RStudio. This function launches 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,

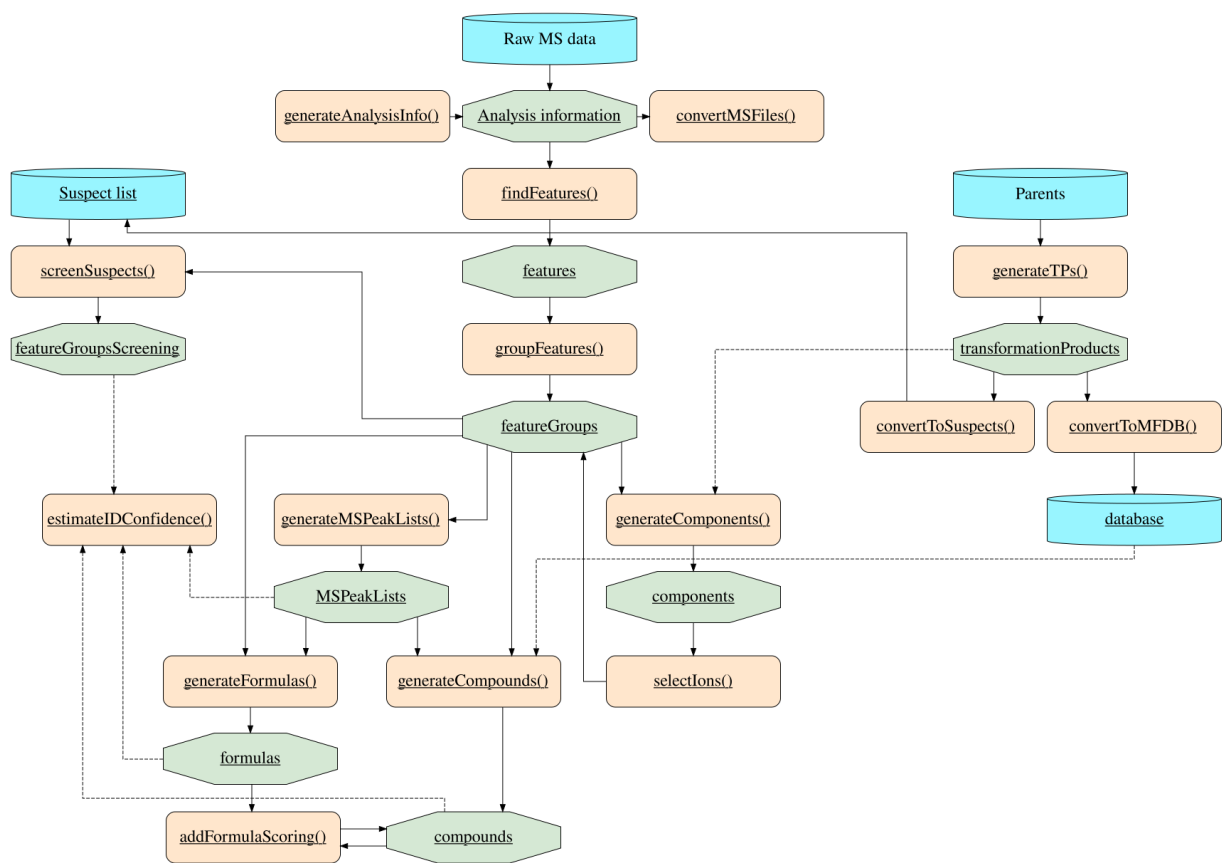


Figure 1: **Workflow functions and output classes.**

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 makes it easier to re-open the project in future data processing sessions.

NOTE At the moment `newProject()` requires RStudio.

The next sections further describe the steps needed to setup the sample analysis and how to perform the workflow. This information can be used to manually create projects or serve as an explanation to what `newProject()` performs automatically for you.

4.3 Sample analyses

In `patRoon` a *sample analysis*, or *analysis*, refers to a single HRMS measurement of a sample. The raw data for an analysis is typically stored in different file types and file formats, which are discussed in the next section. The analysis information informs `patRoon` which analyses should be processed, where to find the raw data and is used to store any other metadata. The data pre-treatment section describes how to convert and prepare the raw data.

4.3.1 Analysis file types and formats and the `msdata` interface

In `patRoon` a distinction is made between four types of raw data files:

- **raw**: the original raw data files from the HRMS instrument, with formats such as **.raw** (Thermo, Waters) or **.d** (Bruker, Agilent).
- **centroid**: exported and centroided data files in the **mzML** or **mzXML** format.
- **profile**: exported but not centroided (i.e. profile) data files in **mzML** or **mzXML** formats.
- **ims**: exported ion mobility HRMS data files in the **mzML** format.

Unfortunately, algorithms within the workflow may require different file types/formats, and it is often necessary to convert raw data to one or more other file types/formats. However, for ‘classical’ (non_ims) workflows it is often sufficient to convert **raw** data to **centroid** data in **mzML** format. In **patRoön** the choice of file type and format is primarily based on:

1. The feature detection algorithm that is used. An overview of requirements is listed in the feature detection section.
2. The internal code of **patRoön** to process raw data. This is called the **msdata** interface.

The **msdata** interface is used throughout many operations within **patRoön**, such as loading mass spectra for feature annotation and generating extracted ion chromatograms (EICs) for plotting and reporting data. The **msdata** interface itself supports different backends, each of which support different file types and formats. By default the most suitable backend is chosen automatically, depending on the available raw data and which backends are available on your system. The currently supported backends are:

Backend	Supported file types and formats
"opentims"	Uses OpenTIMS for highly efficient reading of raw Bruker TIMS data (only available on Windows and Linux). Requires the Bruker TDF-SDK, see the Installation chapter.
"mzr"	uses mzR to read centroid files in the mzML and mzXML formats. This was always used before patRoön 3.0.
"mstoolkit"	Uses mstoolkit to read ims files (mzML format) and centroid files in the mzML and mzXML formats. Requires the Rmstoolkitlib R package (see the Installation chapter).
"streamcraft"	Uses StreamCraft to read ims files (mzML format) and centroid files in the mzML and mzXML formats.

NOTE The **piek** feature detection algorithm uses the **msdata** interface directly and therefore supports a wide range of raw data file types and formats.

See the reference manual for more details on the **msdata** interface (?**msdata**).

4.3.2 Analysis information

In **patRoön**, the *analysis information* describes the analyses that are to be processed, where they are located and holds any metadata such as replicate information. The analysis information should be a **data.frame** and is often stored in a variable called **anaInfo** (of course you are free to choose a different name!).

The analysis information table has a few mandatory columns:

- **path_raw,path_centroid,path_profile,path_ims**: the directory path to the analyses in the raw, centroided, profile and **ims** formats, respectively. See the previous section for details on the file types. Leave empty if the file type is not present.
- **analysis**: the name of the analysis. This should be the file name *without* file extension and *without* directory path (e.g. **C:\MyAnalysis\sample1.d** becomes **sample1**). Each value in the **analysis** column must be unique.
- **replicate**: to which *replicate* the analysis belongs. The analysis which are replicates of each other get the same name.

- **blank**: which replicate should be used for blank subtraction. Can be left empty if no subtraction is desired.

If a workflow requires multiple file formats of a same file type, e.g. centroided `mzML` and `mzXML` files, then simply store both file formats in the directory specified in the `path_XXX` column. If data needs to be exported (discussed in the next section), simply assign its *destination path* to the respective `path_XXX` column.

The analysis information table can be manually constructed in R (e.g. through import of an CSV file), through a graphical interface with `newProject()` (discussed previously) or automatically by the `generateAnalysisInfo()` function. Here is an example of the latter for the example data in the `patRoonaData` package:

```
# Take example data from patRoonaData package (triplicate solvent blank + triplicate
→ standard)
generateAnalysisInfo(fromCentroid = patRoonaData::exampleDataPath(),
  replicate = c(rep("solvent-pos", 3), rep("standard-pos", 3)),
  blank = "solvent-pos")
```

```
#>      analysis                                     path_centroid path_raw path_profile path_im
#> 1 solvent-pos-1 /usr/local/lib/R/site-library/patRoonaData/extdata/pos
#> 2 solvent-pos-2 /usr/local/lib/R/site-library/patRoonaData/extdata/pos
#> 3 solvent-pos-3 /usr/local/lib/R/site-library/patRoonaData/extdata/pos
#> 4 standard-pos-1 /usr/local/lib/R/site-library/patRoonaData/extdata/pos
#> 5 standard-pos-2 /usr/local/lib/R/site-library/patRoonaData/extdata/pos
#> 6 standard-pos-3 /usr/local/lib/R/site-library/patRoonaData/extdata/pos
```

(Note that for the example data the `patRoonaData::exampleAnalysisInfo()` function can also be used.)

It is possible to add more columns to the analysis information: these can be used to attach additional metadata to each sample analysis. These columns can be added later to the table, or specified directly to `generateAnalysisInfo()`:

```
# As above, but add some (nonsensical) metadata: location and exposure
generateAnalysisInfo(fromCentroid = patRoonaData::exampleDataPath(),
  replicate = c(rep("solvent-pos", 3), rep("standard-pos", 3)),
  blank = "solvent-pos",
  location = c("NL", "NL", "NL", "DE", "DE", "DE"),
  exposure = c(0, 0, 0, 2, 2, 2))
```

```
#>      analysis                                     path_centroid path_raw path_profile path_im
#> 1 solvent-pos-1 /usr/local/lib/R/site-library/patRoonaData/extdata/pos
#> 2 solvent-pos-2 /usr/local/lib/R/site-library/patRoonaData/extdata/pos
#> 3 solvent-pos-3 /usr/local/lib/R/site-library/patRoonaData/extdata/pos
#> 4 standard-pos-1 /usr/local/lib/R/site-library/patRoonaData/extdata/pos
#> 5 standard-pos-2 /usr/local/lib/R/site-library/patRoonaData/extdata/pos
#> 6 standard-pos-3 /usr/local/lib/R/site-library/patRoonaData/extdata/pos
```

The metadata (`location` and `exposure` in the example above) can be used in various ways later in the workflow to process the non-target data.

See the reference manual for more details on the analysis information and `generateAnalysisInfo()` (`?analysis-information`).

4.3.3 Data conversion and pre-treatment

As noted in the previous sections, analyses are typically stored in different file types and formats, and algorithms in the workflow typically only support some of these. Hence, it is often required to perform file conversion.

The `convertMSFiles()` function supports various algorithms to perform the necessary file conversions:

Algorithm	Usage	Input file types and formats	Output file types and formats	Remarks
ProteoWizard	<code>convertMSFiles(algorithm = "pwiz", ...)</code>	all formats and types	all except raw	most popular and versatile converter
OpenMS	<code>convertMSFiles(algorithm = "openms", ...)</code>	centroid and profile (mzML and mzXML)	centroid and profile (mzML and mzXML)	Does not support centroiding.
DataAnalysis	<code>convertMSFiles(algorithm = "bruker", ...)</code>	raw (Bruker .d)	centroid and profile (mzML and mzXML)	
IMS collapse	<code>convertMSFiles(algorithm = "imscollapse", ...)</code>	raw (Bruker TIMS) and ims (mzML) (uses msdata)	centroid (mzML and mzXML)	Omits MS2 data by default.
TIMSCONVERT	<code>convertMSFiles(algorithm = "timsconvert", ...)</code>	raw (Bruker TIMS)	centroid, profile, ims (mzML)	

NOTE For the conversion of IMS to centroided data it is highly recommended to use the IMS collapse or TIMSCONVERT algorithms, as ProteoWizard currently does not support accurate centroiding of IMS data. For the conversion of Agilent IMS data, ProteoWizard can be used to convert the raw .d files to the ims (mzML) files, and subsequently IMS collapse can be used to convert these to centroided files.

The `convertMSFiles()` function uses the analysis information to locate the input files and the destination paths for the output. The `path_XXX` columns should contain the desired destination directories for those file types that should be exported. For instance:

```
anaInfoConv <- data.frame(
  analysis = c("sample1", "sample2"),
  replicate = "replicate",
  blank = "",
  path_raw = "raw_files", # directory containing the raw HRMS instrument files (.d,
  ↪ .raw, ...)
  path_centroid = "centroid_files" # destination directory where the centroided files
  ↪ will be placed
)
anaInfoConv
```

```
#>   analysis replicate blank path_raw path_centroid
#> 1 sample1 replicate   raw_files centroid_files
#> 2 sample2 replicate   raw_files centroid_files
```

The `convertMSFiles()` takes the analysis information and performs the necessary conversions:

```
# Convert thermo raw files to centroided mzML files
convertMSFiles(anaInfo, typeFrom = "raw", formatFrom = "thermo", typeTo = "centroid",
  ↪   formatTo = "mzML",
      algorithm = "pwiz")

# convert TIMS data to LC-MS like centroided data
convertMSFiles(anaInfo, typeFrom = "raw", formatFrom = "bruker_ims", typeTo = "centroid",
  ↪   formatTo = "mzML",
      algorithm = "timsconvert")

# convert Agilent IMS-HRMS data to ims data in mzML format
convertMSFiles(anaInfo, typeFrom = "raw", formatFrom = "agilent_ims", typeTo = "ims",
  ↪   formatTo = "mzML",
      algorithm = "pwiz")

# ... and then use IM collapse to LC-MS like centroided mzML files
convertMSFiles(anaInfo, typeFrom = "ims", formatFrom = "mzML", typeTo = "centroid",
  ↪   formatTo = "mzML",
      algorithm = "imscollapse")
```

The `newProject()` utility can automatically generate a proper analysis information table and the required code to perform the desired file conversions.

NOTE The IMS collapse algorithm omits MS/MS data by default to save space and speed up file conversion. This algorithm is typically used in post mobility assignment IMS workflows, which do not use MS/MS data from centroided files. Set `includeMSMS=TRUE` to include MS/MS data.

Besides conversion, other types of data pre-treatment may also need to be performed. For instance, ProteoWizard can be used to apply various data filters, and several utility functions exist to apply mass re-calibration of Bruker data files.

```
# Use ProteoWizard to perform conversion and apply a filter to only keep MS 1 data
# See http://proteowizard.sourceforge.net/tools/msconvert.html for supported filters
convertMSFiles(anaInfo, typeFrom = "raw", formatFrom = "thermo", typeTo = "centroid",
  ↪   formatTo = "mzML",
      algorithm = "pwiz", filters = "msLevel 1")

# perform m/z re-calibration of Bruker data (should be performed prior to file
  ↪ conversion!)
# NOTE: this requires Bruker DataAnalysis
setDAMethod(anaInfo, "path/to/DAMethod.m") # configure Bruker files with given method
  ↪ that has automatic calibration configured
recalibrateDAFiles(anaInfo) # trigger re-calibration for each analysis
getDACalibrationError(anaInfo) # get calibration error for each analysis
```

Please see the reference manual for more details (`?convertMSFiles`, `?`bruker-utils``).

4.4 Example data

The `patRoonData` and `patRoonDataIMS` packages contain HRMS and IMS-HRMS data, respectively, that can be used to explore the workflow. The data files are already exported in `.mzML` format and therefore

ready to use. Furthermore, these packages also contain example suspect and internal standard lists, which will be explained in the next sections. The following functions can be used to obtain the example data:

```
# get analysis information for example data, which can be used directly for feature
↪ detection
patRoonaData::exampleAnalysisInfo("positive")[, c("analysis", "replicate", "blank",
↪ "path_centroid")]
```

```
#>      analysis      replicate      blank      path_centroid
#> 1 solvent-pos-1 solvent-pos solvent-pos /usr/local/lib/R/site-library/patRoonaData/extdata/pos
#> 2 solvent-pos-2 solvent-pos solvent-pos /usr/local/lib/R/site-library/patRoonaData/extdata/pos
#> 3 solvent-pos-3 solvent-pos solvent-pos /usr/local/lib/R/site-library/patRoonaData/extdata/pos
#> 4 standard-pos-1 standard-pos solvent-pos /usr/local/lib/R/site-library/patRoonaData/extdata/pos
#> 5 standard-pos-2 standard-pos solvent-pos /usr/local/lib/R/site-library/patRoonaData/extdata/pos
#> 6 standard-pos-3 standard-pos solvent-pos /usr/local/lib/R/site-library/patRoonaData/extdata/pos
```

```
# negative ionization IMS data
patRoonaDataIMS::exampleAnalysisInfo("negative")[, c("analysis", "replicate", "blank",
↪ "path_ims", "path_centroid")]
```

```
#>      analysis      replicate      blank      path_ims      path_centroid
#> 1 blank-neg-1 blank-neg blank-neg /usr/local/lib/R/site-library/patRoonaDataIMS/extdata/negative
#> 2 blank-neg-2 blank-neg blank-neg /usr/local/lib/R/site-library/patRoonaDataIMS/extdata/negative
#> 3 blank-neg-3 blank-neg blank-neg /usr/local/lib/R/site-library/patRoonaDataIMS/extdata/negative
#> 4 standard-neg-1 standard-neg blank-neg /usr/local/lib/R/site-library/patRoonaDataIMS/extdata/negative
#> 5 standard-neg-2 standard-neg blank-neg /usr/local/lib/R/site-library/patRoonaDataIMS/extdata/negative
#> 6 standard-neg-3 standard-neg blank-neg /usr/local/lib/R/site-library/patRoonaDataIMS/extdata/negative
```

```
# example suspect list
patRoonaData::suspectsPos[1:5, ]
```

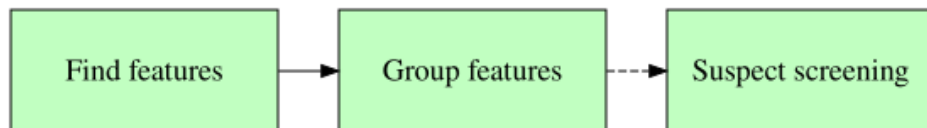
```
#>      name      mz      rt      SMILES
#> 1 DEET 192.1383 355.8 CCN(CC)C(=O)c1cccc(c1)C
#> 2 Diglyme 135.1016 252.0 COCCOCCOC
#> 3 Dimethametryn 256.1590 372.6 CCNc1nc(nc(n1)SC)N[C@@H](C)C(C)C
#> 4 Irgarol 254.1434 370.8 CC(C)(C)Nc1nc(nc(n1)SC)NC2CC2
#> 5 Prometryne 242.1434 364.8 CC(C)NC1=NC(=NC(=N1)SC)NC(C)C
```

```
# example internal standards
patRoonaDataIMS::ISTDListPos[1:5, ]
```

```
#>      name      mz      rt      formula SMILES mobility_[M+H]+ InChI InChIKey neutral
#> 1 1H-benzotriazole-D4 124.0807 248.10 C6H1[2]H4N3 <NA> 0.544 <NA> <NA> 123.
#> 2 Atenolol-D7 274.2143 199.34 C14H15[2]H7N2O3 <NA> 0.756;1.017 <NA> <NA> 273.
#> 3 Atrazine-D5 221.1324 335.87 C8H9[2]H5C1N5 <NA> 0.706;1.081 <NA> <NA> 220.
#> 4 Bezafibrate-D6 368.1530 358.16 C19[2]H6H14C1N04 <NA> 0.857 <NA> <NA> 367.
#> 5 Caffeine-13C3 198.0977 288.04 [13]C3C5H10N4O2 <NA> 0.655 <NA> <NA> 197.
```

4.5 Features

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



4.5.1 Finding features

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

Algorithm	Usage	Input	Remarks
OpenMS	<code>findFeatures(algorithm = "openms", ...)</code>	centroid (mzML)	Uses FeatureFinderMetabo
XCMS	<code>findFeatures(algorithm = "xcms", ...)</code>	centroid (mzML or mzXML)	Uses <code>xcms::xcmsSet()</code>
XCMS3	<code>findFeatures(algorithm = "xcms3", ...)</code>	centroid (mzML or mzXML)	Uses <code>xcms::findChromPeaks()</code>
piek	<code>findFeatures(algorithm = "piek", ...)</code>	raw data loaded with msdata	Introduced in patRoan 3.0
enviPick	<code>findFeatures(algorithm = "envipick", ...)</code>	centroid (mzXML)	Uses <code>enviPick::enviPickwrap()</code>
KPIC2	<code>findFeatures(algorithm = "kpic2", ...)</code>	centroid (mzML or mzXML)	
SIRIUS	<code>findFeatures(algorithm = "sirius", ...)</code>	centroid (mzML or mzXML)	
SAFD	<code>findFeatures(algorithm = "safd", ...)</code>	centroid or profile (mzXML)	Experimental
DataAnalysis	<code>findFeatures(algorithm = "bruker", ...)</code>	raw (Bruker .d)	Uses Find Molecular Features from DataAnalysis (Bruker only)

The feature finding algorithms have different requirements on the data file types and formats (*Input* column). See the data conversion section to perform data conversion.

When using the XCMS algorithms both the ‘classical’ interface and latest XCMS3 interfaces are supported. The implementation and functionality is largely the same within patRoan, however, the latter is recommended as the classical interface is not further developed anymore.

Most often the performance of these algorithms heavily depends 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 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 <code>?findFeaturesOpenMS</code>)
XCMS / XCMS3	peakwidth, mzdiff, prefilter, noise (assuming default <code>centWave</code> algorithm, see <code>?findPeaks.centWave</code> / <code>?CentWaveParam</code>)
piek	genEICParams, peakParams, see next section and <code>?findFeaturesPiek</code>
enviPick	dmzgap, dmzdens, drtgap, drtsmall, drtdens, drtfill, drttotal, minpeak, minint, maxint (see <code>?enviPickwrap</code>)
KPIC2	kmeans, level, min_snr (see <code>?findFeatures</code> and <code>?getPIC</code> / <code>?getPIC.kmeans</code>)
SIRIUS	The sirius algorithm is currently parameterless
SAFD	mzRange, maxNumIter, resolution, minInt (see <code>?findFeaturesSAFD</code>)
DataAnalysis	See <i>Find -> Parameters... -> Molecular Features</i> in DataAnalysis.

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

Some examples of finding 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
fListXCMS3 <- findFeatures(anaInfo, "xcms3", CentWaveParam(peakwidth = c(5, 15))) # XCMS3
fListEP <- findFeatures(anaInfo, "envipick", minint = 1E3) # enviPick
fListKPIC2 <- findFeatures(anaInfo, "kplic2", kmeans = TRUE, level = 1E4) # KPIC2
fListSIRIUS <- findFeatures(anaInfo, "sirius") # SIRIUS
```

4.5.1.1 Feature detection with piek The `piek` algorithm was introduced in `patRoan` 3.0, and extends the work done by Dietrich et al. (2021). This algorithm finds features from extracted ion chromatograms (EICs) and can use various peak detection algorithms to find features. The EICs are generated from fixed-width m/z bins, therefore, it is important to define the proper range for the compounds of interest. The feature detection can be speed up by filtering the EIC bins from one of the following:

- data from a suspect list (the format follows that of suspect lists for suspect screening workflows)
- data from precursors detected in data-dependent MS/MS experiments (DDA)

This will considerably limit the number of EICs that need to be generated and processed.

The `getPiekEICParams()` function is used to create the parameters that are used to generate the EICs. Its output is passed to the `genEICParams` argument to `findFeatures()` function. Similarly, the `getDefPeakParams()` function is used to create the parameters that are used to detect peaks, and its output is passed to the `peakParams` argument of `findFeatures()`.

Some examples are shown below:

```
# find features from m/z bins with piek's native peak detection algorithm
genEICParams <- getPiekEICParams()
peakParams <- getDefPeakParams("chrom", "piek")
```

```
fList <- findFeatures(anaInfo, "piek", genEICParams = genEICParams, peakParams =
  ↳ peakParams)

# as above, but customize binning
genEICParams <- getPiekEICParams(mzRange = c(50, 1000), mzStep = 0.01)
peakParams <- getDefPeakParams("chrom", "piek")
fList <- findFeatures(anaInfo, "piek", genEICParams = genEICParams, peakParams =
  ↳ peakParams)

# find features from a suspect list with XCMS peak detection
# see the suspect screening section for more details on the suspect list format
genEICParams <- getPiekEICParams(filter = "suspects")
peakParams <- getDefPeakParams("chrom", "xcms3")
fList <- findFeatures(anaInfo, "piek", genEICParams = genEICParams, peakParams =
  ↳ peakParams,
                      suspects = suspList, adduct = "[M+H]+")

# use DDA MS/MS data to find features
# only find features with data between two and ten minutes retention time.
# increase minimum spectrum TIC to focus on features with high intensity MS/MS data
genEICParams <- getPiekEICParams(filter = "ms2", retRange = c(120, 600), minTIC = 1E5)
peakParams <- getDefPeakParams("chrom", "openms")
fList <- findFeatures(anaInfo, "piek", genEICParams = genEICParams, peakParams =
  ↳ peakParams)
```

For more details, please see the reference manual (`?findFeaturesPiek`).

4.5.2 Feature grouping

For grouping features across analyses several algorithms are supported.

Algorithm	Usage	Remarks
OpenMS	<code>groupFeatures(algorithm = "openms", ...)</code>	Uses the FeatureLinkerUnlabeled algorithm (and MapAlignerPoseClustering for retention alignment)
XCMS	<code>groupFeatures(algorithm = "xcms", ...)</code>	Uses <code>xcms::group()</code> and <code>xcms::retcor()</code> functions
XCMS3	<code>groupFeatures(algorithm = "xcms3", ...)</code>	Uses <code>xcms::groupChromPeaks()</code> and <code>xcms::adjustRtime()</code> functions
KPIC2	<code>groupFeatures(algorithm = "kplic2", ...)</code>	Uses the KPIC2 package
greedy	<code>groupFeatures(algorithm = "greedy", ...)</code>	Introduced in <code>patRoan</code> 3.0
SIRIUS	<code>groupFeatures(anaInfo, algorithm = "sirius")</code>	Finds <i>and</i> groups features with SIRIUS

NOTE: Grouping features with the `sirius` algorithm will perform both finding and grouping features with SIRIUS. This algorithm cannot work with features from another algorithm.

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. The algorithm used to group features does not have to match to the algorithm that was used to find the features.

Some examples of grouping features are shown below.

```
# Group features, using the fList objects created in the previous feature finding section
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$replicate,
    minFraction = 0))
# group with KPIC2 and set some custom grouping/aligning parameters
fGroupsKPIC2 <- groupFeatures(fListKPIC2, "kpic2", groupArgs = list(tolerance = c(0.002,
  ↪ 18)),
  alignArgs = list(move = "loess"))
# greedy algorithm with custom tolerances and weights
fGroupsGreedy <- groupFeatures(fListXCMS3, "greedy", rtWindow = 5, mzWindow = 0.003,
  scoreWeights = c(retention = 0.5, mz = 3, mobility = 1,
  ↪ intensity = 1))
fGroupsSIRIUS <- groupFeatures(analInfo, "sirius") # find/group features with SIRIUS
```

4.5.3 Suspect screening

After features have been grouped a so called *suspect screening* step may be performed. During this step a *suspect list* is used to screen the detected features and match them to the suspects in the list. Suspect screening can simplify the identification of unknown features, and can simplify the overall workflow by removing the features without matches. The `screenSuspects()` function is used for this purpose, for instance:

```
# Perform a very basic suspect screening workflow. The suspects are matched by m/z
  ↪ values, and any non-hits are removed.
suspects <- data.frame(name = c("1H-benzotriazole", "N-Phenyl urea",
  ↪ "2-Hydroxyquinoline"),
  mz = c(120.0556, 137.0709, 146.0600))
fGroupsSusp <- screenSuspects(fGroups, suspects, onlyHits = TRUE)
```

4.5.3.1 Suspect list format The example above has a very simple suspect list with just three compounds. The format of the suspect list is quite flexible, and can contain the following columns:

- **name**: The name of the suspect. Mandatory and should be unique and file-name compatible (if not, the name will be automatically re-named to make it compatible).
- **rt**: The retention time in seconds. Optional. If specified any feature groups with a different retention time will not be considered to match suspects.
- **mz**, **SMILES**, **InChI**, **formula**, **neutralMass**: *at least* one of these columns must hold data for each suspect row. The **mz** column specifies the ionized mass of the suspect. If this is not available then data from any of the other columns is used to determine the suspect mass.
- **adduct**: The adduct of the suspect. Optional. Set this if you are sure that a suspect should be matched by a particular adduct ion and no data in the **mz** column is available.
- **fragments_mz** and **fragments_formula**: optional columns that may assist ID confidence estimation.
- **mobility** and **CCS** columns: these are for IMS workflows and can increase the confidence of a suspect match. This is discussed here.

In most cases a suspect list is best made as a `csv` file which can then be imported with e.g. the `read.csv()` function. This is exactly what happens when you specify a suspect list when using the `newProject()` function.

Quite often, the ionized masses are not readily available and these have to be calculated. In this case, data in any of the `SMILES`/`InChI`/`formula`/`neutralMass` columns should be provided. Whenever possible, it is *strongly* recommended to fill in `SMILES` column (or `InChI`), as this will assist ID confidence estimation. Applying this to the above example:

```
suspects <- data.frame(name = c("1H-benzotriazole", "N-Phenyl urea",  
  ↪ "2-Hydroxyquinoline"),  
                      SMILES = c("[nH]1nnc2ccccc12", "NC(=O)Nc1ccccc1",  
  ↪ "Oc1ccc2ccccc2n1"))  
fGroupsSusp <- screenSuspects(fGroups, suspects, adduct = "[M+H]+", onlyHits = TRUE)
```

NOTE: It is highly recommended to install OpenBabel to automatically validate and amend chemical properties such as `SMILES`, `InChI`, formulae etc in the suspect list.

Since suspect matching now occurs by the neutral mass it is required that the adduct information for the feature groups are set. This is done either by setting the `adduct` function argument to `screenSuspects` or by feature group adduct annotations.

Finally, when the adduct is known for a suspect it can be specified in the suspect list:

```
# Aldicarb is measured with a sodium adduct.  
suspects <- data.frame(name = c("1H-benzotriazole", "N-Phenyl urea", "Aldicarb"),  
                      SMILES = c("[nH]1nnc2ccccc12", "NC(=O)Nc1ccccc1",  
  ↪ "CC(C)(C=NOC(=O)NC)SC"),  
                      adduct = c("[M+H]+", "[M+H]+", "[M+Na]+"))  
fGroupsSusp <- screenSuspects(fGroups, suspects, onlyHits = TRUE)
```

To summarize:

- If a suspect has data in the `mz` column it will be directly matched with the m/z value of a feature group.
- Otherwise, if the suspect has data in the `adduct` column, the m/z value for the suspect is calculated from its neutral mass and the adduct and then matched with the m/z of a feature group.
- Otherwise, suspects and feature groups are matched by their neutral mass.

The `fragments_mz` and `fragments_formula` columns in the suspect list can be used to specify known fragments for a suspect, which can help ID confidence estimation for suspects. The former specifies the ionized m/z of known MS/MS peaks, whereas the second specifies known formulas. Multiple values can be given by separating them with a semicolon:

```
suspects <- data.frame(name = c("1H-benzotriazole", "N-Phenyl urea",  
  ↪ "2-Hydroxyquinoline"),  
                      SMILES = c("[nH]1nnc2ccccc12", "NC(=O)Nc1ccccc1",  
  ↪ "Oc1ccc2ccccc2n1"),  
                      fragments_formula = c("C6H6N", "C6H8N;C7H6NO", ""),  
                      fragments_mz = c("", "", "118.0652"))
```

4.5.3.2 Removing feature groups without hits Note that any feature groups that were not matched to a suspect are *not* removed by default. If you want to remove these, you can use the `onlyHits` parameter:

```
fGroupsSusp <- screenSuspects(fGroups, suspects, onlyHits = TRUE) # remove any non-hits
↪ immediately
```

The advantage of removing non-hits is that it may significantly reduce the complexity of your dataset. On the other hand, retaining all features allows you to mix a full non-target analysis with a suspect screening workflow. The `filter()` function (discussed here) can also be used to remove feature groups without a hit at a later stage.

4.5.3.3 Combining screening results The `amend` function argument to `screenSuspects` can be used to combine screening results from different suspect lists.

```
fGroupsSusp <- screenSuspects(fGroups, suspects)
fGroupsSusp <- screenSuspects(fGroupsSusp, suspects2, onlyHits = TRUE, amend = TRUE)
```

In this example the suspect lists defined in `suspects` and `suspects2` are both used for screening. By setting `amend=TRUE` the original screening results (i.e. from `suspects`) are preserved. Note that `onlyHits` should only be set in the final call to `screenSuspects` to ensure that all feature groups are screened.

4.5.4 Importing feature data

The `importFeatures()` and `importFeatureGroups()` functions can be used to import existing feature data. This is useful when you have already performed feature finding and/or grouping with another software or package, or when you want to use features from a previous workflow run.

The most important input types are:

Input type	Remarks
"xcms"	Imports an <code>xcmsSet</code> object from XCMS.
"xcms3"	Imports an <code>XCMSnExp</code> object from XCMS.
"kpic2"	Imports a <code>PIC set</code> object from KPIC2.
"table"	Imports data from a table (e.g. <code>data.frame</code> or <code>.csv</code> file).

The `table` input type allows the import of data from any external feature detection algorithm, and follows the the format of the `as.data.table()` function introduced later.

A workflow where data is first exported and later re-imported can be useful to modify feature data outside `patRoan`. For instance, the export functionality can be used to export XCMS feature data, process it with XCMS and then re-import it with `importFeatures()` or `importFeatureGroups` to continue the `patRoan` workflow. Similarly, the `as.data.table()` function can be used to export feature data to a `data.table`, which can then be modified and re-imported.

Some examples are shown below:

```
fListXCMSImp <- importFeatures(xset, "xcms", anaInfo) # import XCMS xcmsSet object
fGroupsXCMS3 <- importFeatureGroups(xs, "xcms3", anaInfo) # import XCMSnExp object with
↪ grouped features

# export, modify and re-import features
fListTab <- as.data.table(fList)
fListTab[, mz := mz * 1.0001] # modify the m/z values
fList <- importFeatures(fListTab, "table", anaInfo) # re-import the modified features
```

See the reference manual for more details, especially for tabular data import (`?importFeatures`, `?importFeatureGroups`).

4.6 Componentization

In `patRoan` *componentization* refers to grouping related feature groups together in components. There are 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.
- MS/MS similarity: feature groups with similar MS/MS spectra are clustered.
- Transformation products: Components are formed by grouping feature groups that have a parent/transformation product relationship. This is further discussed in its own chapter.

The following algorithms are currently supported:

Algorithm	Usage	Remarks
CAMERA	<code>generateComponents(algorithm = "camera", ...)</code>	Clusters feature groups with similar chromatographic elution profiles and annotate by known chemical rules (adducts, isotopologues, in-source fragments).
RAMClustr	<code>generateComponents(algorithm = "ramclustr", ...)</code>	As above.
cliqueMS	<code>generateComponents(algorithm = "cliqueMS", ...)</code>	As above, but using <i>feature components</i> .
OpenMS	<code>generateComponents(algorithm = "openMS", ...)</code>	As above. Uses MetaboliteAdductDecharger.
nontarget	<code>generateComponents(algorithm = "nontarget", ...)</code>	Uses the nontarget R package to perform unsupervised homologous series detection.
Intensity clustering	<code>generateComponents(algorithm = "intclust", ...)</code>	Groups features with similar intensity profiles across analyses by hierarchical clustering.
MS/MS clustering	<code>generateComponents(algorithm = "specclust", ...)</code>	Clusters feature groups with similar MS/MS spectra.
Transformation products	<code>generateComponents(algorithm = "tp", ...)</code>	Discussed in its own chapter.

4.6.1 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 several algorithms can be used to group features that undergo highly similar chromatographic behavior 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.

Note that some 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. For instance, consider the following situation with three feature groups that chromatographically overlap and therefore could be considered a component:

Feature group	<i>m/z</i>	analysis 1	analysis 2	analysis 3
#1	100.08827	Present	Present	Absent
#2	122.07021	Present	Present	Absent
#3	138.04415	Absent	Absent	Present

Based on the mass differences from this example a cluster of $[M+H]^+$, $[M+Na]^+$ and $[M+K]^+$ could be assumed. However, no features of the first two feature groups were detected in the third sample analysis, whereas the third feature group wasn't detected in the first two sample analysis. Based on this it seems unlikely that feature group #3 should be part of the component.

For the algorithms that operate on a 'feature group level' (CAMERA and RAMClustR), the `relMinReplicates` argument can be used to remove feature groups from a component that are not abundant. For instance, when this value is *0.5* (the default), and all the features of a component were detected in four different replicates in total, then only those feature groups are kept for which its features were detected in at least two different replicates (*i.e.* half of four).

Another approach to reduce unlikely adduct annotations is to use algorithms that operate on a 'feature level' (cliqueMS and OpenMS). These algorithms generate components for each sample analysis individually. The 'feature components' are then merged by a consensus approach where unlikely annotations are removed (the algorithm is described further in the reference manual, `?generateComponents`).

Each algorithm supports many different parameters that may significantly influence the (quality of the) output. For instance, care has to be taken to avoid 'over-clustering' of feature groups which do not belong in the same component. This is often easily visible since the chromatographic peaks poorly overlap or are shaped differently. The `checkComponents` function (discussed here) can be used to quickly verify componentization results. For a complete listing all arguments see the reference manual (e.g. `?generateComponents`).

Once the components with adduct and isotopes annotations are generated this data can be used to prioritize and improve the workflow.

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))

# OpenMS with customized parameters
componOpenMS <- generateComponents(fGroups, "openms", ionization = "positive", chargeMax
  ↪ = 2,
                                absMzDev = 0.002)

# cliqueMS with default parameters
componCliqueMS <- generateComponents(fGroups, "cliquems", ionization = "negative")
```

4.6.2 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. 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.

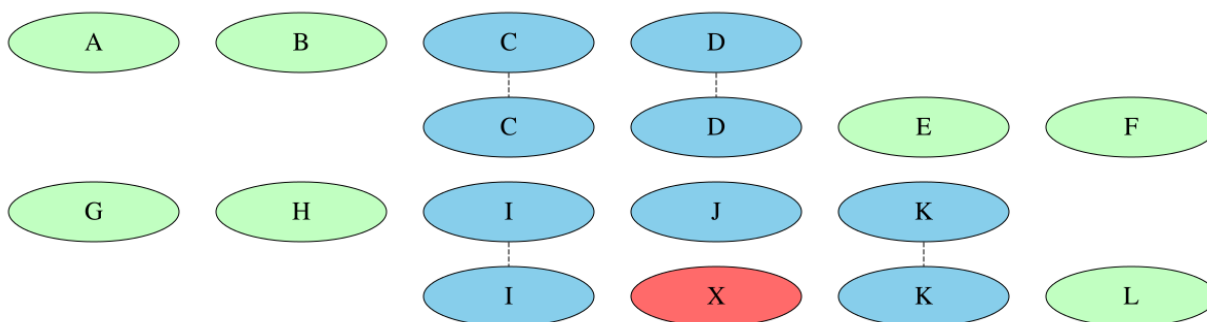


Figure 2: **Linking of homologues series** top: partial overlap and will be linked; bottom: no linkage due to different feature in overlapping series.

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
<code>ionization</code>	Ionization mode: "positive" or "negative". Not needed if adduct annotations are available.
<code>rtRange, mzRange</code>	Retention and m/z increment range. Retention times can be negative to allow series with increasing m/z values and decreasing retention times.
<code>elements</code>	Vector with elements to consider.
<code>rtDev, absMzDev</code>	Maximum retention time and m/z deviation.
<code>...</code>	Further arguments passed to the <code>homol.search()</code> 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))
```

4.6.3 Intensity and MS/MS similarity

The previous componentization methods utilized chemical properties to relate features. The two componentization algorithms described in this section use a statistical approach based on hierarchical clustering. The first algorithm normalizes all feature intensities and then clusters features with similar intensity profiles across sample analyses together. The second algorithm compares all MS/MS spectra from all feature groups, and then uses hierarchical clustering to generate components from feature groups that have a high MS/MS spectrum similarity.

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

Argument	Algorithm	Default	Remarks
<code>method</code>	All	"complete"	Clustering method. See <code>?hclust</code>
<code>metric</code>	<code>intclust</code>	"euclidean"	Metric used to calculate the distance matrix. See <code>?daisy</code> .
<code>normalized</code>	<code>intclust</code>	TRUE	Whether normalized feature intensities should be used. Detailed here.
<code>average</code>	<code>intclust</code>	TRUE	Whether intensities of replicates should first be averaged.
<code>MSPeakLists</code>	<code>specclust</code>	-	The MS peak lists object used for spectral similarity calculations
<code>specSimParams</code>	<code>specclust</code>	<code>getDefaultSpecSimParams()</code>	Parameters used for spectral similarity calculation.
<code>maxTreeHeight</code> , <code>deepSplit</code> , <code>minModuleSize</code>	All	1, TRUE, 1	Used for dynamic cluster assignment. See <code>?cutreeDynamicTree</code> .

The components are generated by automatically assigning clusters using the `dynamicTreeCut` R package. However, the cluster assignment can be performed manually or with different parameters, as is demonstrated below.

The resulting components are stored in an object from the `componentsIntClust` or `componentsSpecClust` S4 class, which are both derived from the `componentsClust` class (which in turn is derived from the `components` class). Several methods are defined that can be used on these 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`, `?componentsSpecClust` and `?componentsClust`).

```
# generate intensity profile 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)

# MS/MS similarity components
componMSMS <- generateComponents(fGroups, "specclust", MSPeakLists = mslists)
```

4.7 Incorporating adduct and isotopic data

With mass spectrometry it is common that multiple m/z values are detected for a single compound. These may be different adducts (e.g. $[M+H]^+$, $[M+Na]^+$, $[M-H]^-$), the different isotopes of the molecule or a combination thereof. When multiple m/z values are measured for the same compound, the feature finding algorithm may yield a distinct feature for each, which adds complexity to the data. In the previous section it was discussed how componentization can help to find feature groups that belong to the same adduct and/or isotope clusters. This section explains how this data can be used to simplify the feature dataset. Furthermore, this section also covers adduct annotations for feature groups which may improve and simplify the general workflow.

4.7.1 Selecting features with preferential adducts/isotopes

The `selectIons` function forms the bridge between feature group and componentization data. This function uses the adduct and isotope annotations to select *preferential* feature groups. For adduct clusters this means that only the feature group that has a preferential adduct (e.g. `[M+H]+`) is kept while others (e.g. `[M+Na]+`) are removed. If none of the adduct annotations are considered preferential, the most intense feature group is kept instead. For isotopic clusters typically only the feature group with the monoisotopic mass (i.e. `M0`) is kept.

The behavior of `selectIons` is configurable with the following parameters:

Argument	Remarks
<code>prefAdduct</code>	The <i>preferential adduct</i> . Usually <code>"[M+H]+"</code> or <code>"[M-H]-"</code> .
<code>onlyMonoIso</code>	If <code>TRUE</code> and a feature group is with isotopic annotations then it is only kept if it is monoisotopic.
<code>chargeMismatch</code>	How charge mismatches between adduct and isotope annotations are dealt with. Valid options are <code>"isotope"</code> , <code>"adduct"</code> , <code>"none"</code> or <code>"ignore"</code> . See the reference manual for <code>selectIons</code> for more details.

In case componentization did not lead to an adduct annotation for a feature group it will never be removed and simply be annotated with the preferential adduct. Similarly, when no isotope annotations are available and `onlyMonoIso=TRUE`, the feature group will not be removed.

Although `selectIons` operates fairly conservative, it is still recommended to verify the componentization results in advance, for instance with the `checkComponents` function discussed here. Furthermore, the next subsection explains how adduct annotations can be corrected manually if needed.

An example usage is shown below.

```
fGroupsSel <- selectIons(fGroups, componCAM, "[M+H]+")
```

```
#> No isotope annotations available!  
#> Removed 21 feature groups detected as unwanted adducts/isotopes  
#> Annotated 13 feature groups with adducts  
#> Remaining 110 feature groups set as default adduct [M+H]+
```

4.7.2 Setting adduct annotations for feature groups

The `adducts()` function can be used to obtain a character vector with adduct annotations for each feature group. When no adduct annotations are available it will simply return an empty character vector.

When the `selectIons` function is used it will automatically add adduct annotations based on the componentization data. In addition, the `adducts()<-` function can be used to manually add or change adduct annotations.

```
adducts(fGroups) # no adduct annotations
```

```
#> character(0)
```

```
adducts(fGroupsSel)[1:5] # adduct annotations set by selectIons()
```

```
#> M109_R192_20 M111_R330_23 M114_R269_25 M116_R317_29 M120_R268_30
#>      "[M+H]+"      "[M+H]+"      "[M+H]+"      "[M+H]+"      "[M+K]+"
```

```
adducts(fGroupsSel)[3] <- "[M+Na]+" # modify annotation
adducts(fGroupsSel)[1:5] # verify
```

```
#> M109_R192_20 M111_R330_23 M114_R269_25 M116_R317_29 M120_R268_30
#>      "[M+H]+"      "[M+H]+"      "[M+Na]+"      "[M+H]+"      "[M+K]+"
```

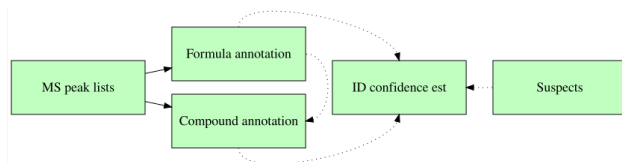
NOTE Adduct annotations are always available with sets workflows.

4.7.3 Using adduct annotations in the workflow

When feature groups have adduct annotations available this may simplify and improve the workflow. The `adduct` and `ionization` arguments used for suspect screening, formula/compound annotation and some componentization algorithms do not have to be set anymore, since this data can be obtained from the adduct annotations. Furthermore, these algorithms may improve their results, since the algorithms are now able to use adduct information for each feature group individually, instead of assuming that all feature groups have the same adduct.

4.8 Annotation

The annotation consists of collecting MS peak lists, formula and/or compound annotations and finally estimation of identification confidence for the candidates (including suspects):



Note that compound annotation is 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. The estimation of ID confidence is optional, and can be performed for candidates from formula annotations, compounds annotations and suspect matches.

4.8.1 MS peak lists

NOTE The `generateMSPeakLists()` function was significantly changed in **patRoan 3.0**, and now uses internal algorithms to retrieve MS and MS/MS spectra. The previous algorithms (`mzr`, `bruker` and `brukerfms`) are now deprecated and should not be used anymore.

The `generateMSPeakLists()` generates MS peak lists for features and feature groups. When it is called it will:

1. Find all MS and MS/MS spectra that ‘belong’ to a feature, i.e. those spectra that were recorded close to the retention time of the feature. For MS/MS data only spectra will be considered with a precursor mass close to that of the feature (unless the data was recorded with data independent acquisition (DIA, MS^E, b/cCID, ...)).
2. Average the MS or MS/MS spectra for each feature to produce the peak lists for features.
3. Average the peak lists for the features within the same group to obtain feature group peak lists.

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 would take a very long time.

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

Argument	Remarks
<code>maxMSRTWindow</code>	Maximum time window +/- the feature retention time (in seconds) to collect spectra for averaging. Higher values may increase processing times.
<code>topMost</code>	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.
<code>avgFeatParams</code> and <code>avgFGroupParams</code>	Parameters that configure the averaging of feature and feature group peak lists, respectively (discussed below).
<code>fixedIsolationWidth</code>	Sets how For MS/MS spectra are selected: see below.

The parameters passed to `avgFeatParams` and `avgFGroupParams` are passed as a list and some typical parameters include:

- `clusterMzWindow`: Maximum m/z window used to cluster mass peaks when averaging. May be lowered for HRMS instruments with very high resolution and accuracy.
- `topMost`: Retain no more than this amount of most intense mass peaks prior to averaging.
- `minIntensityPre` / `minIntensityPost`: Mass peaks below this intensity will be removed before/after averaging.
- `absMinAbundance` / `relMinAbundance`: Minimum absolute or relative abundance of a mass peak to be retained after averaging.

Optimizing the thresholds may be useful to remove noise and speed-up the averaging of spectra.

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, avgFeatParams = avgPListParams, avgFGroupParams =
  ↪ avgPListParams)
```

See `?getDefAvgPListParams` for all possible averaging parameters.

The `fixedIsolationWidth` argument configures the selection of MS/MS spectra for a feature. This is normally performed by comparing the m/z value of the feature with the m/z values from picked up precursors from DDA experiments. Note that current instruments typically have a relatively large isolation window (e.g. 1-2 Da) to select ions for fragmentation. Therefore, all MS/MS spectra with a precursor m/z within this isolation window are considered to belong to the feature. The `fixedIsolationWidth` argument can be used to override this behavior and can be one of the following:

- `FALSE` (default): MS/MS spectra are selected based on the isolation window recorded by the instrument.

- a number (e.g. 1): a fixed value tolerance is used for the precursor selection m/z . This can e.g. be used to limit the spectra selection to precursors with a close m/z to that of the feature value. *NOTE*: it is recommended to allow some tolerance, as the recorded precursor m/z values often slightly deviate from the feature m/z values.
- NA: no precursor m/z matching is performed and all MS/MS spectra recorded close to the retention time of the feature are considered for averaging. This is automatically done for ‘classical’ DIA data without precursor information (i.e. not SWATH).

See `?generateMSPeakLists` for further details.

4.8.2 Formulae

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

Algorithm	Usage	Remarks
GenForm	<code>generateFormulas(algorithm = "genform", ...)</code>	Bundled with <code>patRoan</code> . Reasonable default.
SIRIUS	<code>generateFormulas(algorithm = "sirius", ...)</code>	Requires MS/MS data.
DataAnalysis	<code>generateFormulas(algorithm = "bruker", ...)</code>	Requires FMF features (i.e. <code>findFeatures(algorithm = "bruker", ...)</code>). Uses <i>SmartFormula</i> 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 restrictions. 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, 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 <i>ppm</i>).
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).
featThresholdAnnAll		Minimum relative amount (0-1) that a candidate formula for a feature group should be found among all annotated features (e.g. 1 means that a candidate is only considered if it was assigned to all annotated features).
adduct	All	The adduct to consider for calculation (e.g. "[M+H]+", "[M-H]-", more details in the adduct section). Don't set this when adduct annotations are available.
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, mslists, "genform") # GenForm, default settings
formulasGF2 <- generateFormulas(fGroups, mslists, "genform", calculateFeatures = FALSE) #
  ↳ direct feature group assignment (faster)
formulasSIR <- generateFormulas(fGroups, mslists, "sirius", elements = "CHNOPSClBr") #
  ↳ SIRIUS, common elements for pollutant
formulasSIR2 <- generateFormulas(fGroups, mslists, "sirius", adduct = "[M-H]-") # SIRIUS,
  ↳ negative ionization
formulasBr <- generateFormulas(fGroups, mslists, "bruker", MSMode = "MSMS") # Only
  ↳ consider MSMS data (SmartFormula3D)
```

4.8.3 Compounds

An important step in a typical non-target workflow is structural identification for features of interest, as this information may finally reveal *what* a feature is. In a first step all possible candidate structures for a feature are obtained from a database (based on e.g. monoisotopic mass or formula). These candidates are then ranked, for instance, by matching the feature MS/MS data with in-silico or library MS/MS spectra or its relevance to the environment.

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

Algorithm	Usage	Remarks
MetFrag	<code>generateCompounds(algorithm = "metfrag", ...)</code>	Supports many databases (including offline and custom), matching MS/MS data with in-silico and library MS/MS data, and many other scorings to rank candidates.

Algorithm	Usage	Remarks
SIRIUS with CSI:FingerID	<code>generateCompounds(algorithm = "sirius", ...)</code>	Matches with in-silico MS/MS data, incorporates formula annotations to improve candidate selection.
Library	<code>generateCompounds(algorithm = "library", ...)</code>	Obtains candidates by matching MS/MS data with an offline MS library, <i>e.g.</i> obtained from MassBank.eu or MoNA.

All algorithms rank their candidates by matching MS/MS data with in-silico generated MS/MS data (MetFrag and SIRIUS) and/or experimental MS/MS data from an MS library (MetFrag with MoNA scoring and Library algorithm). The latter may yield better candidates, and the Library algorithm is also generally much faster. However, in-silico annotation is not limited by the availability of experimental MS/MS data.

Compound annotation is often a relative time and resource intensive procedure. For this reason, annotation occurs for each feature group and not individual features. 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.

4.8.3.1 Database selection for MetFrag and SIRIUS Selecting the right database is important for proper candidate assignment. 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
pubchemlite	"metfrag"	A specialized subset of the PubChem database. Often a good default. May need manual installation (see next section).
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. May need manual installation (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.

Database	Algorithm(s)	Remarks
csv, psv, sdf	"metfrag"	Custom database (see next section). CSV example.

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

```
# 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(patRooin.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		FALSE
individualMoNAScore	OfflineIndividualMoNAScore		TRUE
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
RF_SMILES			FALSE
RF_SIRFP			FALSE
LC50_SMILES			FALSE
LC50_SIRFP			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
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	AnnoTypeCount	pubchemlite	TRUE
annotHitCount	AnnotHitCount	pubchemlite	TRUE
libMatch			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",
                             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 be 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.

4.8.3.3 MetFrag error and timeout handling The use of online databases has the drawback that an error may occur, for instance, as a result of a connection error or when the aforementioned maximum number of candidates is reached (`maxCandidatesToStop` argument). By default, the 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.

4.8.3.4 Annotation with the *Library* algorithm To use the *Library* algorithm we first need to load an MS library. Currently, MS libraries in the MSP and MoNA JSON formats are supported. Note that the former format is not so well standardized, and the support in `patRoan` was mainly tailored for MSP files from MassBank.eu and MoNA. To load the MS library the `loadMSLibrary()` function is used:

```
mslibrary <- loadMSLibrary("~/MassBank_NIST.msp", "msp") # MassBank.eu MSP library
mslibrary <- loadMSLibrary("~/MoNA-export-CASMI_2016.msp", "msp") # MoNA MSP library
mslibrary <- loadMSLibrary("~/MoNA-export-MassBank.json", "json") # MoNA JSON library
```

NOTE Currently it is only possible to load formula annotated MS/MS peaks with the MoNA JSON format.

Once loaded, the MS library can be post-processed with various filtering, subsetting and export functionality, which may be useful for more tailored compound annotation. This is further discussed in the advanced chapter.

The compound annotation is performed with `generateCompounds()`:

```
compsLib <- generateCompounds(fGroups, mslists, "library", MSLibrary = mslibrary)

# set minimum MS/MS spectral match for candidates to 0.5
compsLib <- generateCompounds(fGroups, mslists, "library", MSLibrary = mslibrary, minSim
↪ = 0.5)
```

4.8.3.5 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 sensible unless scoring from another formula calculation algorithm is desired.

4.8.3.6 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`).

4.8.4 Estimation of identification confidence

NOTE The `annotateSuspects()` function was replaced by the `estimateIDConfidence()` function in `patRoom 3.0`. This section describes the updated interface.

Full identification of features is often difficult, and often you are left with tentative suspect, formula and compound assignments. A common way to communicate the confidence of these assignments is by using identification levels, as proposed by Schymanski et al. (2014).

The `estimateIDConfidence()` function provides an *estimate* of the identification confidence for candidates assigned to suspects, formulae and compounds. It uses a rule based approach to estimate identification levels:

Level	Description	Rules
1	Target match	Retention time deviates <12 seconds from suspect list. At least 3 (or all if the suspect list contains less) fragments from the suspect list must match.
2a	Good MS/MS library match	Candidate is top ranked in the <code>compounds</code> results. The <code>individualMoNAScore</code> (MetFrag) or <code>libMatch</code> (Library algorithm) is at least 0.9 and no other candidates were matched with the MS library.
3a	Fair library match	The <code>individualMoNAScore</code> or <code>libMatch</code> is at least 0.7.
3b	Known MS/MS match	At least 3 (or all if the suspect list contains less) fragments from the suspect list must match.
3c	Good in-silico MS/MS match	The compound annotation MS/MS similarity (discussed below) is at least 0.7.
4a	Good formula MS/MS match	Candidate is top ranked, formula annotation MS/MS similarity (discussed below) is at least 0.7 and isotopic match (<code>isoScore</code>) of at least 0.5. The latter two scores are at least 0.2 higher than next best ranked candidate.

Level	Description	Rules
4b	Good formula isotopic pattern match	Candidate is top ranked and isotopic match (<code>isoScore</code>) of at least 0.9 and at least 0.2 higher than next best ranked candidate.
5	Unknown	All else.

NOTE The current identification level rules are *only* optimized when GenForm is used for formula annotation and MetFrag or Library is used for compound annotation.

The `genIDLevelRulesFile()` can be used to inspect the default rules or to create your own rules file, which can subsequently be passed to `estimateIDConfidence()` with the `IDFile` argument. A file with defaults is automatically created inside the project directory when using the `newProject` tool. Note that some rules are limited to suspect annotation.

The *annotation similarity* is a measure of how well the experimental MS/MS data is explained by the annotation algorithm. This is calculated as a spectral similarity between the spectrum with only the explained mass peaks and the full spectrum. A value of 1.0 means a full match, and is obtained when all experimental mass peaks could be explained. The annotation similarity is a straightforward measure to quickly evaluate the identification confidence, and is used for the estimation of identification levels (see Table above). For formulae and compounds, the annotation similarity is calculated automatically by `generateFormulas()` and `generateCompounds()`, respectively. The `estimateIDConfidence()` function for compounds can incorporate formula annotation similarities into compounds results. Furthermore, the `estimateIDConfidence()` function can be used to calculate formula and compound annotation similarities for suspects.

The estimation of confidence depends on the data available to `estimateIDConfidence()`. For instance, either the `fragments_mz` or `fragments_formula` suspect data is necessary to be able to assign a level 3b. Similarly, the suspect list needs retention times as well as fragment data to be able to assign level 1. For compounds and suspects, providing additional annotation workflow objects (`MSPeakLists`, `formulas`, `compounds`) as input to `estimateIDConfidence()` is necessary for calculation of most levels.

The `estimateIDConfidence()` function logs decisions for identification level assignments. This is useful to inspect level assignments and especially useful when you customized any rules. Note that this is disabled by default for formulae and compounds, since this often leads to a large number of log files. The `logPath` function argument is used to set to configure logging.

Some examples on how to use `estimateIDConfidence()` are shown below:

```
# calculate ID confidence for formulae
formulas <- estimateIDConfidence(formulas)

# calculate ID confidence for compounds. Supply peak lists and formulae to improve the
  ↳ estimations and enable logging.
compounds <- estimateIDConfidence(compounds, MSPeakLists = mslists, formulas = formulas,
  ↳ logPath = "log/comp-ident")

# calculate ID confidence for suspects. Supply peak lists, formulae and compounds to
  ↳ improve the estimations.
fGroupsSusp <- estimateIDConfidence(fGroupsSusp, MSPeakLists = mslists, formulas =
  ↳ formulas, compounds = compounds)
```

4.8.5 Account login for SIRIUS

Recent version of SIRIUS require an active account login to make queries to CSI:FingerID. This is primarily relevant when performing a compound annotation workflow with SIRIUS or a formula annotation workflow with `getFingerprints=TRUE`, e.g. when predicting toxicities or concentrations.

As a first step, please create an account as described in the SIRIUS documentation: <https://v6.docs.sirius-ms.io/account-and-license/>.

Then, to login there are two options:

1. Manually login: either by using the SIRIUS GUI or the CLI. For the latter, see e.g. `sirius.exe login --help` for more details.
2. Let `patRoön` automatically handle logins.

The `login` parameter for `generateCompounds()` and `generateFormulas()` determines how logins are dealt with by `patRoön`. There are four options:

1. `login=FALSE`: *no* logins are performed and *no* checks are performed to verify if there is an existing login.
2. `login="check"`: *no* logins are performed, but an active login is required to proceed.
3. `login="interactive"`: if no active login is present, then the username and password will be asked interactively and used to automatically login.
4. `login=c(username="...", password="...")`: if no active login is present, then the provided username and password will be used to automatically login.

NOTE: For the fourth option, please don't provide the login details directly as plain-text for security reasons. See below for proper alternatives.

The first two options are primarily meant for manual login. The function parameter `alwaysLogin=TRUE` can be set to force a login for the third and fourth options.

The fourth option is primarily useful for e.g. heavy users of SIRIUS or unattended automatic workflows. To securely provide the login details, it is best to store them elsewhere. This webpage provides a detailed overview of how credentials can be safely stored. For instance, you can save the credentials in your `.Renviron` file and retrieve them when calling `generateCompounds()`:

In your `.Renviron` file add:

```
SIRIUS_USERNAME=MY_USERNAME
SIRIUS_PASSWORD=MY_PASSWORD
```

and then in your R script:

```
compounds <- generateCompounds(..., login = c(username = Sys.getenv("SIRIUS_USERNAME"),
                                              password = Sys.getenv("SIRIUS_PASSWORD")))
```

Alternatively, you could use the `keyring` package, e.g.

```
install.packages("keyring") # execute in case you don't have keyring installed yet
keyring::key_set("SIRIUS", username = "myaccount@email.com") # execute this once to store
↳ the password

compounds <- generateCompounds(..., login = c(username = "myaccount@email.com",
                                              password = keyring::key_get("SIRIUS",
↳ "myaccount@email.com"))))
```

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
<code>length()</code>	All	Returns the length of the object (e.g. number of features, compounds etc)
<code>algorithm()</code>	All	Returns the name of the algorithm used to generate the object.
<code>groupNames()</code>	All	Returns all the unique identifiers (or names) of the feature groups for which this object contains results.
<code>names()</code>	<code>featureGroups</code> , <code>components</code>	Returns names of the feature groups (similar to <code>groupNames()</code>) or components
<code>show()</code>	All	Prints general information.
<code>"[" / "\$" operators</code>	All	Extract general information, see below.
<code>as.data.table() / as.data.frame()</code>	All	Convert data to a <code>data.table</code> or <code>data.frame</code> , see below.
<code>analysisInfo()</code> , <code>analyses()</code> , <code>replicates()</code>	<code>features</code> , <code>featureGroups</code>	Returns the analysis information, analyses or replicates for which this object contains data.
<code>groupInfo()</code>	<code>featureGroups</code>	Returns feature group information (<i>m/z</i> and retention time values).
<code>screenInfo()</code>	<code>featureGroupsScreening</code>	Returns information on hits from suspect screening.
<code>componentInfo()</code>	<code>components</code>	Returns information for all components.
<code>annotatedPeakList()</code>	<code>formulas</code> , <code>compounds</code>	Returns a table with annotated mass peaks (see below).

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

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

```
#>           ID      ret      mz      area intensity  retmin
#>      <char>  <num>    <num>    <num>    <num>    <num>
#>  1: f_4232806698149760419 14.139 98.97533 4778875.00   330176    5.945
#>  2: f_14882746253687013806  4.750 98.97542   96640.85   125856    4.351
#>  3: f_16682098354710394155  7.144 100.11199  470442.00   283356    5.945
```

```
#> 4: f_13995426688177709511 28.127 100.11208 5225358.00 304644 11.141
#> 5: f_13829096361419556271 4.550 100.11208 168080.50 76724 1.961
#> ---
#> 543: f_13168328237471625054 383.363 415.21303 364565.80 135352 381.122
#> 544: f_2439056283487317965 9.143 425.15511 415152.60 121928 8.143
#> 545: f_10389192741939298084 319.805 425.18879 732124.40 210844 317.060
#> 546: f_9282486000926943124 10.142 427.03246 365056.90 114896 8.143
#> 547: f_9837132258956386493 9.143 433.00456 3165097.00 946000 8.143
```

```
# Feature group intensities
fGroups$M120_R268_30
```

```
#> [1] 264836 245372 216560
```

```
fGroups[[1, "M120_R268_30"]] # only first analysis
```

```
#> [1] 264836
```

```
# obtains MS/MS peak list (feature group averaged data)
mslists[["M120_R268_30"]]$MSMS
```

```
#>      ID      mz  intensity fgroup_abundance_rel fgroup_abundance_abs feat_abundance_rel feat_abun
#>      <int>    <num>      <num>                <num>                <num>                <num>
#> 1:      5 105.0698   6183.111                    1                    3                    1
#> 2:      6 106.0652   7643.556                    1                    3                    1
#> 3:      8 107.0728   7760.667                    1                    3                    1
#> 4:     15 120.0556  168522.672                    1                    3                    1
#> 5:     17 121.0587  13894.667                    1                    3                    1
#> 6:     18 121.0883  10032.888                    1                    3                    1
#> 7:     19 122.0964  147667.766                    1                    3                    1
#> 8:     20 123.0803  36631.109                    1                    3                    1
#> 9:     21 123.0996  15482.445                    1                    3                    1
#> 10:    22 124.0806  35580.668                    1                    3                    1
```

```
# get all formula candidates for a feature group
formulas[["M120_R268_30"]][, 1:7]
```

```
#>      neutral_formula ion_formula neutralMass ion_formula_mz      error      dbc isoScore
#>      <char>         <char>         <num>         <num>         <num> <num>      <num>
#> 1:      C6H5N3      C6H6N3      119.0483      120.0556 1.566667      6 0.92461
```

```
# get all compound candidates for a feature group
compounds[["M120_R268_30"]][, 1:4]
```

```
#>      explainedPeaks      score neutralMass      SMILES
#>      <int>          <num>      <num>          <char>
#> 1:      0 3.0000000      119.0483      C1=CC2=NNN=C2C=C1
#> 2:      0 0.7285793      119.0483      C1=CC2=C(N=C1)N=CN2
#> 3:      0 0.4262336      119.0483      C1=CNC2=CN=CN=C21
```

```
#> 4:      0 0.3712626      119.0483      C1=CC2=C(C=NN2)N=C1
#> 5:      0 0.3356653      119.0483      C1=CN2C(=CC=N2)N=C1
#> ---
#> 37:     0 0.1259205      119.0483      CC1=CN=CC(=N1)C#N
#> 38:     0 0.1257969      119.0483      C1=CN=CN=C1CC#N
#> 39:     0 0.1250824      119.0483      CC1=CN=CN=C1C#N
#> 40:     0 0.1250137      119.0483      C#CC1=NC=CN=C1N
#> 41:     0 0.1250000      119.0483      C1=CC(=[N+]=[N-])C=CC1=N
```

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

```
#>      group      ret      mz isogroup isonr charge
#>      <char>    <num>    <num>    <num> <num>  <num>
#> 1: M143_R206_64 205.787 143.0700      NA     NA     NA
#> 2: M159_R208_103 208.280 159.0650      NA     NA     NA
#> 3: M161_R208_104 207.582 161.0806      NA     NA     NA
#> 4: M181_R209_159 208.580 181.0469      NA     NA     NA
```

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 `patRoan` 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 intensity
#>      <char>      <char>    <num>    <num>    <num>    <num>
#> 1: solvent-pos-1 f_11604688393433660807 13.176 98.97537 4345232.0 391476
#> 2: solvent-pos-1 f_13795944494377381066  7.181 100.11197 797112.1 426956
#> 3: solvent-pos-1 f_9994115579808516414 192.178 100.11211 9609998.0 750532
#> 4: solvent-pos-1 f_7616357632595552956  19.171 100.11217 5784411.0 370376
#> 5: solvent-pos-1 f_14235772935233178535  4.786 100.11220 551723.6 567312
#> ---
#> 2922: standard-pos-3 f_9650600993545486389 318.892 425.18866 666531.5 232636
#> 2923: standard-pos-3 f_7973765326008001809  9.114 427.03242 362024.1 114744
#> 2924: standard-pos-3 f_13437633323133925028 318.892 427.18678 200193.5 77768
#> 2925: standard-pos-3 f_12182984943585750598 382.682 432.23984 217612.9 97648
#> 2926: standard-pos-3 f_3361210316269368642  9.114 433.00457 3086864.0 912920
```

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

```
#>           group      ret      mz standard-pos_intensity
#>           <char>    <num>    <num>                <num>
#>  1: M109_R192_20 191.8717 109.0759                183482.67
#>  2: M111_R330_23 330.4078 111.0439                84598.67
#>  3: M114_R269_25 268.6906 114.0912                 85796.00
#>  4: M116_R317_29 316.7334 116.0527                766888.00
#>  5: M120_R268_30 268.4078 120.0554                242256.00
#> ---
#> 137: M316_R363_635 363.4879 316.1741                 89904.00
#> 138: M318_R349_638 349.1072 318.1450                 83320.00
#> 139: M352_R335_664 334.9403 352.2019                 74986.67
#> 140: M407_R239_672 239.3567 407.2227                186568.00
#> 141: M425_R319_676 319.4944 425.1885                214990.67
```

```
# As above, but with suspect matches on separate rows and additional screening
↳ information
# (select some columns to simplify the output below)
as.data.table(fGroupsSusp, average = TRUE, collapseSuspects = NULL,
              onlyHits = TRUE)[, c("group", "susp_name", "susp_compRank",
                                  ↳ "susp_annSimComp", "susp_estIDLevel")]
```

```
#>           group      susp_name susp_compRank susp_annSimComp susp_estIDLevel
#>           <char>          <char>         <int>         <num>          <char>
#>  1: M120_R268_30 1H-benzotriazole           1           0.0000000          4b
#>  2: M137_R249_53  N-Phenyl urea            1           0.6443557          3a
#>  3: M146_R309_68 2-Hydroxyquinoline         2           0.9896892          3a
#>  4: M146_R248_69 2-Hydroxyquinoline        NA              NA              5
#>  5: M146_R225_70 2-Hydroxyquinoline        NA              NA              5
```

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

```
#>           group  type  ID      mz intensity fgroup_abundance_rel fgroup_abundance_abs feat_ab
#>           <char> <char> <int>    <num>    <num>                <num>                <num>
#>  1: M120_R268_30  MS     1 100.1120 178952.38                1.0000000                3
#>  2: M120_R268_30  MS     2 102.1277 202359.67                1.0000000                3
#>  3: M120_R268_30  MS     3 114.0912  37647.55                1.0000000                3
#>  4: M120_R268_30  MS     4 115.0752  66685.24                1.0000000                3
#>  5: M120_R268_30  MS     5 120.0555 113335.85                1.0000000                3
#> ---
#> 236: M192_R355_191  MS    52 298.1328 16943.31                0.6666667                2
#> 237: M192_R355_191  MS    53 299.1274  45880.92                1.0000000                3
#> 238: M192_R355_191 MSMS   14 119.0496 588372.44                1.0000000                3
#> 239: M192_R355_191 MSMS   19 120.0524  70273.34                1.0000000                3
#> 240: M192_R355_191 MSMS   32 192.1383  71978.66                1.0000000                3
```

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

```
#>      group neutral_formula ion_formula neutralMass ion_formula_mz      error
#>      <char>      <char>      <char>      <num>      <num>      <num>
#> 1: M120_R268_30      C6H5N3      C6H6N3      119.0483      120.0556  1.566667
#> 2: M137_R249_53      C7H8N2O      C7H9N2O      136.0637      137.0709  2.400000
#> 3: M146_R309_68      C9H7NO      C9H8NO      145.0528      146.0600  1.400000
#> 4: M192_R355_191     C12H17NO     C12H18NO      191.1310      192.1383 -1.966667
```

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

```
#>      group explainedPeaks      score neutralMass
#>      <char>      <int>      <num>      <num>
#> 1: M120_R268_30          0 3.0000000      119.0483
#> 2: M120_R268_30          0 0.7285793      119.0483
#> 3: M120_R268_30          0 0.4262336      119.0483
#> 4: M120_R268_30          0 0.3712626      119.0483
#> 5: M120_R268_30          0 0.3356653      119.0483
#> ---
#> 293: M192_R355_191        1 0.7620768      191.1310
#> 294: M192_R355_191        1 0.7620768      191.1310
#> 295: M192_R355_191        1 0.7620768      191.1310
#> 296: M192_R355_191        1 0.7620768      191.1310
#> 297: M192_R355_191        1 0.7620768      191.1310
```

```
# 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
#>      <char>      <num>      <num>      <char>      <char>      <int>
#> 1:  CMP1 347.2914 0.0000000      <NA> standard-pos-2          2
#> 2:  CMP1 347.2914 0.0000000      <NA> standard-pos-2          2
#> 3:  CMP2 349.6328 4.6804985 225.1589/188.20157 standard-pos-3          6
#> 4:  CMP2 349.6328 4.6804985 225.1589/188.20157 standard-pos-3          6
#> 5:  CMP2 349.6328 4.6804985 225.1589/188.20157 standard-pos-3          6
#> ---
#> 88: CMP29 313.3475 0.3105035      <NA> standard-pos-2          3
#> 89: CMP29 313.3475 0.3105035      <NA> standard-pos-2          3
#> 90: CMP30 268.3430 0.3840764      81.08705 standard-pos-1          3
#> 91: CMP30 268.3430 0.3840764      81.08705 standard-pos-1          3
#> 92: CMP30 268.3430 0.3840764      81.08705 standard-pos-1          3
```

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

```
# formula annotations for the first formula candidate of feature group M137_R249_53
annotatedPeakList(formulas, index = 1, groupName = "M137_R249_53",
  MSPeakLists = mslists)
```

```
#>      ID      mz intensity fgroup_abundance_rel fgroup_abundance_abs feat_abundance_rel feat_abun
#>    <int>    <num>    <num>                <num>                <num>                <num>
#> 1:      2  94.06500  9406.110                  1                  3                  1
#> 2:      6  98.97521  2212.000                  1                  3                  1
#> 3:      7 105.06971  1662.111                  1                  3                  1
#> 4:     14 120.04435  7176.222                  1                  3                  1
#> 5:     19 122.07218  2246.000                  1                  3                  1
#> 6:     21 135.08005  1565.556                  1                  3                  1
#> 7:     23 137.07040  5348.667                  1                  3                  1
#> 8:     24 137.09570  2026.889                  1                  3                  1
#> 9:     26 138.09116 12356.667                  1                  3                  1
#> 10:    27 139.07501  5020.667                  1                  3                  1
```

```
# compound annotation for first candidate of feature group M137_R249_53
annotatedPeakList(compounds, index = 1, groupName = "M137_R249_53",
                  MSPeakLists = mslists)
```

```
#>      ID      mz intensity fgroup_abundance_rel fgroup_abundance_abs feat_abundance_rel feat_abun
#>    <int>    <num>    <num>                <num>                <num>                <num>
#> 1:      2  94.06500  9406.110                  1                  3                  1
#> 2:      6  98.97521  2212.000                  1                  3                  1
#> 3:      7 105.06971  1662.111                  1                  3                  1
#> 4:     14 120.04435  7176.222                  1                  3                  1
#> 5:     19 122.07218  2246.000                  1                  3                  1
#> 6:     21 135.08005  1565.556                  1                  3                  1
#> 7:     23 137.07040  5348.667                  1                  3                  1
#> 8:     24 137.09570  2026.889                  1                  3                  1
#> 9:     26 138.09116 12356.667                  1                  3                  1
#> 10:    27 139.07501  5020.667                  1                  3                  1
```

More advanced examples for these functions are shown below.

```
# Feature table, can also be accessed by numeric index
fList[[1]]
mslists[["standard-pos-1", "M120_R268_30"]] # feature data (instead of feature group
→ averaged)
formulas[[1, "M120_R268_30"]] # 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) # also include feature information
as.data.table(fGroups, average = "fGroups") # output a simple data with feature
→ group-averaged intensities
as.data.table(fGroups, average = "fGroups",
              features = TRUE) # include averaged/collapsed feature data

as.data.table(mslists, averaged = FALSE) # peak lists for 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)
```



```

# 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 information
as.data.table(compounds, fragments = TRUE) # include information of all annotated
↪ fragments

annotatedPeakList(formulas, index = 1, groupName = "M120_R268_30",
                  MSPeakLists = mslists, onlyAnnotated = TRUE) # only include annotated
↪ peaks
annotatedPeakList(compounds, index = 1, groupName = "M120_R268_30",
                  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 **patRoön** 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 may need to explicitly specify which one to use in your script. This can be done by prefixing it with the package name, e.g. **patRoön::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)
absMinMaxIntensity, relMinMaxIntensity	featureGroups	The complete feature group is removed if the maximum intensity of the group is below this value.
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
maxReplicateIntRSD	featureGroups	Maximum relative standard deviation of feature intensities in a replicate.
blankThreshold	featureGroups	Minimum intensity factor above blank intensity
replicate	featureGroups	Only keep (features of) these replicates
results	featureGroups	Only keep feature groups with formula/compound annotations or componentization results

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 `patRoön` 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
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, even when replicates 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
  ↪ replicates
  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(fGroups,
  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 replicates
fGroups <- filter(fGroups, relMinReplicates = 0.2)

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

# only keep feature groups with compound annotations
fGroups <- filter(fGroups, results = compounds)
# only keep feature groups with formula or compound annotations
fGroups <- filter(fGroups, results = list(formulas, compounds))

```

5.2.2 Suspect screening

Several additional filters are available for feature groups obtained with `screenSuspects()`:

Filter	Classes	Remarks
<code>onlyHits</code>	<code>featureGroupsScreening</code>	Only retain feature groups assigned to one or more suspects.
<code>selectHitsBy</code>	<code>featureGroupsScreening</code>	Select the feature group that matches best with a suspect (in case there are multiple).
<code>selectBestFGroups</code>	<code>featureGroupsScreening</code>	Select the suspect that matches best with a feature group (in case there are multiple).
<code>maxLevel, maxFormRank, maxCompRank</code>	<code>featureGroupsScreening</code>	Only retain suspect hits with identification/annotation ranks below a threshold.
<code>minAnnSimForm, minAnnSimComp, minAnnSimBoth</code>	<code>featureGroupsScreening</code>	Remove suspect hits with annotation similarity scores below this value.

Filter	Classes	Remarks
absMinFragMatches, relMinFragMatches	featureGroupsScreenOnly	Only keep suspect hits with a minimum (relative) number of fragment matches from the suspect list.

NOTE: most filters only remove suspect hit results. Set `onlyHits=TRUE` to also remove any feature groups that end up without suspect hits.

The `selectHitsBy` and `selectBestFGroups` filters are useful to remove duplicate hits, i.e. the same suspect assigned to multiple feature groups or multiple suspects assigned to the same feature group, respectively. The former selects based on either best identification level (`selectHitsBy="level"`) or highest mean intensity (`selectHitsBy="intensity"`). The `selectBestFGroups` can only be TRUE/FALSE and always selects by best identification level.

Some examples are shown below.

```
# only keep feature groups assigned to at least one suspect
fGroupsSusp <- filter(fGroupsSusp, onlyHits = TRUE)
# remove duplicate suspect to feature group matches and keep the best
fGroupsSusp <- filter(fGroupsSusp, selectHitsBy = "level")
# remove suspect hits with ID levels >3 and make sure no feature groups
# are present without suspect hits afterwards
fGroupsSusp <- filter(fGroupsSusp, maxLevel = 3, onlyHits = TRUE)
```

5.2.3 Annotation

There are various filters available for handling annotation data:

Filter	Classes	Remarks
MSLevel	MSPeakLists	To which MS level the filters should be applied (1, 2 or 1:2)
absMinIntensity, relMinIntensity	MSPeakLists	Minimum intensity for mass peaks
topMostPeaks	MSPeakLists	Only keep this number of most intense mass peaks
minPeaks	MSPeakLists	Only keep a peak list if it contains a minimum number of peaks (excluding the precursor peak, only for MSLevel=2)
maxMZOverPrec	MSPeakLists	Mass peaks with this m/z higher than the precursor peak are removed
withMSMS	MSPeakLists	Only keep results with MS/MS data
annotatedBy	MSPeakLists	Only keep peaks that have formula or compound annotations (only for MSLevel=2)
removeMZs	MSPeakLists	Removal of background peaks
minExplainedPeaks	formulas, compounds	Minimum number of annotated mass peaks
elements, fragElements, lossElements	formulas, compounds	Restrain elemental composition
fragFormulas, lossFormulas	formulas, compounds	Only keep candidates with at least one fragment formula or neutral loss match

Filter	Classes	Remarks
topMost	formulas, compounds	Only keep highest ranked candidates
minScore, minFragScore, minFormulaScore	compounds	Minimum compound scorings
scoreLimits	formulas, compounds	Minimum/Maximum scorings
OM	formulas, compounds	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 precursor mass peak of the feature itself.

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, MSLevel = 1, # only MS1 data
                  absMinIntensity = 500, # minimum mass peak intensity of 500
                  relMinIntensity = 0.1) # and minimum mass peak intensity of 10%

# only retain 10 most intense mass peaks (MS and MS/MS)
# (feature mass is always retained)
mslists <- filter(mslists, MSLevel = 1:2, topMostPeaks = 10)

# remove MS/MS peaks without compound annotations
mslists <- filter(mslists, MSLevel = 2, annotatedBy = compounds)

# remove MS/MS peaks not annotated by either a formula or compound candidate
mslists <- filter(mslists, MSLevel = 2, annotatedBy = list(formulas, compounds))

# 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 = "C1-1000-2")

# only keep candidates with CO2 or H2O neutral loss
formulas <- filter(formulas, lossFormulas = c("CO2", "H2O"))

# only keep candidates with formula annotation similarity >=0.6
formulas <- filter(formulas, scoreLimits = list(annSim = c(0.6, Inf)))

# 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)))

# only keep candidates with formula annotation similarity >=0.8 and
# compound annotation similarity >= 0.7
# (requires that estimateIDConfidence() was called on the compounds objects)
compounds <- filter(compounds,
                    scoreLimits = list(annSimForm = c(0.8, Inf), annSim = c(0.7, Inf)))

```

NOTE As of patRoön 2.0 MS peak lists are **not** re-generated after a filtering operation (unless the `reAverage` parameter is explicitly set to `TRUE`). The reason for this change is that re-averaging invalidates any formula/compound annotation data (e.g. used for plotting and reporting) that were generated prior to the filter operation.

5.2.4 Components

Finally several filters are available for components:

Filter	Remarks
<code>size</code>	Minimum component size
<code>adducts, isotopes</code>	Filter features by adduct/isotopes annotation
<code>rtIncrement, mzIncrement</code>	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]+"))

# 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!

5.2.5 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.

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

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

# remove duplicate suspect hits and only keep the _worst_ hit
fGroupsSusp <- filter(fGroupsSusp, selectHitsBy = "level", 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	set <code>reAverage=TRUE</code> to re-average peak lists after subsetting
formulas	feature groups		
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)]

# select five feature groups
mslists[, 1:5]

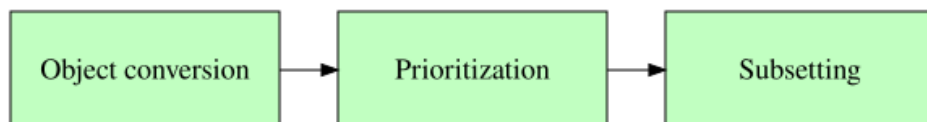
# ... and perform re-averaging
# NOTE: this will invalidate any existing formulas/compound annotation data!
mslists[, 1:5, reAverage = TRUE]
```


In addition, feature groups can also be subset by given replicates or annotation/componentization results (similar to `filter()`). Similarly, suspect screening results can also be subset by given suspect names.

```
# equal as filter(fGroups, replicates = ...)
fGroups[replicates = c("rep11", "rep12")]
# equal as filter(fGroups, results = ...)
fGroups[results = compounds]
# only keep feature groups assigned to given suspects
fGroupsSusp[suspects = c("1H-benzotriazole", "2-Hydroxyquinoline")]
```

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
# (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 were in the top 5
```

5.4 Deleting data

The `delete()` generic function can be used to manually delete workflow data. This function is used internally within `patRoan` to implement filtering and subsetting operations, but may also be useful for advanced data processing.

Similar to the subset operator, this function accepts a `i`, `j` and additionally `k` parameter to specify which data should be operated on:

Class	Argument i	Argument j	Argument k
features	analysis	feature index	
featureGroups	analysis	feature group	

Class	Argument i	Argument j	Argument k
featureGroupsScreening	analysis	feature group	suspect (can be NA, must be NULL) MSPeakLists feature group mass peak analysis (if set, no deletion occurs on group-averaged peak lists) formulas, compounds feature group candidate index components

If i, j and/or k is not specified (NULL) then data is removed for the complete selection. Some examples are shown below:

```
# delete 2nd feature in analysis-1
fList <- delete(fList, i = "analysis-1", j = 2)
# delete first ten features in all analyses
fList <- delete(fList, i = NULL, j = 1:10)

# completely remove third+fourth analyses from feature groups
fGroups <- delete(fGroups, i = 3:4)
# delete specific feature group
fGroups <- delete(fGroups, j = "M120_R268_30")
# delete range of feature groups
fGroups <- delete(fGroups, j = 500:750)

# remove all hits for atrazine
fGroupsSusp <- delete(k = "atrazine")
# for a specific feature group
fGroupsSusp <- delete(j = "M120_R268_30", k = "atrazine")
# remove all hits for a feature group
fGroupsSusp <- delete(j = "M120_R268_30", k = NA)

# remove all MS peak lists for a feature group
mslists <- delete(mslists, i = "M120_R268_30")
# removes the first 5 peaks in all peak lists
mslists <- delete(mslists, j = 1:5)
# remove all MS peak lists for a specific analysis
mslists <- delete(mslists, k = "standard-pos-1")

# remove all results for a feature group
formulas <- delete(formulas, i = "M120_R268_30")

# remove top candidate for all feature groups
compounds <- delete(compounds, j = 1)

# remove a component
components <- delete(components, i = "CMP1")
# remove specific feature group from a component
components <- delete(components, i = "CMP1", j = "M120_R268_30")
# remove specific feature group from all components
components <- delete(components, j = "M120_R268_30")
```

The value set to the j and k (for suspect screening results) can also be a function: the function called

repeatedly on parts of the data to select what should be deleted. How the function is called and what it should return depends on the workflow data class:

Class	Called on every	Arguments	Return value
features	analysis	features (data.table), analysis name	Features indices (as integer or logical)
featureGroups , featureGroupsScreening	feature group	group intensities (vector), feature group name	The analyses of the features to remove (as character , integer , logical)
featureGroupsScreening (k)		The screening table	A logical vector for the rows to delete.
mslists	peak list	mass peaks (data.table), feature group name, analysis name (NULL for group averaged data), type ("MS" or "MSMS")	Mass peak indices (as integer , logical)
formulas , compounds	feature group	annotations (data.table), feature group name	Candidate indices (rows)
components	component	component (data.table), component name	The feature groups (as character , integer)

Some examples for this:

```
# remove features with intensities below 5000
fList <- delete(fList, j = function(f, ...) f$intensity <= 5E3)

# same, but for features in all feature groups from specific analyses
fGroups <- delete(fGroups, i = 1:3, j = function(g, ...) g <= 5E3)

# remove hits for suspects with mass above 400 Da
fGroupsSusp <- delete(fGroupsSusp, k = function(tab) tab$neutralMass > 400)

mslists <- delete(mslists, j = function(pl, grp, ana, type)
{
  if (!is.null(ana) || type == "MS")
    return(FALSE) # only delete peaks from group averaged MS/MS peak lists
  return(pl$mz > 500) # remove peaks with m/z > 500
})

# remove formula candidates with high relative mass deviation
formulas <- delete(formulas, j = function(ft, ...) ft$error > 5)
```

5.5 Interactively explore and review data

The **checkFeatures** and **checkComponents** functions start a graphical user interface (GUI) which allows you to interactively explore and review feature and components data, respectively.

```
checkFeatures(fGroups) # inspect features and feature groups
checkComponents(componCAM, fGroups) # inspect components
```

Both functions allow you to easily explore the data in an interactive way. Furthermore, these functions allow you to remove unwanted data. This is useful to remove for example features that are actually noise and

feature groups that shouldn't be in the same component. To remove an unwanted feature, feature group or components, simply uncheck its 'keep' checkbox. The next step is to save the selections you made. A *check session* is a file that stores which data should be removed. Once the session file is saved the `filter` function can be used to actually remove the data:

```
fGroupsF <- filter(fGroups, checkFeaturesSession = TRUE)
componCAMF <- filter(componCAM, checkComponentsSession = TRUE)
```

If you saved the session and you re-launch the GUI it will restore the selections made earlier. The `clearSession` argument can be used to fully clear a session before starting the GUI, hence, all the data will be restored to their 'keep state'.

```
checkFeatures(fGroups, clearSession = TRUE) # start GUI with fresh session
```

It is also possible to use multiple different sessions. This is especially useful if you do not want to overwrite previous session data or want to inspect different objects. In this case the session file name should be specified:

```
checkFeatures(fGroups, "mysession.yml")
fGroupsF <- filter(fGroups, checkFeaturesSession = "mysession.yml")
```

The default session names are "checked-features.yml" and "checked-components.yml" for feature and component data, respectively.

The extension of session file names is .yml since the YAML file format is used. An advantage of this format is that it is easily readable and editable with a text editor.

Note that the session data is tied to the feature group names of your data. This means that, for instance, when you re-group your feature data after changing some parameters, the session data you prepared earlier cannot be used anymore. Since probably quite some manual work went into creating the session file, a special function is available to import a session that was made for previous data. This function tries its best to guess the new feature group name based on similarity of their retention times and *m/z* values.

```
checkFeatures(fGroups) # do manual inspection

fGroups <- groupFeatures(fList, ...) # re-group with different parameters

importCheckFeaturesSession("checked-features.yml", "checked-features-new.yml", fGroups)

checkFeatures(fGroups, session = "checked-features-new.yml") # inspect new data
```

Take care to monitor the messages that `importCheckFeaturesSession` may output, as it may be possible that some 'old' feature groups are not found or are matched by multiple candidates of the new dataset.

Some additional parameters exist to the functions described in this section. As usually check the reference manual for more details (*e.g.* `?checkFeatures`).

NOTE Although the GUI tools described here allow you to easily filter out results, it is highly recommended to first prioritize your data to avoid doing a lot of unneeded manual work.

5.6 Updating feature group data

The feature group properties such as retention time and m/z are calculated during the feature grouping process. These properties usually represent an average calculated from the features within the group, and are e.g. used to match suspects. It may be useful to recalculate these values after e.g. filtering noisy features to improve the accuracy. This is achieved with the `updateGroups` function:

```
fGroups <- filter(fGroups, ...) # perform filtering steps

fGroups <- updateGroups(fGroups) # update retention time and m/z by re-averaging data
fGroups <- updateGroups(fGroups, intWeight = TRUE) # use intensity weighting for
  ↪ averaging
fGroups <- updateGroups(fGroups, what = c("mz")) # only update m/z values
```

5.7 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 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 analyses or replicates. Both functions return a subset of the given `featureGroups` object. An overview of their arguments is given below.

Argument	Function(s)	Remarks
<code>which</code>	<code>unique()</code> , <code>overlap()</code>	Which data is compared (e.g. names of replicates). Can be NULL for overlap to include all data.
<code>aggregate</code>	<code>unique()</code> , <code>overlap()</code>	What should be compared: analyses (<code>aggregate=FALSE</code>), replicates (<code>aggregate=TRUE</code>) or other groups of analyses based on their metadata (discussed later).
<code>relativeTo</code>	<code>unique()</code>	Uniqueness of features is only determined by these analyses, replicates, ... (NULL for all). Entries in <code>which</code> are ignored.
<code>outer</code>	<code>unique()</code>	If TRUE then only return features which are <i>also</i> unique among the compared data selected in <code>which</code> .
<code>exclusive</code>	<code>overlap</code>	Only keep features that <i>only</i> overlap between the compared data selected by <code>which</code> .

Some examples:

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

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

```
fGroupsOv2 <- overlap(fGroups, which = c("repl1", "repl2"), aggregate = TRUE,
                      exclusive = TRUE)
```

Note that several plotting functions that are discussed in the visualization section can also assist in determining overlap and uniqueness of features.

5.8 MS similarity

The *spectral similarity* is used to compare spectra from different features. For this purpose the `spectrumSimilarity` function can be used. This function operates on MS peak lists, and accepts the following function arguments:

Argument	Remarks
<code>MSPeakLists</code>	The MS peak lists object from which peak lists data should be taken.
<code>groupName1,</code> <code>groupName2</code>	The name(s) of the first and second feature group(s) to compare
<code>analysis1, analysis2</code>	The analysis names of the data to be compared. Set this when feature data (instead of feature group data) should be compared.
<code>MSLevel</code>	The MS level: 1 or 2 for MS and MS/MS, respectively.
<code>specSimParams</code>	Parameters that define how similarities are calculated.
<code>NAToZero</code>	If TRUE then NA values are converted to zeros. NA values are reported if a comparison cannot be made because of missing peak list data.

The `specSimParams` argument defines the parameters for similarity calculations. It is a `list`, and the default values are obtained with the `getDefSpecSimParams()` function:

```
getDefSpecSimParams()
```

```
#> $method
#> [1] "cosine"
#>
#> $removePrecursor
#> [1] FALSE
#>
#> $mzWeight
#> [1] 0
#>
#> $intWeight
#> [1] 1
#>
#> $absMzDev
#> [1] 0.005
#>
#> $relMinIntensity
#> [1] 0.05
#>
#> $minPeaks
#> [1] 1
#>
#> $shift
```

```
#> [1] "none"
#>
#> $setCombineMethod
#> [1] "mean"
```

The `method` field describes the calculation measure: this is either `"cosine"` or `"jaccard"`.

The `shift` field is primarily useful when comparing MS/MS data and defines if and how a spectral shift should be performed prior to similarity calculation:

- `"none"`: The default, no shifting is performed.
- `"precursor"` The mass difference between the precursor mass of both spectra (*i.e.* the feature mass) is first calculated. This difference is then subtracted from each of the mass peaks of the second spectrum. This shifting increases similarity if the MS fragmentation process itself occurs similarly (*i.e.* if both features show similar neutral losses).
- `"both"` This combines both shifting methods: first peaks are aligned that have the same mass, then the `precursor` strategy is applied for the remaining mass peaks. This shifting method yields higher similarities if either fragment masses or neutral losses are similar.

To override a default setting, simply pass it as an argument to `getDefSpecSimParams`:

```
getDefSpecSimParams(shift = "both")
```

For more details on the various similarity calculation parameters see the reference manual (`?getDefSpecSimParams`).

Some examples are shown below:

```
# similarity between MS spectra with default parameters
spectrumSimilarity(mslists, groupName1 = "M120_R268_30", groupName2 = "M137_R249_53")
```

```
#> [1] 0.4088499
```

```
# similarity between MS/MS spectra with default parameters
spectrumSimilarity(mslists, groupName1 = "M120_R268_30", groupName2 = "M192_R355_191",
  MSLevel = 2)
```

```
#> [1] 0.08589849
```

```
# As above, with jaccard calculation
spectrumSimilarity(mslists, groupName1 = "M120_R268_30", groupName2 = "M192_R355_191",
  MSLevel = 2, specSimParams = getDefSpecSimParams(method = "jaccard"))
```

```
#> [1] 0.1111111
```

```
# With shifting
spectrumSimilarity(mslists, groupName1 = "M120_R268_30", groupName2 = "M192_R355_191",
  MSLevel = 2, specSimParams = getDefSpecSimParams(shift = "both"))
```

```
#> [1] 0.08589849
```

The `spectrumSimilarity` function can also be used to calculate *multiple* similarities. Simply specify multiple feature group names for the `groupNameX` parameters. Alternatively, if you want to compare the same set of feature groups with each other pass their names only as the `groupName1` parameter:

```
# compare two pairs
spectrumSimilarity(mslists,
  groupName1 = c("M120_R268_30", "M137_R249_53"),
  groupName2 = c("M146_R309_68", "M192_R355_191"),
  MSLevel = 2, specSimParams = getDefSpecSimParams(shift = "both"))
```

```
#>           M146_R309_68 M192_R355_191
#> M120_R268_30      0.520052    0.08589849
#> M137_R249_53      0.197720    0.03372542
```

```
# compare all
spectrumSimilarity(mslists, groupName1 = groupNames(mslists),
  MSLevel = 2, specSimParams = getDefSpecSimParams(shift = "both"))
```

```
#>           M120_R268_30 M137_R249_53 M146_R309_68 M192_R355_191
#> M120_R268_30      1.00000000    0.20406382    0.52005204    0.08589849
#> M137_R249_53      0.20406382    1.00000000    0.19772003    0.03372542
#> M146_R309_68      0.52005204    0.19772003    1.00000000    0.08524784
#> M192_R355_191     0.08589849    0.03372542    0.08524784    1.00000000
```

5.9 Visualization

5.9.1 Features and annotation data

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

Generic	Classes	Remarks
<code>plot()</code>	<code>featureGroups</code> , <code>featureGroupsComparison</code>	Scatter plot for retention and m/z values
<code>plotInt()</code>	<code>featureGroups</code>	Intensity profiles across analyses
<code>plotChroms()</code>	<code>featureGroups</code> , <code>components</code>	Plot extracted ion chromatograms (EICs)
<code>plotSpectrum()</code>	<code>MSPeakLists</code> , <code>formulas</code> , <code>compounds</code> , <code>components</code>	Plots (annotated) spectra
<code>plotStructure()</code>	<code>compounds</code>	Draws candidate structures
<code>plotScores()</code>	<code>formulas</code> , <code>compounds</code>	Barplot with candidate scoring
<code>plot()</code>	<code>componentsClust</code>	Creates a dendrogram for the clustered components
<code>plotGraph()</code>	<code>componentsNT</code>	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
retMin	plotChroms()	If TRUE plot retention times in minutes
EICParams	plotChroms()	Advanced parameters to control the creation of extracted ion chromatograms (described below)
showPeakArea, showFGroupRect	plotChroms()	Fill peak areas / draw rectangles around feature groups?
title	plotChroms(), plotSpectrum()	Override plot title
showLegend	plot(), plotInt(), plotChroms()	Display a legend?
xlim, ylim	plotChroms(), plotSpectrum()	Override x/y axis ranges, i.e. to manually set plotting range.
groupBy	plot(), plotInt(), plotChroms()	Defines if and how data is grouped in the plot (e.g. by replicate)
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 annotations should be added (only compounds)
plotStruct	plotSpectrum()	Whether the structure should be added to the plot (only compounds)
mincex	plotSpectrum()	Minimum annotation font size (only formulas/compounds)

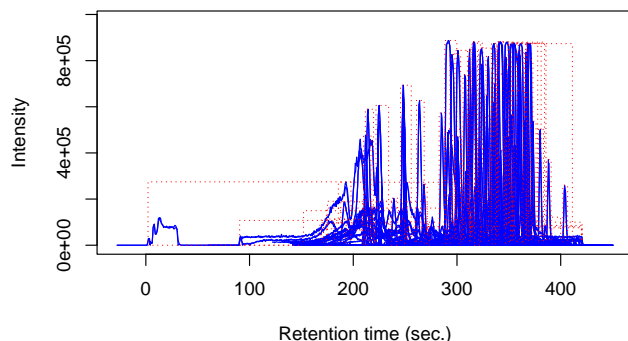
Note that we can use subsetting to select which feature data we want to plot, e.g.

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

```
#> Using 'mzr' backend for reading MS data.
```

```
#> =====
```

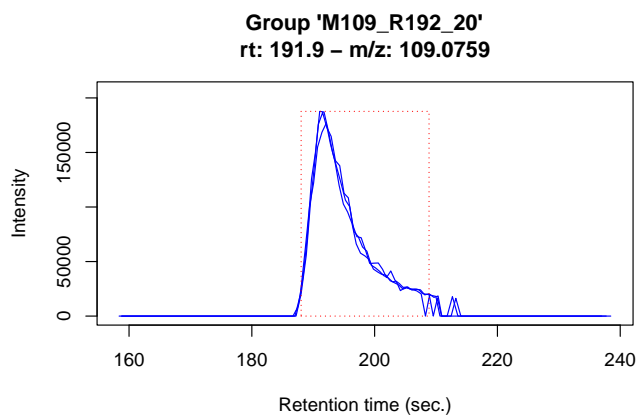
141 feature groups



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

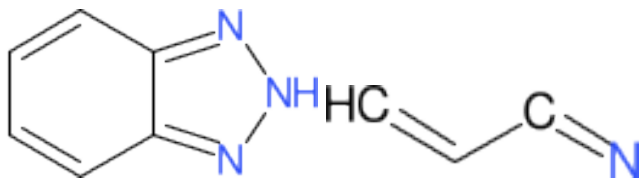
```
#> Using 'mzr' backend for reading MS data.
```

```
#> =====
```



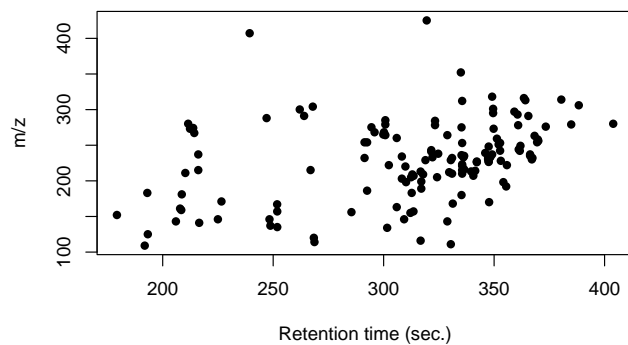
The `plotStructure()` function will draw a chemical structure for a compound candidate. In addition, this function can draw the maximum common substructure (MCS) of multiple candidates in order to assess common structural features.

```
# structure for first candidate
plotStructure(compounds, index = 1, groupName = "M120_R268_30")
# MCS for first three candidates
plotStructure(compounds, index = 1:3, groupName = "M120_R268_30")
```

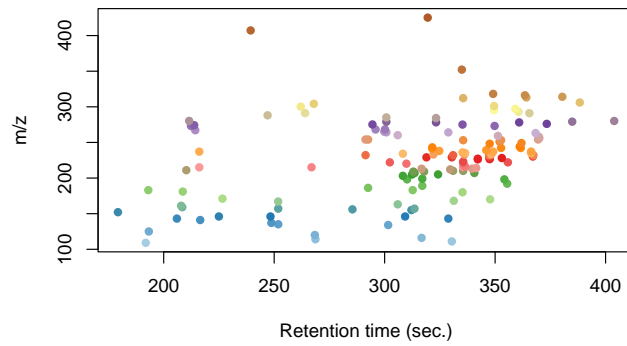


Some other common and less common plotting operations are shown below.

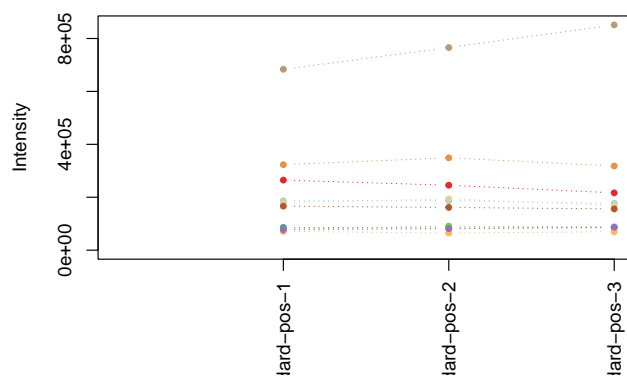
```
plot(fGroups) # simple scatter plot of retention and m/z values
```



```
plot(fGroups, groupBy = "fGroups", showLegend = FALSE) # colour each feature group
```



```
plotInt(fGroups[, 1:10], groupBy = "fGroups") # plot intensities of first ten feature
↳ groups and group them
```

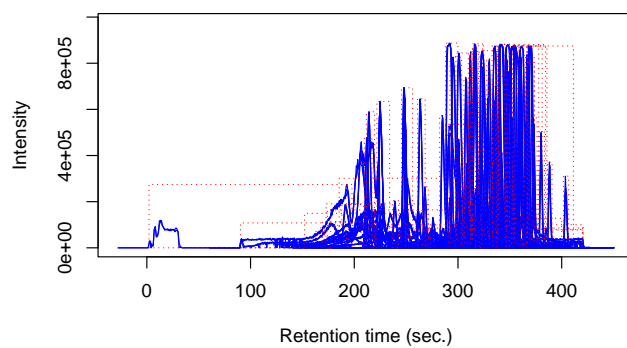


```
plotChroms(fGroups) # plot EICs for all features
```

```
#> Using 'mzr' backend for reading MS data.
```

```
#> =====
```

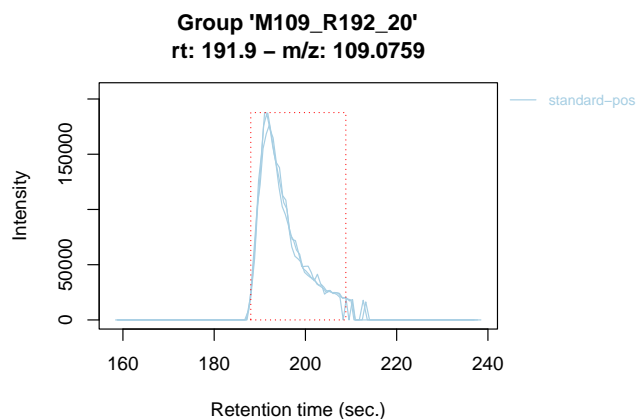
141 feature groups



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

```
#> Using 'mzr' backend for reading MS data.
```

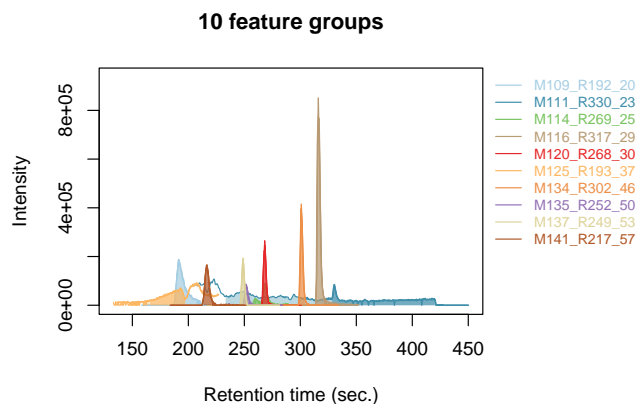
```
#> =====
```



```
plotChroms(fGroups[, 1:10], # only first ten feature groups
  groupBy = "fGroups", # group each feature group
  showPeakArea = TRUE, # show integrated areas
  showFGroupRect = FALSE) # no rectangles around feature groups
```

#> Using 'mzr' backend for reading MS data.

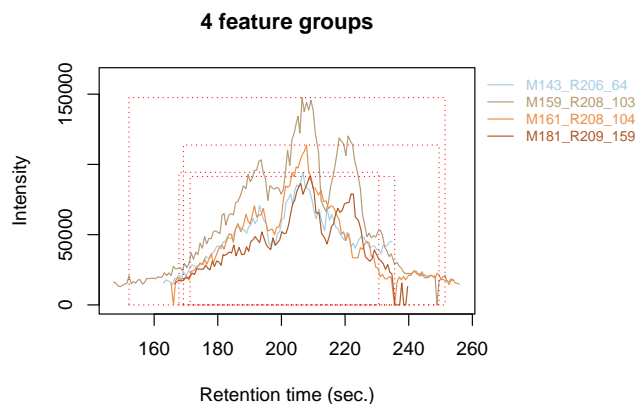
#> =====



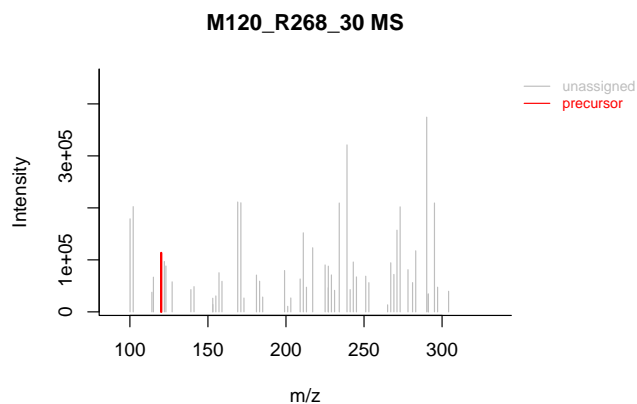
```
plotChroms(components, index = 7, fGroups = fGroups) # EICs from a component
```

#> Using 'mzr' backend for reading MS data.

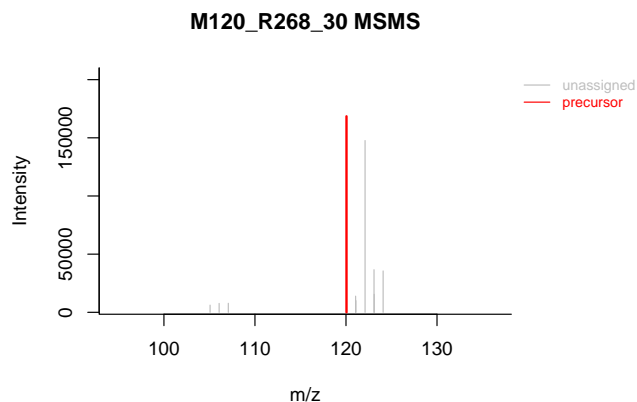
#> =====



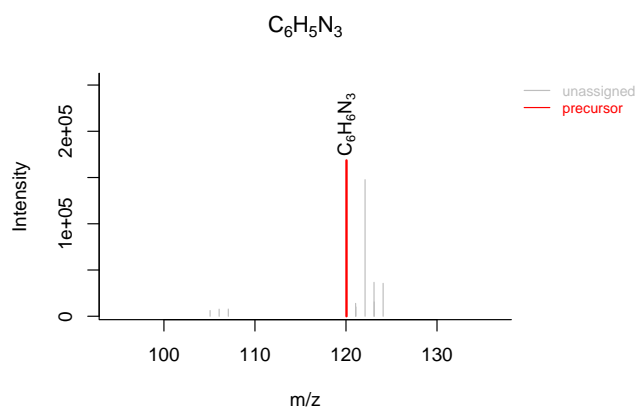
```
plotSpectrum(mslists, "M120_R268_30") # non-annotated MS spectrum
```



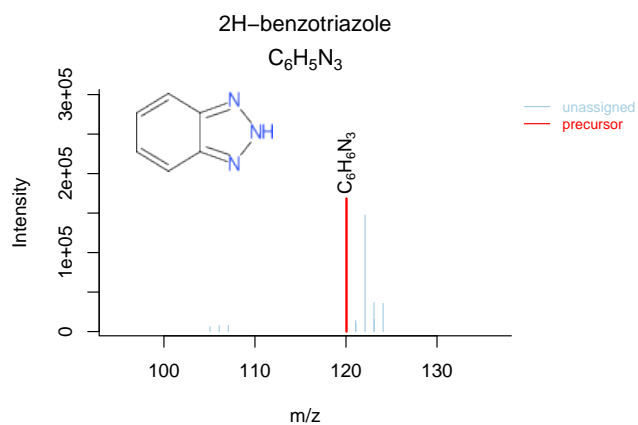
```
plotSpectrum(mslists, "M120_R268_30", MSLevel = 2) # non-annotated MS/MS spectrum
```



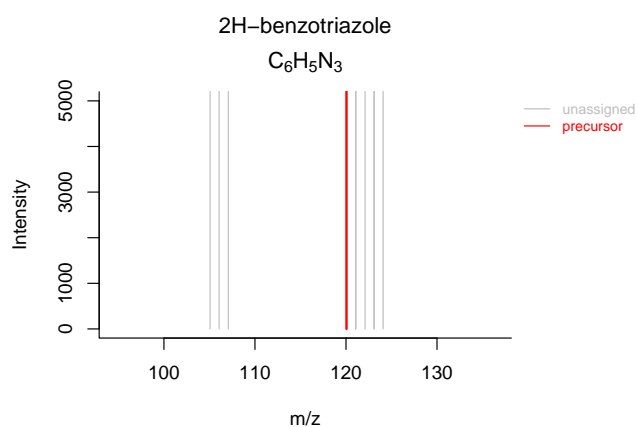
```
# formula annotated spectrum
plotSpectrum(formulas, index = 1, groupName = "M120_R268_30",
             MSPeakLists = mslists)
```



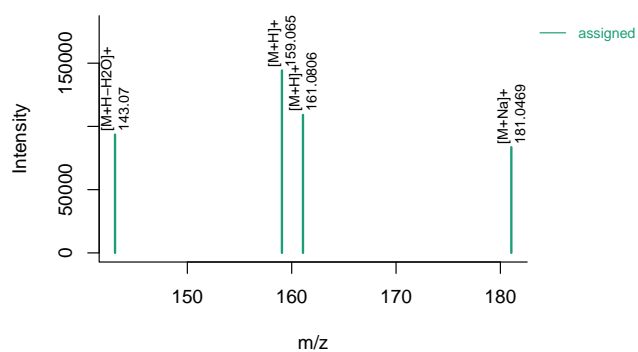
```
# compound annotated spectrum, with added formula annotations
plotSpectrum(compounds, index = 1, groupName = "M120_R268_30", MSPeakLists = mslists,
             formulas = formulas, plotStruct = TRUE)
```



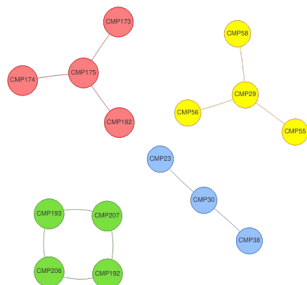
```
# custom intensity range (e.g. to zoom in)
plotSpectrum(compounds, index = 1, groupName = "M120_R268_30", MSPeakLists = mslists,
  ylim = c(0, 5000), plotStruct = FALSE)
```



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



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



5.9.1.1 Extracted Ion Chromatogram parameters The `EICParams` argument to `plotChroms()` is used to specify more advanced parameters for the creation of extracted ion chromatograms (EICs). Some parameters of interest:

Parameter	Description
<code>window</code>	Expands the EIC retention time range +/- the feature peak width (in seconds). This is e.g. useful to zoom out.
<code>topMost</code>	Only consider this amount of highest intensity features in a group.
<code>topMostByReplicate</code>	If <code>TRUE</code> then the <code>topMost</code> parameter concerns the top most intense features in a replicate (e.g. <code>topMost=1</code> would draw the most intense feature for each replicate).
<code>onlyPresent</code>	Only create EICs for analyses where a feature was detected? Setting to <code>FALSE</code> is useful to inspect if a feature was ‘missed’.

The parameters are configured by giving a named `list` to the `EICParams` argument. To obtain such a `list` with default settings, the `getDefEICParams()` function can be used:

```
getDefEICParams()
```

```
#> $topMost
#> NULL
#>
#> $topMostByReplicate
#> [1] FALSE
#>
#> $onlyPresent
#> [1] TRUE
#>
#> $mzExpWindow
#> [1] 0.001
#>
#> $mobExpWindow
#> [1] 0.01
#>
#> $mzExpMobWindow
#> [1] 0.005
#>
#> $minIntensityIMS
#> [1] 25
#>
```

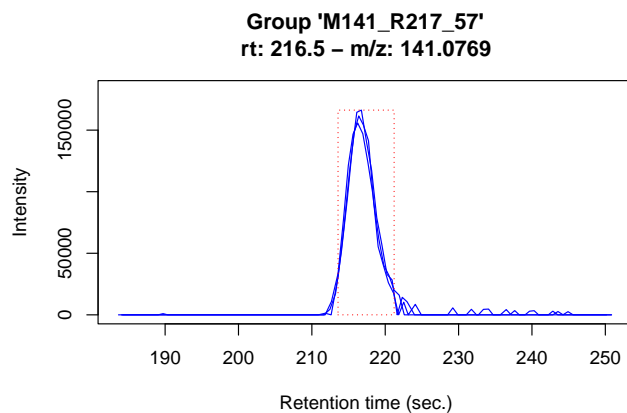
```
#> $setsAdductPos
#> [1] "[M+H]+"
#>
#> $setsAdductNeg
#> [1] "[M-H]-"
#>
#> $window
#> [1] 30
#>
#> $gapFactor
#> [1] 3
```

Any arguments specified to this function will alter the values of the returned parameter list. Some examples:

```
# investigate if any features were not detected in a feature group
plotChroms(fGroups[, 10], EICParams = getDefEICParams(onlyPresent = FALSE))
```

```
#> Using 'mzr' backend for reading MS data.
```

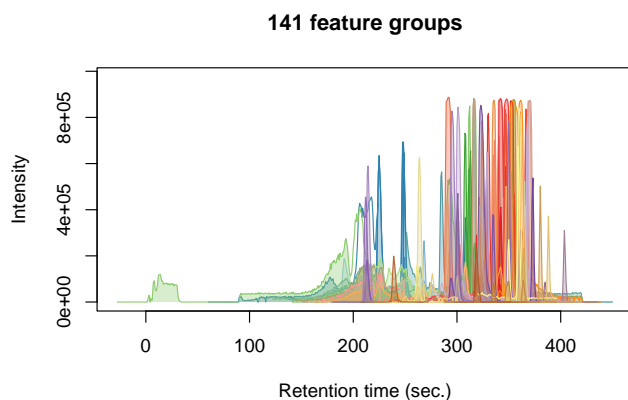
```
#> =====
```



```
# get overview of all feature groups
plotChroms(fGroups,
  groupBy = "fGroups", # unique colour for each group
  EICParams = getDefEICParams(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)
```

```
#> Using 'mzr' backend for reading MS data.
```

```
#> =====
```

The reference manual (`?EICParams`) gives a full detail on all parameters.

5.9.2 Overlapping and unique data

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

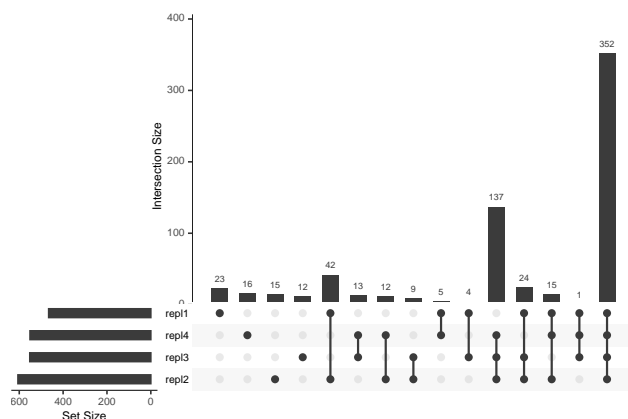
Generic	Classes
<code>plotVenn</code>	<code>featureGroups</code> , <code>featureGroupsComparison</code> , <code>formulas</code> , <code>compounds</code>
<code>plotUpSet</code>	<code>featureGroups</code> , <code>featureGroupsComparison</code> , <code>formulas</code> , <code>compounds</code>
<code>plotChord</code>	<code>featureGroups</code> , <code>featureGroupsComparison</code>

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.

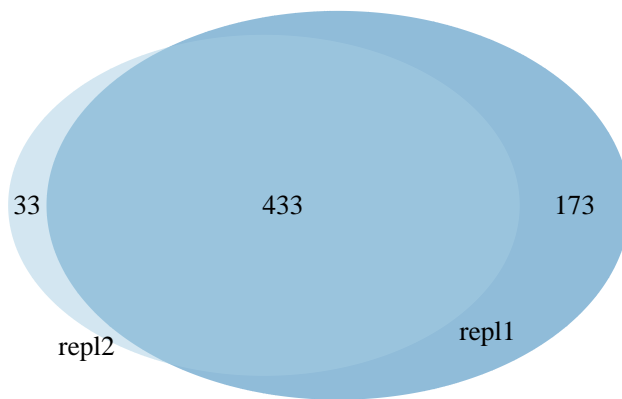
These functions can either be used to compare feature data or different objects of the same type. The former is typically used to compare overlap or uniqueness between features in different replicates, whereas comparison between objects is useful to visualize differences in algorithmic output. Besides 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.

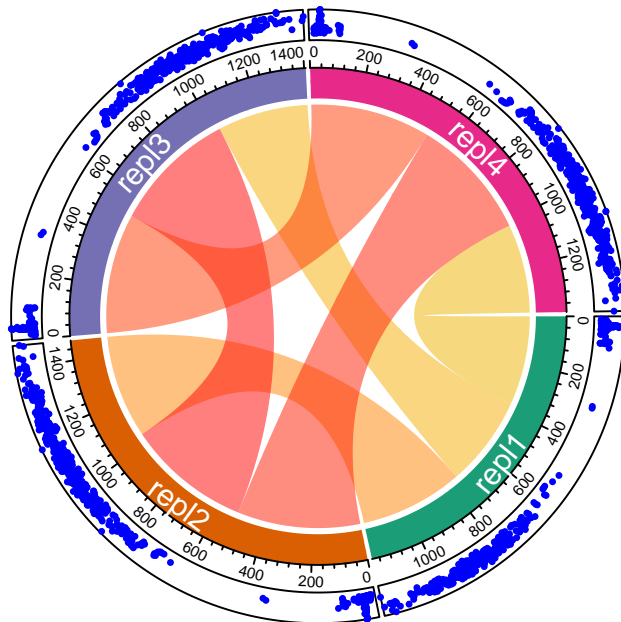
```
plotUpSet(fGroups) # compare replicates
```



```
plotVenn(fGroups, which = c("repl1", "repl2"), aggregate = TRUE) # compare some
↪ replicates
```

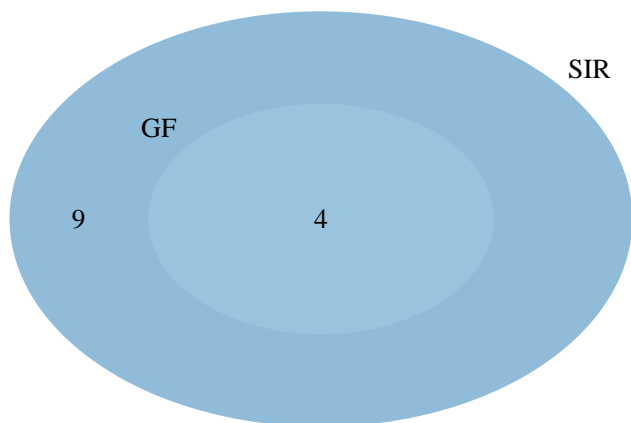


```
plotChord(fGroups, aggregate = TRUE) # overlap between replicates
```



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

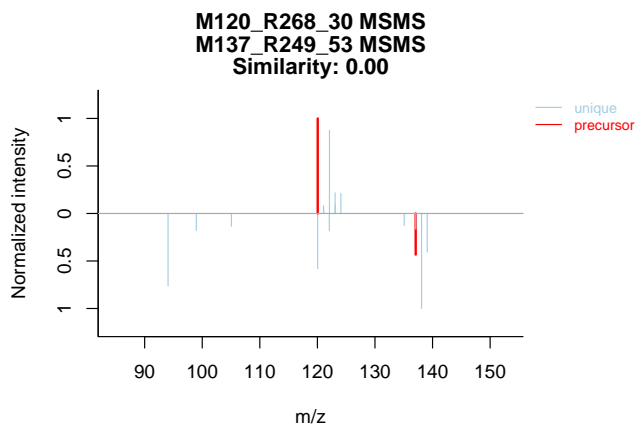
# compare GenForm and SIRIUS results
plotVenn(formsGF, formsSIR,
          labels = c("GF", "SIR")) # manual labeling
```



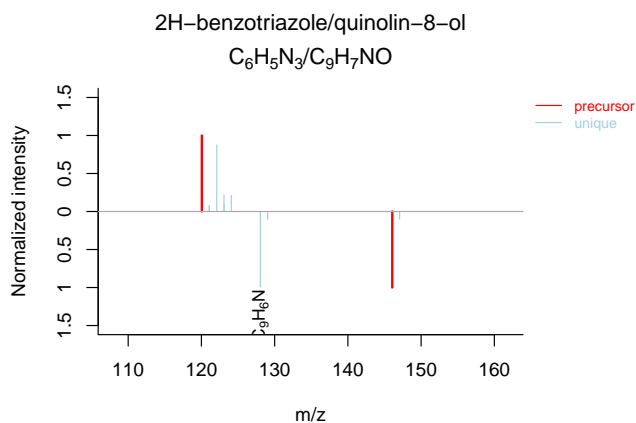
5.9.3 MS similarity

The `plotSpectrum` function is also useful to visually compare (annotated) spectra. This works for `MSPeakLists`, `formulas` and `compounds` object data.

```
plotSpectrum(mslists, groupName = c("M120_R268_30", "M137_R249_53"), MSLevel = 2)
```

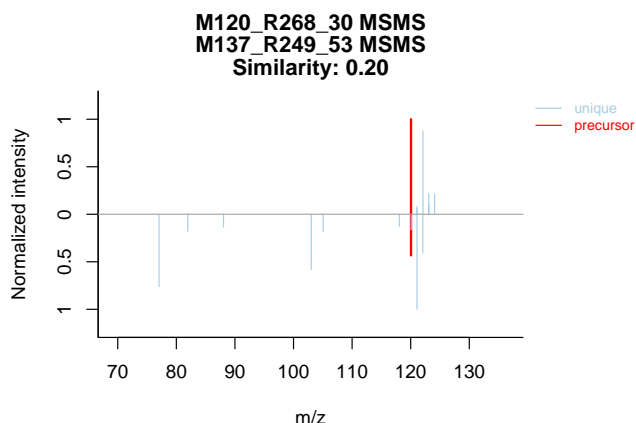


```
plotSpectrum(compounds, groupName = c("M120_R268_30", "M146_R309_68"), index = c(1, 1),  
             MSPeakLists = msls)
```



The `specSimParams` argument, which was discussed in MS similarity, can be used to configure the similarity calculation:

```
plotSpectrum(mslists, groupName = c("M120_R268_30", "M137_R249_53"), MSLevel = 2,
             specSimParams = getDefSpecSimParams(shift = "both"))
```

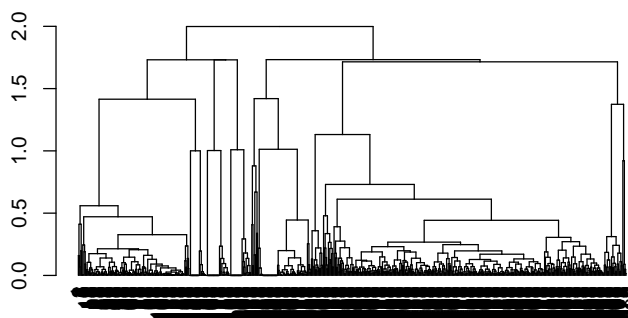


5.9.4 Hierarchical clustering results

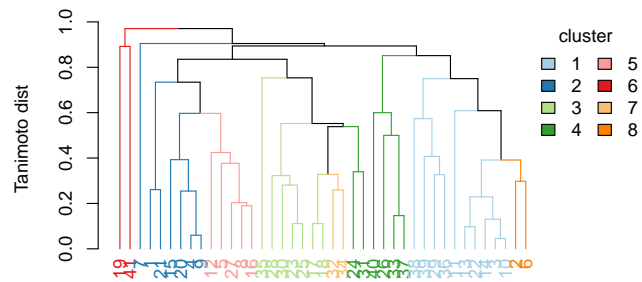
In `patRoan` hierarchical clustering is used for some componentization algorithms and to cluster candidate compounds with similar chemical structure (see compound clustering). The functions below can be used to visualize their results.

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

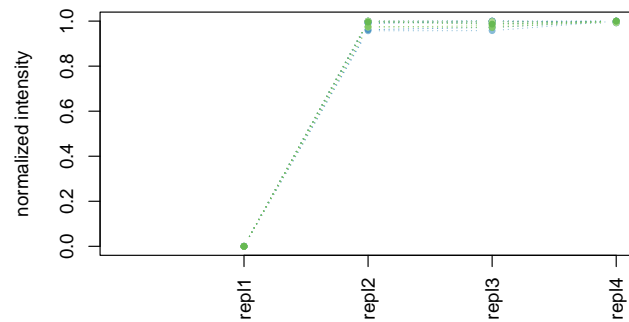
```
plot(componInt) # dendrogram
```



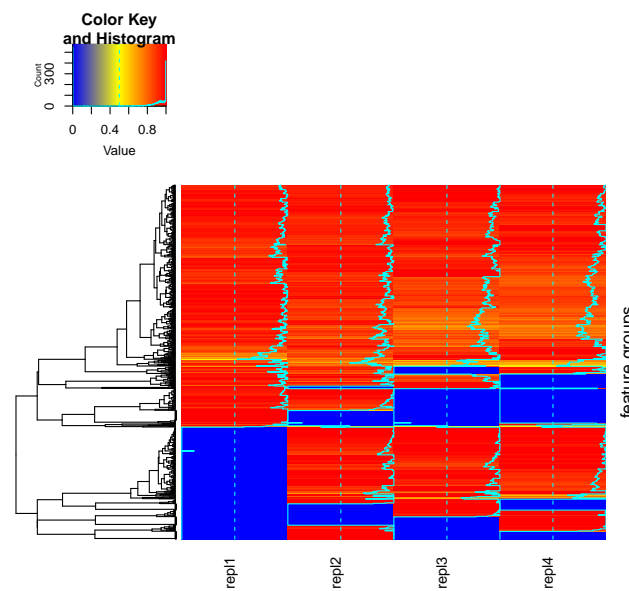
```
plot(compsClust, groupName = "M120_R268_30") # dendrogram for clustered compounds
```



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

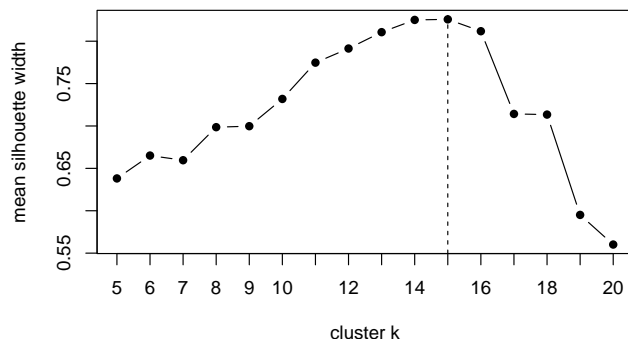


```
plotHeatMap(componInt) # plot heatmap
```



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

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



5.9.5 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
```

5.10 Use and modify sample analysis metadata

The analysis information section briefly introduced the incorporation of sample metadata. For instance, consider the following analysis information for a degradation experiment:

```
# create analysis information: for demonstrative purposes we just base it on the example
↪ data
anaInfoMeta <- anaInfo
anaInfoMeta$experiment <- c("UV", "UV", "H2O2", "H2O2", "O3", "O3")
anaInfoMeta$exposure <- c(0, 10, 20, 30, 0, 10) # time in minutes
anaInfoMeta[, c("analysis", "experiment", "exposure")]
```

```
#>      analysis experiment exposure
#> 1 solvent-pos-1      UV         0
#> 2 solvent-pos-2      UV        10
#> 3 solvent-pos-3    H2O2        20
#> 4 standard-pos-1    H2O2        30
#> 5 standard-pos-2      O3         0
#> 6 standard-pos-3      O3        10
```

As briefly mentioned before, the `analysisInfo()` function can be used to query the analysis information of a workflow object.

```
# fGroupsMeta is a feature groups object with the anaInfoMeta as analysis information
analysisInfo(fGroupsMeta[, c("analysis", "experiment", "exposure")])
```

```
#>      analysis experiment exposure
#>      <char>      <char>      <num>
#> 1: solvent-pos-1      UV         0
#> 2: solvent-pos-2      UV        10
#> 3: solvent-pos-3     H2O2        20
#> 4: standard-pos-1     H2O2        30
#> 5: standard-pos-2       O3         0
#> 6: standard-pos-3       O3        10
```

The `ni` argument can be used to easily subset a features and feature groups objects. It works very similar as subsetting a `data.table` with the `i` argument.

```
analyses(fGroupsMeta[ni = experiment == "H2O2"])
```

```
#> [1] "solvent-pos-3" "standard-pos-1"
```

```
analyses(fGroupsMeta[ni = exposure >= 20])
```

```
#> [1] "solvent-pos-3" "standard-pos-1"
```

```
# we can also use standard analysis information columns
analyses(fGroups[ni = replicate == "standard-pos"])
```

```
#> [1] "standard-pos-1" "standard-pos-2" "standard-pos-3"
```

The `anaInfoCols` argument can be used to add metadata to the output table generated by `as.data.table()` (and `as.data.frame()`).

```
as.data.table(fGroupsMeta, features = TRUE, anaInfoCols = c("experiment",
↪ "exposure"))[1:5, .(group, analysis, anaInfo_experiment, anaInfo_exposure)]
```

```
#>      group      analysis anaInfo_experiment anaInfo_exposure
#>      <char>      <char>      <char>          <num>
#> 1: M99_R14_1 solvent-pos-1      UV              0
#> 2: M99_R14_1 solvent-pos-2      UV             10
#> 3: M99_R14_1 solvent-pos-3     H2O2            20
#> 4: M99_R14_1 standard-pos-1     H2O2            30
#> 5: M99_R14_1 standard-pos-2       O3              0
```

5.10.1 Grouping and aggregating data

The analysis metadata can also be used to group and aggregate data. This is especially useful if you quickly want to compare different groups of analyses. The following functions support this:

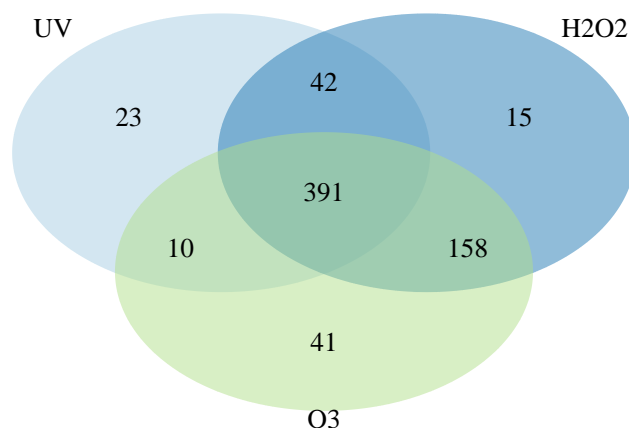
Functions	Argument	Remarks
<code>as.data.table()</code> , <code>plotInt()</code>	<code>average</code>	Averages intensities from analyses grouped by metadata.
<code>unique()</code> , <code>overlap()</code> , <code>plotVenn()</code> , <code>plotUpSet()</code> , <code>plotChord()</code>	<code>aggregate</code>	Compares analysis groups defined by metadata.
<code>plot()</code> , <code>plotInt()</code> , <code>plotChord()</code> , <code>plotChroms()</code>	<code>groupBy</code>	Groups data in plots.

Some examples are shown below.

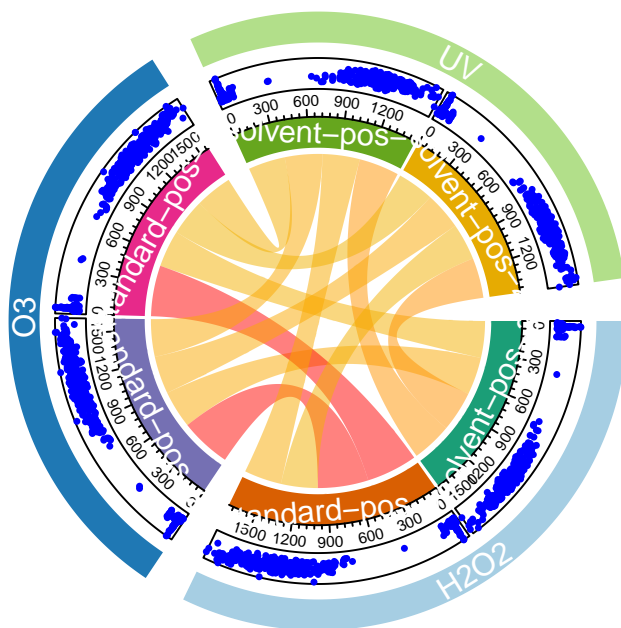
```
as.data.table(fGroupsMeta, average = "experiment")[1:5] # average intensities by
↪ experiment
```

```
#>      group      ret      mz UV_intensity H2O2_intensity O3_intensity
#>    <char>    <num>    <num>      <num>      <num>      <num>
#> 1: M99_R14_1 13.643769 98.97530    363518    325318    297774
#> 2: M99_R4_2  4.448981 98.97535    114428    144348    113840
#> 3: M100_R7_3 7.042372 100.11197    349810    283356    278576
#> 4: M100_R5_4 5.020052 100.11198         0         0      41400
#> 5: M100_R28_5 27.729320 100.11203    302680    303282    275468
```

```
plotVenn(fGroupsMeta, aggregate = "experiment") # compare feature presence by experiment
```



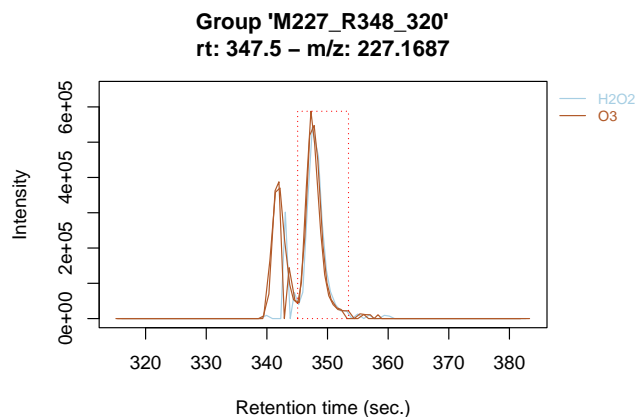
```
plotChord(fGroupsMeta, aggregate = FALSE, groupBy = "experiment") # group feature overlap
↪ by experiment
```

```
plotChroms(fGroupsMeta[, 320], groupBy = "experiment", showLegend = TRUE) # EICs grouped
  ↳ by experiment
```

```
#> Using 'mzr' backend for reading MS data.
```

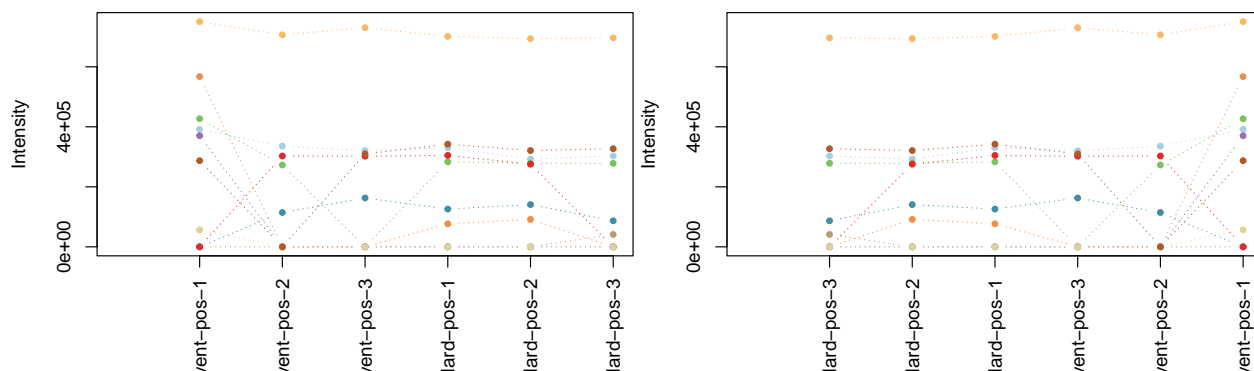
```
#> =====
```



5.10.2 Modifying analysis information

The analysis information in `features` and `featureGroups` objects can be modified with in two ways. The first is to use the subset operator (`[]`) to re-order the analyses within the object. This may be useful, for instance, to change the order in plots:

```
fGroupsToPlot <- fGroupsMeta[, 1:10]
plotInt(fGroupsToPlot, average = FALSE) # default order
fGroupsReverse <- fGroupsToPlot[rev(analyses(fGroupsToPlot)), reorder = TRUE]
plotInt(fGroupsReverse, average = FALSE) # reversed order
```



The second option is to modify the analysis information table itself by using the `analysisInfo()` assignment method function.

```
# add pH column
analysisInfo(fGroupsMeta)$pH <- c(7.0, 7.0, 3.5, 3.5, 2.0, 2.0)
analysisInfo(fGroupsMeta)[, .(analysis, experiment, exposure, pH)]
```

```
#>      analysis experiment exposure  pH
#>      <char>      <char>    <num> <num>
#> 1: solvent-pos-1      UV        0  7.0
#> 2: solvent-pos-2      UV       10  7.0
#> 3: solvent-pos-3     H2O2       20  3.5
#> 4: standard-pos-1     H2O2       30  3.5
#> 5: standard-pos-2       O3        0  2.0
#> 6: standard-pos-3       O3       10  2.0
```

```
# and remove it: take a copy, modify and replace it
tab <- data.table::copy(analysisInfo(fGroupsMeta))
tab[, pH := NULL] # remove pH column
analysisInfo(fGroupsMeta) <- tab
colnames(analysisInfo(fGroupsMeta)) # check if column is removed
```

```
#> [1] "analysis"      "path_centroid" "path_raw"      "path_profile"  "path_ims"      "replicate"
```

NOTE The analysis information is internally stored as a `data.table`. Do not attempt to modify it directly with the reference assignment operator (`:=`) of `data.table`. Instead, alter a copy and replace the table, as demonstrated by the second example above.

The `analysisInfo()` function can be used to add, remove or modify any metadata in the analysis information. It is also possible to change other columns such as the `replicate` or `blank` columns. However, it is currently not possible to modify the `analysis` column. Finally, the table rows can be re-ordered to change the order of analyses in the object.

5.11 Reporting

The previous sections showed various functionality to inspect and visualize results. An easy way to do this automatically is by using the *reporting* functionality of `patRoan`. There are currently two interfaces: a legacy interface that is described in the next subsection, and the modernized version discussed here.

The reports are generated by the `report()` function, which combines the data generated during the workflow. This function outputs an HTML file (other formats may follow in future versions), which can be opened with a regular web browser to interactively explore and visualize the data. The report combines chromatograms, mass spectra, tables with feature and annotation properties and many other useful ways to easily explore your data.

Which data is reported is controlled by the following function arguments:

Argument	Description
<code>fGroups</code>	The <code>featureGroups</code> object that should be reported (mandatory).
<code>MSPeakLists</code>	The MS peak lists object used for annotations (mandatory if <code>formulas/compounds</code> are specified).
<code>formulas, compounds</code>	The <code>formulas</code> and <code>compounds</code> objects that should be used to report feature annotations.
<code>compsCluster</code>	The result object from compound clustering.
<code>components</code>	Any componentization results, <i>e.g.</i> with adduct annotations and from transformation product screening.
<code>TPs</code>	Output from object from generated transformation products.

Most of these arguments are optional, and if not specified this part of the workflow is simply not reported. This also 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. More advanced arguments to `report()` are discussed in the reference manual ([?reporting](#)).

Some examples:

```
report(fGroups) # only report feature groups
report(fGroups[, 1:50]) # same, but only first 50, e.g. to do a quick inspection

# include feature annotations
report(fGroups, MSPeakLists = mslists, formulas = formulas, compounds = compounds)

# TP screening
report(fGroups, MSPeakLists = mslists, formulas = formulas, compounds = compounds,
       components = componentsTP, TPs = TPs)
```

The report itself is primarily configured through a *report settings file*, which is an easily editable YAML file. The default file is as follows:

```
general:
  version: 3
  format: html
  path: report
  keepUnusedPlots: 7
  selfContained: false
  noDate: false
summary: [ chord, venn, upset ]
features:
  retMin: true
  chromatograms:
    large: true
    small: true
    features: false
```

```

    intMax: eic
  mobilograms:
    large: true
    small: true
    features: false
  intensityPlots: false
  aggregateConcs: mean
  aggregateTox: mean
MSPeakLists:
  spectra: true
formulas:
  include: true
  normalizeScores: max
  exclNormScores: [ ]
  topMost: 25
compounds:
  normalizeScores: max
  exclNormScores: [ score, individualMoNAScore, annoTypeCount, annotHitCount, libMatch
↪ ]
  onlyUsedScorings: true
  topMost: 25
TPs:
  graphs: true
  graphStructuresMax: 25
internalStandards:
  graph: true

```

A detailed description for all the settings can be found in the reference manual ([?reporting](#)). The table below summarizes the most interesting options:

Parameter	Description
<code>general --> format</code>	The output format. Currently only "html".
<code>general --> selfContained</code>	If set (<code>true</code>) then the output will be a self contained .html file. Handy to share reports, but not recommended for large amounts of data.
<code>features --> chromatograms --> features</code>	If enabled (<code>true</code>) then the report includes chromatograms of individual features. If set to <code>all</code> then also chromatograms are generated for analyses in which a feature was not detected. This is especially useful to inspect if features were 'missed' during feature detection or accidentally removed after a filter step.
<code>formulas/compounds --> topMost</code>	Specifies the maximum number of top-ranked candidates to plot. Often it will take a considerable amount of time to report all candidates, hence, by default this is limited.

When the `newProject` tool is used to create a new `patRoan` project a template settings file (`report.yml`) is automatically created. Otherwise, this file can be generated with the `genReportSettingsFile()` function. Simply running this function without any arguments is enough:

```
genReportSettingsFile()
```

5.11.1 Legacy interface

The legacy interface was the default reporting interface for **patRoön** versions older than 2.2. The interface now mainly serves for backward compatibility reasons, but may still be useful since the new interface does not (yet) support all the formats from the legacy interface. 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)

Like the **report()** function described above, the arguments to these functions control which data will be reported. However, these functions do not use a report settings file, and all configuration happens through function arguments. Some common arguments are listed below; for a complete listing see the reference manual (`?reporting-legacy`).

Argument	Functions	Remarks
fGroups , formulas , compounds , formulas , components , compsCluster , TPs	All	Objects to plot. Only fGroups is mandatory.
MSPeakLists	reportPDF()	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()	Report no more than this amount of highest ranked candidates.

Some typical examples:

```
reportPDF(fGroups) # only report 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
reportPDF(fGroups, formulas = formulas, compounds = compounds,
          components = components, MSPeakLists = mslists,
          compsCluster = compsClust,
          compoundsTopMost = 10)
```

6 Sets workflows

In LC-HRMS screening workflows it is typically desired to be able to detect a broad range of chemicals. For this reason, the samples are often measured twice: with positive and negative ionization. Most data processing steps are only suitable for data with the same polarity, for instance, due to the fact that the m/z values in mass spectra are inherently different (e.g. $[M+H]^+$ vs $[M-H]^-$) and MS/MS fragmentation occurs

differently. As a result, the screening workflow has to be done twice, which generally requires more time and complicates comparing and interpretation of the complete (positive and negative) dataset.

In **patRoön** version 2.0 the *sets workflow* is introduced. This allows you to perform a single non-target screening workflow from different *sets* of analyses files. Most commonly, each set represents a polarity, hence, there is a positive and negative set. However, more than two sets are supported, and other distinctions between sets are also possible, for instance, samples that were measured with different MS/MS techniques. Another important advantage of the sets workflow is that MS/MS data from different sets can be combined to provide more comprehensive annotations of features. The most important limitation is that (currently) the chromatographic method that was used when analyzing the samples from each set needs to be equal, since retention times are used to group features among the sets.

Performing a sets workflow usually only requires small modifications compared to a ‘regular’ **patRoön** workflow. This chapter outlines how to perform such workflows and how to use its unique functionality for data processing. It is assumed that the reader is already familiar with performing ‘regular’ workflows, which were discussed in the previous chapters.

6.1 Initiating a sets workflow

A sets workflow is not much different than a ‘regular’ (or non-sets) workflow. For instance, consider the following workflow:

```
anaInfo <- patRoönData::exampleAnalysisInfo("positive")
fList <- findFeatures(anaInfo, "openms")
fGroups <- groupFeatures(fList, "openms")
fGroups <- filter(fGroups, absMinIntensity = 10000, relMinReplicateAbundance = 1,
  ↪ maxReplicateIntRSD = 0.75,
      blankThreshold = 5, removeBlanks = TRUE)

mslists <- generateMSPeakLists(fGroups)
formulas <- generateFormulas(fGroups, mslists, "genform", adduct = "[M+H]+")
compounds <- generateCompounds(fGroups, mslists, "metfrag", adduct = "[M+H]+")

report(fGroups, MSPeakLists = mslists, formulas = formulas, compounds = compounds)
```

This example uses the example data from **patRoönData** to obtain a feature group dataset, which is cleaned-up afterwards. Then, feature groups are annotated and all the results are reported.

Converting this to a *sets workflow*:

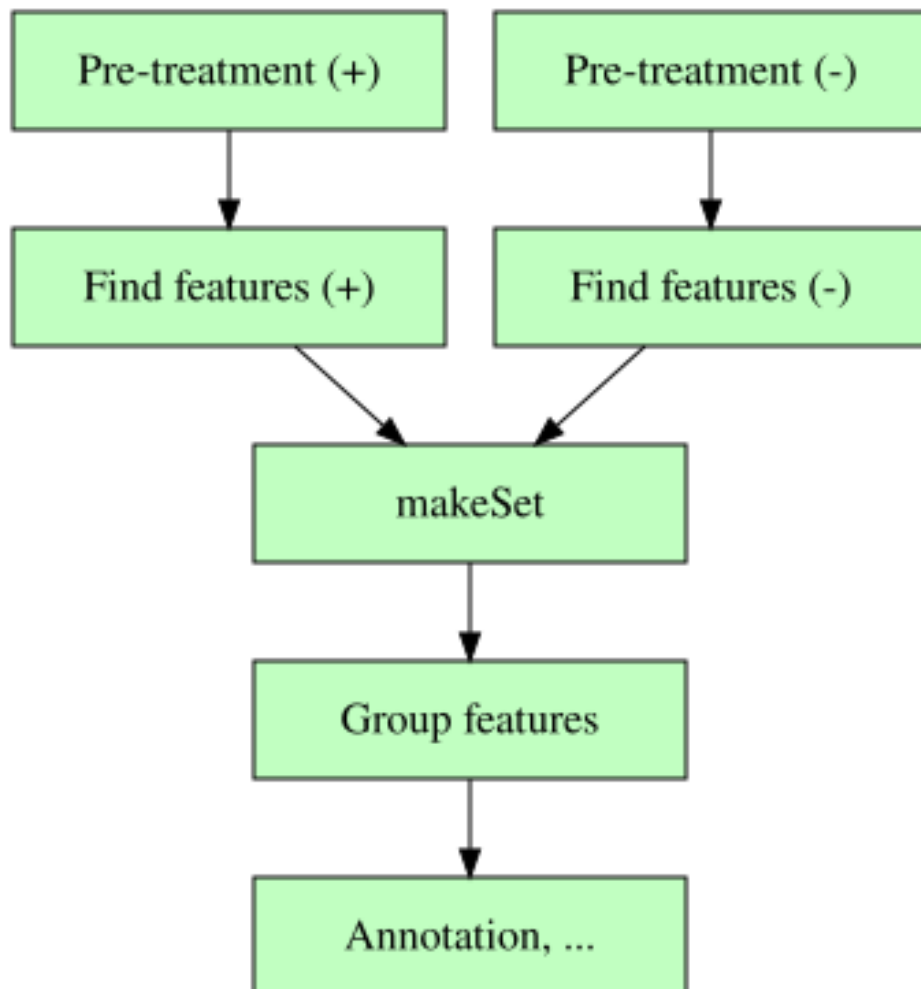
```
anaInfoPos <- patRoönData::exampleAnalysisInfo("positive")
anaInfoNeg <- patRoönData::exampleAnalysisInfo("negative")
fListPos <- findFeatures(anaInfoPos, "openms")
fListNeg <- findFeatures(anaInfoNeg, "openms")
fList <- makeSet(fListPos, fListNeg, adducts = c("[M+H]+", "[M-H]-"))

fGroups <- groupFeatures(fList, "openms")
fGroups <- filter(fGroups, absMinIntensity = 10000, relMinReplicateAbundance = 1,
  ↪ maxReplicateIntRSD = 0.75,
      blankThreshold = 5, removeBlanks = TRUE)

mslists <- generateMSPeakLists(fGroups)
formulas <- generateFormulas(fGroups, mslists, "genform")
compounds <- generateCompounds(fGroups, mslists, "metfrag")
```

```
report(fGroups, MSPeakLists = mslists, formulas = formulas, compounds = compounds)
```

This workflow will do all the steps for positive *and* negative data.



Only a few modifications were necessary:

- The analysis information is obtained for positive and negative data (i.e. per set)
- Features are found for each set separately.
- `makeSet` is used to combine the feature data
- There is no need to specify the adduct anymore in the annotation steps.

NOTE The **analysis** names for the analysis information must be *unique* for each row, even among sets. Furthermore, replicates should not contain analyses from different sets.

The key principle to make sets workflows work is performed by `makeSet`. This method function takes different **features** objects (or **featureGroups**, discussed later) to combine the feature data across sets. During this step features are *neutralized*: the feature m/z data is converted to neutral feature masses. This step ensures that when features are grouped with `groupFeatures`, its algorithms are able to find the same feature among different sets, even when different MS ionization modes were used during acquisition. However, please note

that (currently) no additional chromatographic alignment steps between sets are performed. For this reason, the chromatographic methodology that is used to acquire the data must be the same for all sets.

The feature neutralization step relies on adduct data. In the example above, it is simply assumed that all features measured with positive mode are protonated (M+H) species, and all negative features are deprotonated (M-H). It is also possible to use adduct annotations for neutralization; this is discussed later.

NOTE The newProject tool can be used to easily generate a sets workflow. Simply select “both” for the *Ionization* option.

6.2 Generating sets workflow data

As was shown in the previous section, the generation of workflow data with a sets workflow largely follows that as what was discussed in the previous chapters. The same generator functions are used:

Workflow step	Function	Output S4 class
Grouping features	<code>groupFeatures()</code>	<code>featureGroupsSet</code>
Suspect screening	<code>screenSuspects()</code>	<code>featureGroupsScreeningSet</code>
MS peak lists	<code>generateMSPeakLists()</code>	<code>MSPeakListsSet</code>
Formula annotation	<code>generateFormulas()</code>	<code>formulasSet</code>
Compound annotation	<code>generateCompounds()</code>	<code>compoundsSet</code>
Componentization	<code>generateComponents()</code>	algorithm dependent

(the data pre-treatment and feature finding steps have been omitted as they are not specific to sets workflows).

While the same function generics are used to generate data, the class of the output objects differ (e.g. `formulasSet` instead of `formulas`). However, since all these classes *inherit* from their non-sets workflow counterparts, using the workflow data in a sets workflow is nearly identical to what was discussed in the previous chapters (further discussed in the next section).

As discussed before, an important step is the neutralization of features. Other workflow steps also have internal mechanics to deal with data from different sets:

Workflow step	Handling of set data
Finding/Grouping features	Neutralization of m/z values
Suspect screening	Merging results from screening performed for each set
Componentization	Algorithm dependent (discussed below)
MS peak lists	MS data is obtained and stored per set. The final peak lists are combined (<i>not</i> averaged)
Formula/Compound annotation	Annotation is performed for each set separately and used to generate a final consensus

In most cases the algorithms of the workflow steps are first performed for each set, and this data is then merged. To illustrate the importance of this, consider these examples

- A suspect screening with a suspect list that contains known MS/MS fragments
- Annotation where MS/MS fragments are used to predict the chemical formula
- Componentization in order to establish adduct assignments for the features

In all cases data is used that is highly dependent on the MS method (eg polarity) that was used to acquire the sample data. Nevertheless, all the steps needed to obtain and combine set data are performed automatically in the background, and are therefore largely invisible.

NOTE Because feature groups in sets workflows always have adduct annotations, it is never required to specify the adduct or ionization mode when generating annotations, components or do suspect screening (*i.e.* the `adduct/ionization` arguments should not be specified).

6.2.1 Componentization

When the componentization algorithms related to adduct/isotope annotations (e.g. CAMERA, RAMClustR and cliqueMS) and nontarget are used, then componentization occurs per set and the final object (a `componentsSet` or `componentsNTSet`) contains all the components together. Since these algorithms are highly dependent upon MS data polarity, no attempt is made to merge components from different sets.

The other componentization algorithms work on the complete data. For more details, see the reference manual (`?generateComponents`).

6.2.2 Formula and compound annotation

For formula and compound annotation, the data generated for each set is combined to generate a *set consensus*. The annotation tables are merged, scores are averaged and candidates are re-ranked. More details can be found in the reference manual (e.g. `?generateCompounds`). In addition, it is possible to only keep candidates that exist in a minimum number of sets. For this, the `setThreshold` and `setThresholdAnn` argument can be used:

```
# candidate must be present in all sets
formulas <- generateFormulas(fGroups, mslists, "genform", setThreshold = 1)
# candidate must be present in all sets with annotation data
compounds <- generateCompounds(fGroups, mslists, "metfrag", setThresholdAnn = 1)
```

In the first example, a formula candidate for a feature group is only kept if it was found for all of the sets. In the second example, a compound candidate is only kept if it was present in all of the sets with annotation data available. The following examples of a common positive/negative sets workflow illustrate the differences:

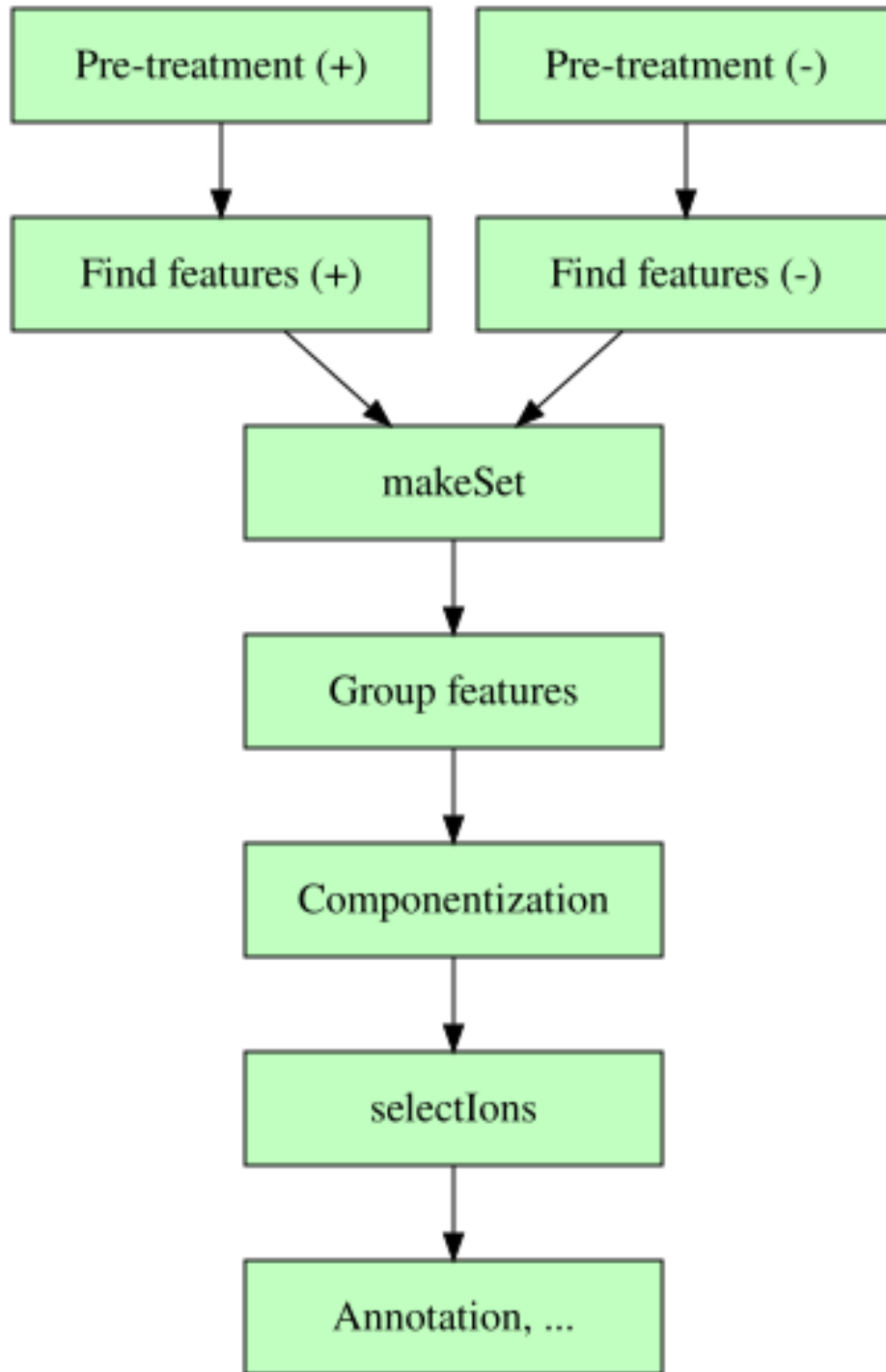
Candidate	annotations	candidate present	setThreshold=1	setThresholdAnn=1
#1	+, -	+, -	Keep	Keep
#2	+, -	+	Remove	Remove
#3	+	+	Remove	Keep

For more information refer to the reference manual (e.g. `?generateCompounds`).

6.3 Selecting adducts to improve grouping

The `selectIons()` and `adduct()` functions discussed before can also improve sets workflows. This is because the adduct annotations can be used to improve feature neutralization, which in turn will improve grouping features between positive and negative ionization data. Once adduct annotations are set the features will be re-neutralized and re-grouped.

A typical workflow with `selectIons` looks like this:



```
# as before ...
anaInfoPos <- patRoonData::exampleAnalysisInfo("positive")
anaInfoNeg <- patRoonData::exampleAnalysisInfo("negative")
fListPos <- findFeatures(anaInfoPos, "openms")
fListNeg <- findFeatures(anaInfoNeg, "openms")

fGroupsPos <- groupFeatures(fListPos, "openms")
fGroupsNeg <- groupFeatures(fListNeg, "openms")
```

```
fList <- makeSet(fListPos, fListNeg, adducts = c("[M+H]+", "[M-H]-"))

fGroups <- groupFeatures(fList, "openms")
fGroups <- filter(fGroups, absMinIntensity = 10000, relMinReplicateAbundance = 1,
  ↳ maxReplicateIntRSD = 0.75,
    blankThreshold = 5, removeBlanks = TRUE)

components <- generateComponents(fGroups, "openms")
fGroups <- selectIons(fGroups, components, c("[M+H]+", "[M-H]-"))

# do rest of the workflow...
```

The first part of the workflow is exactly the same as was introduced in the beginning of this chapter. Furthermore, note that for sets workflows, `selectIons` needs a preferential adduct for each set.

The `adducts` function can also be used to obtain and modify adduct annotations. For sets workflows, these functions operate *per set*:

```
adducts(fGroups, set = "positive")[1:5]
adducts(fGroups, set = "positive")[4] <- "[M+K]+"
```

If you want to modify annotations for multiple sets, it is best to delay the re-grouping step:

```
adducts(fGroups, set = "positive", reGroup = FALSE)[4] <- "[M+K]+"
adducts(fGroups, set = "negative", reGroup = TRUE)[10] <- "[M-H2O]-"
```

Setting `reGroup=FALSE` will not perform any re-neutralization and re-grouping, which preserves feature group names and saves processing time. However, it is **crucial** that the re-grouping step is eventually performed at the end.

6.4 Processing data

All data objects that are generated during a sets workflow *inherit* from the classes from a ‘regular’ workflow. This means that, with some minor exceptions, *all* of the data processing functionality discussed in the previous chapter (e.g. subsetting, inspection, filtering, plotting, reporting) is also applicable to a sets workflow. In addition, data from sets workflows also bring some additional data processing functionality. Some examples:

```
# only keep feature groups that have positive data
fGroupsPos <- fGroups[, sets = "positive"]
# only keep feature groups with features present in all sets
fGroupsF <- filter(fGroups, relMinSets = 1)
```

```
#> Applying minimum sets filter... Done! Filtered 3407 (88.84%) features and 809 (94.07%) feature groups
```

```
# In sets workflows, the m/z values of features are 'neutralized', the `ion_mz` columns
  ↳ contains the original 'ionized' m/z values.
as.data.table(fGroups)[1:5, c("group", "mz", "ion_mz-positive", "ion_mz-negative")]
```

```
#>      group      mz ion_mz-positive ion_mz-negative
#>      <char>    <num>          <num>          <num>
```

```
#> 1: M98_R7_1 97.96702      NA      96.95974
#> 2: M98_R30_2 97.96708      NA      96.95981
#> 3: M98_R10_3 97.96709      NA      96.95981
#> 4: M98_R5_4 97.96769     98.97497     96.96042
#> 5: M98_R14_5 97.96787     98.97515     96.96060
```

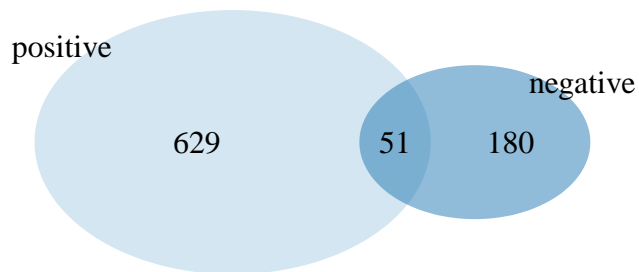
```
# Inspect set specific data.
as.data.table(compounds)[1:5, c("group", "score-positive", "score-negative",
  ↪  "compoundName", "set")]
```

```
#>           group score-positive score-negative      compoundName
#>      <char>      <num>      <num>      <char>
#> 1: M198_R317_273    3.5190115    4.569478 3-(4-chlorophenyl)-1,1-dimethylurea posi
#> 2: M198_R317_273    2.5198763    1.563191 5-[[ (2R)-azetidin-2-yl]methoxy]-2-chloropyridine posi
#> 3: M198_R317_273    1.2528529    1.350556 1-(3-chloro-4-methylphenyl)-3-methylurea posi
#> 4: M198_R317_273    1.1469202    1.276057 3-(3-chlorophenyl)-1,1-dimethylurea posi
#> 5: M198_R317_273    0.9981602    1.127297 1-(4-chlorophenyl)-3-ethylurea posi
```

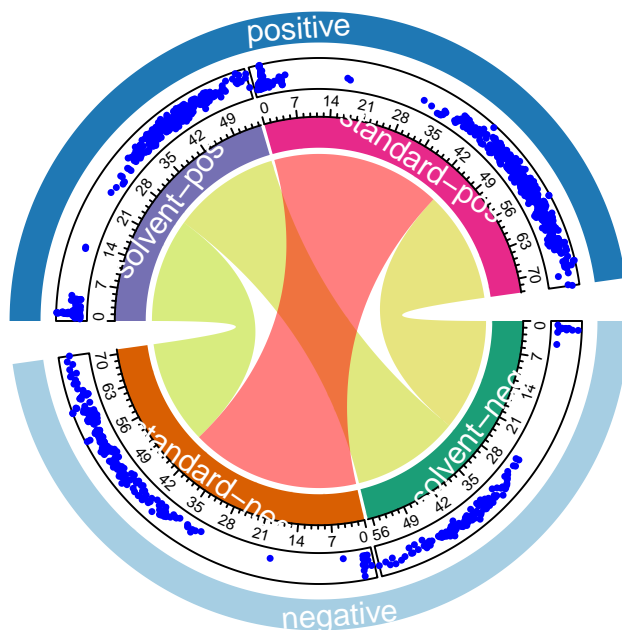
In sets workflows the analysis information is amended with a `set` column to specify the set each analysis belongs to. Just like other columns in the analysis information, the `set` column can be used to group and aggregate data:

```
# only keep feature groups with features present in both polarities
fGroupsPosNeg <- overlap(fGroups, which = c("positive", "negative"), aggregate = "set")
# only keep feature groups with features that are present only in positive mode
fGroupsOnlyPos <- unique(fGroups, which = "positive", aggregate = "set")

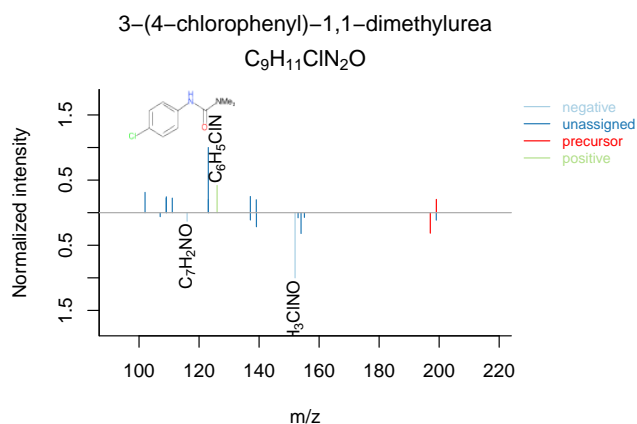
plotVenn(fGroups, aggregate = "set", margin = 0.1) # compare positive/negative features
```



```
plotChord(fGroups, aggregate = TRUE, groupBy = "set") # compare replicate aggregated
  ↪ positive/negative features
```



```
# plot annotated positive/negative mirror spectrum
plotSpectrum(compounds, index = 1, groupName = "M198_R317_273", MSPeakLists = mslists,
              plotStruct = TRUE)
```



The reference manual for the workflow objects contains specific notes applicable to sets workflows (?featureGroups, ?compounds etc).

6.5 Advanced

6.5.1 Initiating a sets workflow from feature groups

The makeSet function can also be used to initiate a sets workflow from feature groups:

```
# as before ...
anaInfoPos <- patRoonData::exampleAnalysisInfo("positive")
anaInfoNeg <- patRoonData::exampleAnalysisInfo("negative")
fListPos <- findFeatures(anaInfoPos, "openms")
fListNeg <- findFeatures(anaInfoNeg, "openms")
```

```
fGroupsPos <- groupFeatures(fListPos, "openms")
fGroupsNeg <- groupFeatures(fListNeg, "openms")

fGroups <- makeSet(fGroupsPos, fGroupsNeg, groupAlgo = "openms",
                  adducts = c("[M+H]+", "[M-H]-"))

# do rest of the workflow...
```

In this case `makeSet` takes the positive and negative features, neutralizes them and creates new feature groups by grouping the original set specific groups (with the algorithm specified by `groupAlgo`).

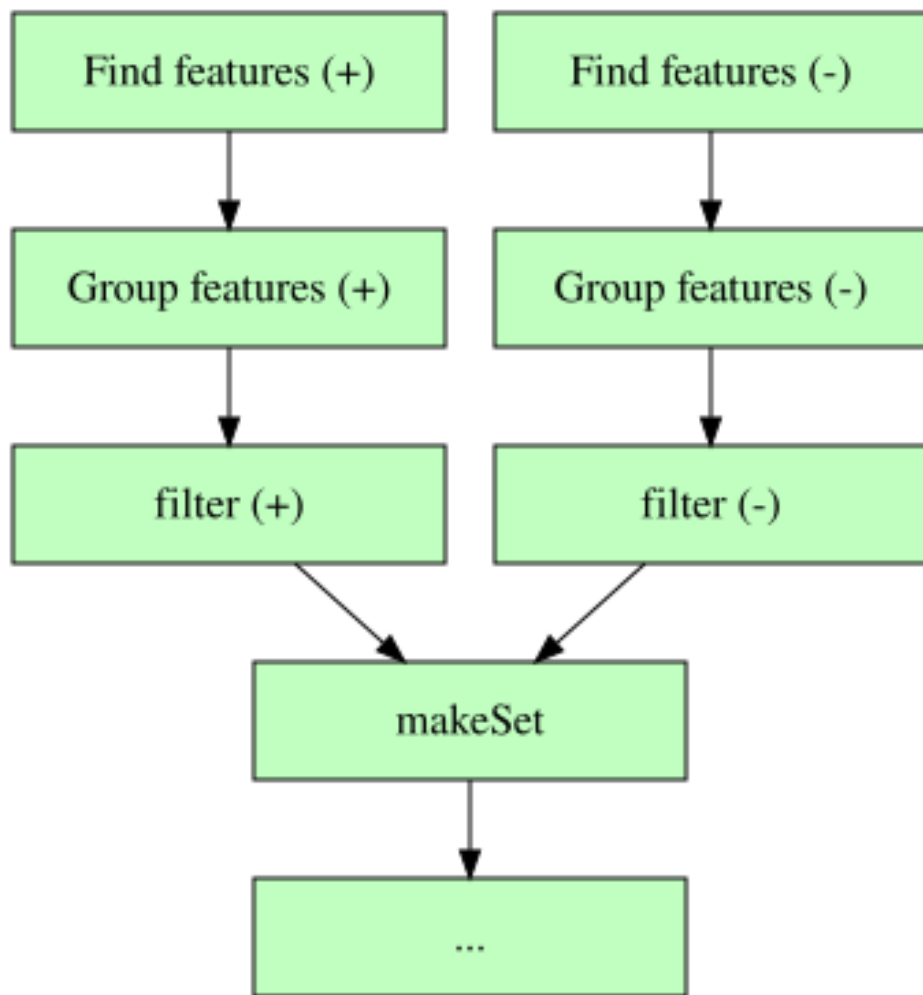
While this option involves some extra steps, an advantage is that allows processing the feature data before they are combined, e.g.:

```
fGroupsPos <- groupFeatures(fListPos, "openms")
fGroupsNeg <- groupFeatures(fListNeg, "openms")

# apply intensity threshold filters. Lower threshold for negative.
fGroupsPos <- filter(fGroupsPos, absMinIntensity = 1E4)
fGroupsNeg <- filter(fGroupsNeg, absMinIntensity = 1E3)

fGroups <- makeSet(fGroupsPos, fGroupsNeg, groupAlgo = "openms",
                  adducts = c("[M+H]+", "[M-H]-"))
```

Visually, this workflow looks like this:



Of course, any other processing steps on the feature groups data such as subsetting and visually checking features are also possible before the sets workflow is initiated. Furthermore, it is also possible to perform adduct annotations prior to grouping, which is an alternative way to improve neutralization to what was discussed before.

6.5.2 Inspecting and converting set objects

The following generic functions may be used to inspect or convert data from sets workflows:

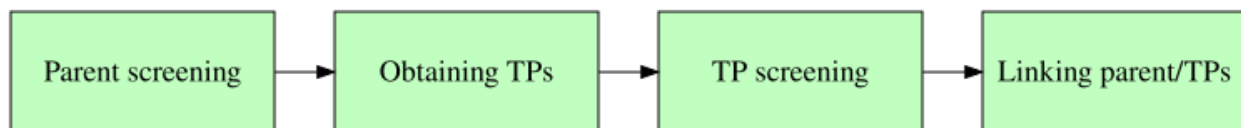
Generic	Purpose	Notes
<code>sets</code>	Return the names of the sets in this object.	
<code>setObjects</code>	Obtain the raw data objects that were used to construct this object.	Not available for features and feature groups.
<code>unset</code>	Converts this object to a regular workflow object.	The <code>set</code> argument must be given to specify which of the set data is to be converted. This function will restore the original m/z values of features.

These methods are heavily used internally, but rarely needed otherwise. More details can be found in the

7 Transformation product screening

This chapter describes the various functionality for screening of *transformation products* (TPs), which are introduced since **patRoön** 2.0. Screening for TPs, i.e. chemicals that are formed from a *parent* chemical by e.g. chemical or biological processes, has broad applications. For this reason, the TP screening related functionality is designed to be flexible, thus allowing one to use a workflow that is best suited for a particular study.

Regardless, the TP screening workflow in **patRoön** can be roughly summarized as follows:



- **Parent screening** During this step a common **patRoön** workflow is used to screen for the parent chemicals of interest. This could be a full non-target analysis with compound annotation or a relative simple suspect or target screening.
- **Obtaining TPs** Data is obtained of potential TPs for the parents of interest. The TPs may originate from a library or predicted *in-silico*. Note that in some workflows this step is omitted (discussed later).
- **TP screening** A suspect screening is performed to find the TPs in the analysis data. Some workflows skip this step.
- **Linking parents and TPs** In the step the parent features are linked with the TP features. Several post-processing functionality exists to improve and prioritize the data.

The next sections will outline more details on these steps are performed and configured. The last section in this chapter outlines several example workflows.

NOTE The `newProject` tool can be used to easily generate a workflow with transformation product screening.

7.1 Obtaining transformation product data

The `generateTPs` function is used to obtain TPs for a particular set of parents. Like other workflow generator functions (`findFeatures`, `generateCompounds`), several algorithms are available that do the actual work.

Algorithm	Usage	Input	Output
BioTransformer	<code>generateTPs(algorithm = "biotransformer", ...)</code>	Parents	TPs structural information
CTS	<code>generateTPs(algorithm = "cts", ...)</code>	Parents	TPs with structural information
Library	<code>generateTPs(algorithm = "library", ...)</code>	Parents (optional), Library (PubChem or custom)	TPs with structural information
Formula library	<code>generateTPs(algorithm = "library_formula", ...)</code>	Parents	TPs with formula information

Algorithm	Usage	Input	Output
Metabolic logic	<code>generateTPs(algorithm = "logic", ...)</code>	Feature groups	TPs from m/z differences from pre-defined elemental transformations (based on Schollee et al. (2015)).
Formula Annotations	<code>generateTPs(algorithm = "ann_form", ...)</code>	Parents, formula annotations	Prioritized TP from annotation candidates (based on Helmus et al. (2025)).
Compound Annotations	<code>generateTPs(algorithm = "ann_comp", ...)</code>	Parents, compound annotations	Prioritized TP from annotation candidates (based on Helmus et al. (2025)).

For most workflows the `biotransformer`, `cts` and `library` algorithms are a good starting point. They are fairly straightforward to use and output TP with full structural information. The `library_formula` algorithm may be a suitable alternative if only parents with formula information are available. The `ann_comp` and `ann_form` algorithms are meant to elucidate completely unknown TPs, i.e. those that are not predicted or found in literature, and are discussed further below. Finally, the `logic` algorithm is primarily meant if no or very little information on possible parents and/or TPs is available.

The parent information that is needed for most algorithms is taken from one of the following:

1. The data in a suspect list (follows the same format as suspect screening)
2. The data from suspects that were matched to feature groups (e.g. obtained with `screenSuspects`, see suspect screening)
3. A `compounds` object obtained with compound annotation (only `biotransformer`, `cts` and `library`)

The second option is often used in most workflows. The use of compound annotations as parent input could be considered if the parents are not known. However, care must be taken since *all* the candidates are used, and it is highly recommend to filter the object in advance with e.g. the `topMost` filter. For `library` and `library_formula`, the parent input is optional: if no parents are specified (`parents=NULL`) then TP data for *all* parents in the database is used.

An overview of common arguments for TP generation is listed below.

Argument	Algorithm(s)	Remarks
<code>parents</code>	<code>biotransformer</code> , <code>cts</code> , <code>library</code> , <code>library_formula</code> , <code>ann_form</code> , <code>ann_comp</code>	The input parents.
<code>fGroups</code>	<code>logic</code>	The input feature groups to calculate TPs for.
<code>type</code>	<code>biotransformer</code>	The prediction type: "env", "ecbased", "cyp450", "phaseII", "hgut", "superbio", "allHuman". See BioTransformer for more details.
<code>transLibrary</code>	<code>cts</code>	The transformation library that should be used: "hydrolysis", "abiotic_reduction", "photolysis_unranked", "photolysis_ranked", "mammalian_metabolism", "combined_abioticreduction_hydrolysis", "combined_photolysis_abiotic_hydrolysis", "pfas_environmental", "pfas_metabolism". See <code>cts</code> for more details.

Argument	Algorithm(s)	Remarks
TPLibrary	library,	Custom TP library.
transformations	library_formula	
generations	logic	Custom TP transformation rules.
	biotransformer,	The number of transformation generations to consider.
	cts, library,	
	library_formula	
adduct	logic	The assumed adduct of the parents (e.g. "[M+H]+"). Not needed when adduct annotations are available.
formulas/compounds	ann_form/ann_comp	The input formula/compound annotations to extract TPs from.
TPsRef, fGroupsComps	ann_comp	The reference TPs and feature groups to use for candidate ranking. See below for more details.
TPStructParams	biotransformer,	Other advanced parameters, e.g. to calculate structural
	cts, library,	similarities. See <code>?getDefTPStructParams</code> for more
	ann_comp	details.

Some examples on how to generate TPs are shown below:

```
# predict environmental TPs with BioTransformer for all parents in a suspect list
TPsBT <- generateTPs("biotransformer", parents = patRoonaData::suspectsPos,
                     type = "env")
# obtain all TPs from the default library
TPsLib <- generateTPs("library")
# get TPs for the parents matched in a suspect screening
TPsLib <- generateTPs("library", parents = fGroupsScr)

# calculate TPs for all feature groups
TPsLogic <- generateTPs("logic", fGroups, adduct = "[M+H]+")

# use formula annotations to obtain TPs
TPsAnnForm <- generateTPs("ann_form", formulas = formulas)

# use compound annotations to obtain TPs and provide additional suspect and feature data
↳ for candidate ranking
TPsAnnComp <- generateTPs("ann_comp", compounds = compounds, TPsRef = TPsBT, fGroupsComps
↳ = fGroups)
```

7.1.1 (Custom) Libraries and transformations

By default the library and logic algorithms use data that is installed with `patRoona` (based on PubChem transformations and Schollee et al. (2015), respectively). However, it is also possible to use custom data. For the `library_formula` no default library is provided, however, these can easily be generated as is discussed at the end of the section.

To use a custom TP structure library for the `library` algorithm a simple `data.frame` is needed with the names, SMILES and optionally log P values for the parents and TPs. The log P values are used for prediction of the retention time direction of a TP compared to its parent, as is discussed later. The following small library has two TPs for benzotriazole and one for DEET:

```
myTPLib <- data.frame(parent_name = c("1H-Benzotriazole", "1H-Benzotriazole", "DEET"),
                     parent_SMILES = c("C1=CC2=NNN=C2C=C1", "C1=CC2=NNN=C2C=C1",
↳ "CCN(CC)C(=O)C1=CC=CC(=C1)C"),
```

```

TP_name = c("1-Methylbenzotriazole", "1-Hydroxybenzotriazole",
  ↪ "N-ethyl-m-toluamide"),
TP_SMILES = c("CN1C2=CC=CC=C2N=N1", "C1=CC=C2C(=C1)N=NN2O",
  ↪ "CCNC(=O)C1=CC=CC(=C1)C")
myTPLib

```

```

#>      parent_name      parent_SMILES      TP_name      TP_SMILES
#> 1 1H-Benzotriazole      C1=CC2=NNN=C2C=C1 1-Methylbenzotriazole      CN1C2=CC=CC=C2N=N1
#> 2 1H-Benzotriazole      C1=CC2=NNN=C2C=C1 1-Hydroxybenzotriazole      C1=CC=C2C(=C1)N=NN2O
#> 3      DEET      CCN(CC)C(=O)C1=CC=CC(=C1)C      N-ethyl-m-toluamide      CCNC(=O)C1=CC=CC(=C1)C

```

To use this library, simply pass it to the `TPLibrary` argument:

```
TPs <- generateTPs("library", TPLibrary = myTPLib)
```

For `library_formula` the library follows the same format. However, here the formula should be specified instead of the SMILES with the `parent_formula` and `TP_formula` columns (although it is still allowed to only specify SMILES, in which case the formulae are calculated from the SMILES).

For the `logic` algorithm a table with custom transformation rules can be specified for TP calculations:

```

myTrans <- data.frame(transformation = c("hydroxylation", "demethylation"),
  add = c("O", ""),
  sub = c("", "CH2"),
  retDir = c(-1, -1))
myTrans

```

```

#>      transformation add sub retDir
#> 1 hydroxylation      0      -1
#> 2 demethylation      CH2     -1

```

The `add` and `sub` columns are used to denote the elements that are added or subtracted by the reaction. These are used to calculate mass differences between parents and TPs. The `retDir` column is used to indicate the retention time direction of the parent compared to the TP: -1 (elutes before parent), 1 (elutes after parent) or 0 (similar or unknown). This is discussed later how this data can be used to filter TP candidates. The custom rules can be used by passing them to the `transformations` argument:

```
TPs <- generateTPs("logic", fGroups, adduct = "[M+H]+", transformations = myTrans)
```

The `genFormulaTPLibrary()` utility function can be used to automatically generate TP libraries suitable for the `library_formula` algorithm. The transformation rules to calculate TPs are specified in the same format as used by the `logic` algorithm.

```

myTPFormLib <- genFormulaTPLibrary(parents = patRoonData::suspectsPos, transformations =
  ↪ myTrans)
# also calculate second generation TPs (TPs of TPs)
myTPFormLib2 <- genFormulaTPLibrary(parents = patRoonData::suspectsPos, transformations =
  ↪ myTrans,
                                generations = 2)

# Use library
TPs <- generateTPs("library_formula", TPLibrary = myTPFormLib)

```

Compared to the `logic` algorithm, the `library_formula` algorithm is more (and only) suitable for suspect/target screening workflows, allows multiple transformation generations and allows better customization through manually adding/removing TPs from the library prior to passing it to `generateTPs()`.

7.1.2 TPs from feature annotation candidates

The `ann_form` and `ann_comp` algorithms are intended to provide a thorough screening and elucidation of TPs that are otherwise difficult to find with other algorithms. These algorithms assume that TPs of interest can be revealed from a thorough feature annotation workflow based on formulae (`ann_form`) or compounds (`ann_comp`). The annotation candidates are prioritized and ranked by the *TP Score*, which is calculated from properties such as the fit of the candidate structure or formula into the parent (or vice versa) and similarity to suspects (i.e. TPs obtained by other algorithms). For more details see Helmus et al. (2025), `?generateTPsAnnForm` and `?generateTPsAnnComp`.

To provide a thorough screening with `ann_comp`, often a large compound database such as PubChem is used for compound annotation. This typically results in tens of thousands of candidates for each parent. Hence, obtaining the compound annotations and generating TPs with `ann_comp` is computationally intensive and can take multiple hours. Afterwards, the `topMost` filter should be used to reduce the number of candidates to a manageable size (see the next section).

7.1.3 Processing data

Similar to other workflow data, several generic functions are available to inspect the TP data:

Generic	Remarks
<code>length()</code>	Returns the total number of transformation products
<code>names()</code>	Returns the names of the parents
<code>parents()</code>	Returns a table with information about the parents
<code>products()</code>	Returns a <code>list</code> with for each parent a table with TPs
<code>as.data.table()</code> , <code>as.data.frame</code>	Convert all the object information into a <code>data.table/data.frame</code>
<code>"[" / "\$</code> operators	Extract TP information for a specified parent

Some examples:

```
# just show a few columns in this example, there are many more!
# note: the double dot syntax (..cols) is necessary since the data is stored as
↳ data.tables
cols <- c("name", "formula", "InChIKey")
parents(TPs)[1:5, ..cols]
```

```
#>           name      formula      InChIKey
#>      <char>    <char>      <char>
#> 1:      DEET    C12H17NO  MMOXZBCLCQITDF-UHFFFAOYSA-N
#> 2:    Irgarol  C11H19N5S  HDHLIWCXDDZUFH-UHFFFAOYSA-N
#> 3:  Prometryne C10H19N5S  AAEVYOYXGOFMJO-UHFFFAOYSA-N
#> 4: Trimethoprim C14H18N4O3  IEDVJHCEMCRBQM-UHFFFAOYSA-N
#> 5: 1H-benzotriazole C6H5N3  QRUDEWIWKLJBPS-UHFFFAOYSA-N
```

```
TPs[["DEET"]][, ..cols]
```

```
#>      name      formula      InChIKey
#>      <char>      <char>      <char>
#> 1: DEET-TP1 C12H17NO2 FRZJZRVZZNTMAW-UHFFFAOYSA-N
#> 2: DEET-TP2 C12H17NO2 KVTUZBGZTRABBQ-UHFFFAOYSA-N
#> 3: DEET-TP3      C2H4O IKHGUXGNUITLKF-UHFFFAOYSA-N
#> 4: DEET-TP4 C10H13NO  FPNATACRXASTP-UHFFFAOYSA-N
#> 5: DEET-TP4 C10H13NO  FPNATACRXASTP-UHFFFAOYSA-N
#> 6: DEET-TP5 C4H11N   HPNMFZURTQLUMO-UHFFFAOYSA-N
#> 7: DEET-TP6 C8H7O2   GPSDUZXPYCFOSQ-UHFFFAOYSA-M
#> 8: DEET-TP7 C8H9NO   WGRPQCFFBRDZFV-UHFFFAOYSA-N
#> 9: DEET-TP1 C12H17NO2 FRZJZRVZZNTMAW-UHFFFAOYSA-N
```

```
TPs[[2]][, ..cols]
```

```
#>      name      formula      InChIKey
#>      <char>      <char>      <char>
#> 1: Irgarol-TP1 C8H15N5S MWBBDLRPMWTLRX-UHFFFAOYSA-N
#> 2: Irgarol-TP2 C11H19N5OS HFCMSBLJLJOGGL-UHFFFAOYSA-N
```

```
as.data.table(TPs)[1:5, 1:3]
```

```
#>      parent      transformation
#>      <char>      <char>
#> 1: DEET Aliphatic hydroxylation of methyl carbon adjacent to aromatic ring / Human Phase I
#> 2: DEET      Hydroxylation of terminal methyl / Human Phase I N-Ethyl
#> 3: DEET      N-dealkylation of tertiary carboxamide / Human Phase I
#> 4: DEET      N-dealkylation of tertiary carboxamide / Human Phase I
#> 5: DEET      Metabolism
```

In addition, the following generic functions are available to modify or convert the object data:

Generic	Remarks
"[" operator	Subset this object on given parents
filter	Filters this object (available functionality depends on TP generation algorithm)
convertToSuspects	Generates a suspect list of all TPs (and optionally parents) that is suitable for screenSuspects
convertToMFDB	Generates a MetFrag database for all TPs (and optionally parents, only for TPs with structural information)
plotGraph	Generates an interactive plot to explore transformation hierarchies (only for TPs with structural or formula information)
plotVenn, plotUpSet	Compare results between different algorithms with Venn/UpSet diagrams (only for TPs with structural information)
consensus	Combine results from different algorithms (only for TPs with structural information). See the algorithm consensus section.

The `convertToMFDB` function is especially handy with predicted TPs, as it allows generating a compound database for TPs that may not be available in commonly used databases. This is further demonstrated in the first example.

```

TPs2 <- TPs[1:10] # only keep results for first ten parents

# only keep TPs with likely/probably likelihood (specific property for CTS algorithm)
TPsF <- filter(TPs, properties = list(likelihood = c("LIKELY", "PROBABLE")))

# remove transformation products that are isomers to their parent or sibling TPs
# may simplify data as these are often difficult to identify
TPsF <- filter(TPs, removeParentIsomers = TRUE, removeTPIsomers = TRUE)

# remove duplicate transformation products from each parent
# these can occur if different pathways yield the same TPs
TPsF <- filter(TPs, removeDuplicates = TRUE)

# only keep TPs that have a structural similarity to their parent of >= 0.5
# (set TPStructParams=getDefTPStructParams(calcSims=TRUE) when executing generateTPs())
TPsF <- filter(TPs, minSimilarity = 0.5)

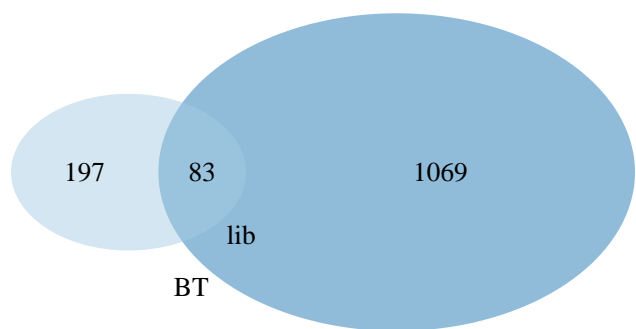
# only keep TPs with a TP Score >= 0.5 and the highest 25 TPs per parent
# see ?transformationProductsAnnComp for specific filters
TPsAnnCompF <- filter(TPsAnnComp, minTPScore = 0.5, topMost = 25)

# do a suspect screening for all TPs and their parents
# this is often part of a workflow and is discussed further in the next section.
suspects <- convertToSuspects(TPs, includeParents = TRUE)
fGroupsScr <- screenSuspects(fGroups, suspects, onlyHits = TRUE)

# use the TP data for a specialized MetFrag database
convertToMFDB(TPs, "TP-database.csv", includeParents = FALSE)
compoundsTPs <- generateCompounds(fGroups, mslists, "metfrag", database = "csv",
                                  extraOpts = list(LocalDatabasePath =
↳ "TP-database.csv"))

plotVenn(TPsLib, TPsBT, labels = c("lib", "BT"))

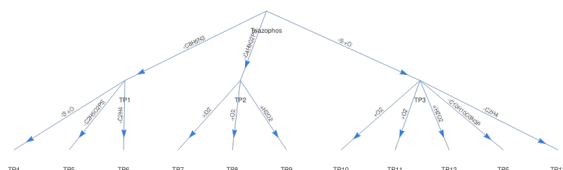
```



```

plotGraph(TPsBT, which = "Triazophos") # hierarchy for Triazophos parent

```



7.2 Linking parent and transformation product features

This section discusses one of the most important steps in a TP screening workflow, which is to link feature groups of parents with those of candidate transformation products. During this step, *TP components* are made, where each component consist of one or more feature groups of detected TPs for a particular parent. Note that componentization was already introduced before, but for very different algorithms. However, the data format for TP componentization is quite similar. After componentization, several filters are available to clean and prioritize the data. These can even allow workflows without obtaining potential TP candidates in advance, which is discussed in the last subsection.

7.2.1 Componentization

Like other algorithms, the `generateComponents` generic function is used to generate TP components, by setting the `algorithm` parameter to "tp".

The following arguments are of importance:

Argument	Remarks
<code>fGroups</code>	The input feature groups for the <i>parents</i>
<code>fGroupsTPs</code>	The input feature groups for the <i>TPs</i>
<code>ignoreParents</code>	Set to TRUE to ignore feature groups in <code>fGroupsTPs</code> that also occur in <code>fGroups</code>
<code>TPs</code>	The input transformation products, ie as generated by <code>generateTPs()</code>
<code>MSPeakLists,</code> <code>formulas,</code> <code>compounds</code>	Annotation objects used for similarity calculation between the parent and its TPs
<code>minRTDiff</code>	The minimum retention time difference (seconds) of a TP for it to be considered to elute differently than its parent.

7.2.1.1 Feature group input The `fGroups`, `fGroupsTPs` and `ignoreParents` arguments are used by the componentization algorithm to identify which feature groups can be considered as parents and which as TPs. Three scenarios are possible:

1. `fGroups=fGroupsTPs` and `ignoreParents=FALSE`: in this case no distinction is made, and all feature groups are considered a parent or TP (default if `fGroupsTPs` is not specified).
2. `fGroups` and `fGroupsTPs` contain different subsets of the *same featureGroups* object and `ignoreParents=FALSE`: only the feature groups in `fGroups/fGroupsTPs` are considered as parents/TPs.
3. As above, but with `ignoreParents=TRUE`: the same distinction is made as above, but any feature groups in `fGroupsTPs` are ignored for TP candidate selection if also present in `fGroups`.

The first scenario is often used if it is unknown which feature groups may be parents or which are TPs. Furthermore, this scenario may also be used if the dataset is sufficiently simple, for instance, because a suspect screening with the results from `convertToSuspects` (discussed in the previous section) would reliably discriminate between parents and TPs. A workflow with the first scenario is demonstrated in the second example.

In all other cases it is recommended to use either the second or third scenario, since making a prior distinction between parent and TP feature groups greatly simplifies the dataset and reduces false positives. A relative simple example where this can be used is when there are two sample groups: before and after treatment.

```
componTP <- generateComponents(algorithm = "tp",
                              fGroups = fGroups[ni = treatment == "before"],
                              fGroupsTPs = fGroups[ni = treatment == "after"])
```

In this example, only those feature groups with features present in the “before” treatment group are considered as parents, and those in “after” may be considered as a TP (see this section on how to group sample analyses with metadata). Since it is likely that there will be some overlap in feature groups between both sample groups, the `ignoreParents` flag can be used to not consider any of the overlap for TP assignments:

```
componTP <- generateComponents(algorithm = "tp",
                              fGroups = fGroups[ni = treatment == "before"],
                              fGroupsTPs = fGroups[ni = treatment == "after"],
                              ignoreParents = TRUE)
```

More sophisticated ways are of course possible to provide an upfront distinction between parent/TP feature groups. In the fourth example a workflow is demonstrated where fold changes are used.

NOTE The feature groups specified for `fGroups`/`fGroupsTPs` *must* always originate from the same `featureGroups` object.

If TPs were generated with an algorithm that requires parent input (see Obtaining TPs), then it is often mandatory that a suspect screening of parents and TPs is performed prior to componentization. This is necessary for the componentization algorithm to map the feature groups that belong to a particular parent or TP. Note that this step should *not* be performed for the `ann_form` and `ann_comp` algorithms, as these algorithms already provide the necessary mappings. The `convertToSuspects` function is used to prepare the suspect list:

```
# perform suspect screening
# NOTE: set includeParents to TRUE since both the parents and TPs should be screened
# NOTE: for the ann_form and ann_comp algorithms no suspect screening is necessary
suspects <- convertToSuspects(TPs, includeParents = TRUE)
fGroupsScr <- screenSuspects(fGroups, suspects, onlyHits = TRUE)

# do the componentization
# a similar distinction between fGroups/fGroupsScr as discussed above can of course also
  ↳ be done
componTP <- generateComponents(fGroups = fGroupsScr, ...)
```

If a parent screening was already performed in advance, for instance when the input parents to `generateTPs` are screening results, the screening results for parents and TPs can also be combined. The second example demonstrates this.

Note that in the case a parent suspect is matched to multiple feature groups, a component is made for each match. Similarly, if multiple feature groups match to the same TP suspect, all of them will be incorporated in the component.

When TPs were generated with the `logic` algorithm a suspect screening must also be carried out in advance. However, in this case it is not necessary to include the parents (since each parent equals a feature group no mapping is necessary). The `onlyHits` variable to `screenSuspects` must not be set in order to keep the parents.

```
# only screen for TPs
suspects <- convertToSuspects(TPs, includeParents = FALSE)
# but keep all other feature groups as these may be parents
fGroupsScr <- screenSuspects(fGroups, suspects, onlyHits = FALSE)

# do the componentization...
```

7.2.1.2 Annotation similarity calculation If additional annotation data for parents and TPs is given to the componentization algorithm, it will be used to calculate various similarity metrics. Often, the chemical structure for a transformation product is similar to that of its parent. Hence, there is a good chance that a parent and its TPs also share similar MS/MS data.

Firstly, if MS peak lists are provided, then the spectrum similarity is calculated between each parent and its potential TP candidates. This is performed with all the three different alignment shifts (see the spectrum similarity section for more details).

In case `formulas` and/or `compounds` objects are given as input to `generateComponents()`, then a parent/TP comparison is made by counting the number of fragments and neutral losses that they share (based on the formulae assigned to the MS/MS fragments). The counts are calculated in two different ways:

1. from the matches between the parent and the TP candidate
2. from the matches between the parent and *all* the annotation candidates in the input `formulas/compounds` object for the TP feature group

The latter is mainly used (and only available) in workflows where componentization is performed without previously generated TPs. To improve the usefulness of the total similarity metric, it is highly recommend to pre-treat the annotation objects with e.g. the `topMost` filter. Both calculation methods pool the data from the input `formulas` and `compounds` and only count unique fragment/neutral loss matches.

7.2.2 Processing data

The output of TP componentization is an object of the `componentsTPs` class. This *derives* from the ‘regular’ `components` class, therefore, all the data processing functionality described before (extraction, subsetting, filtering etc) are also valid for TP components.

For the `as.data.table()` method function (and `as.data.frame()`) the `candidates` argument can be used to specify if individual candidates for each feature group in the component should be included in the output:

```
# only output feature group data
as.data.table(componTP, candidates = FALSE)[name == "CMP2", .(name, parent_name, group)]
```

```
#>      name  parent_name      group
#>   <char>      <char>    <char>
#> 1:  CMP2 Dimethametryn M228_R353_323
#> 2:  CMP2 Dimethametryn M226_R342_312
#> 3:  CMP2 Dimethametryn M214_R341_264
```

```
# also include the candidates for each feature group
as.data.table(componTP, candidates = TRUE)[name == "CMP2", .(name, parent_name, group,
  ↪ candidate_name, formula)]
```

```
#>      name parent_name      group candidate_name formula
#>   <char>      <char>      <char>      <char>   <char>
#> 1:   CMP2 Dimethametryn M228_R353_323 Dimethametryn-TP2 C9H17N5S
#> 2:   CMP2 Dimethametryn M226_R342_312 Dimethametryn-TP5 C10H19N5O
#> 3:   CMP2 Dimethametryn M214_R341_264 Dimethametryn-TP9 C8H15N5S
```

Several additional filters are available to prioritize the data:

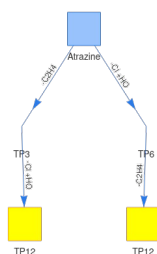
Filter	Remarks
<code>retDirMatch</code>	If <code>TRUE</code> only keep TPs with an expected chromatographic retention direction compared to the parent.
<code>minSpecSim</code> , <code>minSpecPrec</code> , <code>minSpecSimBoth</code>	The minimum spectrum similarity between the parent and TP. Calculated with no, "precursor" and "both" alignment shifting (see spectrum similarity).
<code>minFragMatches</code> , <code>minNLMatches</code>	Minimum number of formula fragment/neutral loss matches between parent and TP (discussed in previous section).
<code>minTotFragMatches</code> , <code>minTotNLMatches</code>	As above, but for total matches.
<code>formulas</code>	A <code>formulas</code> object used to further verify candidate TPs that were generated by the <code>logic</code> algorithm.

The `retDirMatch` filter compares the expected and observed *retention time direction* of a TP in order to decide if it should be kept. The direction is a value of either `-1` (TP elutes before parent), `+1` (TP elutes after parent) or `0` (TP elutes very close to the parent or its direction is unknown). The directions are taken from the generated transformation products. In most cases the log P values are compared of a TP and its parent. Here, it is assumed that lower log P values result in earlier elution (i.e. typical with reversed phase LC). For the `logic` algorithm the retention time direction is taken from the transformation rules table. Note that specifying a large enough value for the `minRTDiff` argument to `generateComponents` is important to ensure that some tolerance exists while comparing retention time directions of parent and TPs. Furthermore, the `TPStructParams` argument to `generateTPs()` can be used to tweak the calculation of expected retention time directions from log P values (see `?getDefTPStructParams`). This filter does nothing if either the observed or expected direction is zero.

When TP data was generated with the `logic` algorithm it is recommended to use the `formulas` filter. This filter uses formula annotations to verify that (1) a parent feature group contains the elements that are subtracted during the transformation and (2) the TP feature group contains the elements that were added during the transformation. Since the 'right' candidate formula is most likely not yet known, this filter looks at *all* candidates. Therefore, it is recommended to filter the `formulas` object, for instance, with the `topMost` filter.

Finally, the `plotGraph()` method function that was introduced exploring transformation hierarchies for structure TPs, can also incorporate componentization results to simplify the plot and mark TP hits:

```
plotGraph(TPsBT, which = "Atrazine", components = componTP)
```



7.2.3 Omitting transformation product input

The `TPs` argument to `generateComponents` can also be omitted (i.e. `TPs=NULL`). In this case every feature group from `fGroupTPs` is considered to be a potential TP for the potential parents specified for `fGroups`. An advantage is that the screening workflow is not limited to any known TPs or transformations. However, such a workflow has high demands on prioritization steps before and after the componentization to rule out the many false positives that may occur.

When no transformation data is supplied it is crucial to make a prior distinction between parent and TP feature groups. Afterwards, the MS/MS spectral and other annotation similarity filters mentioned in the previous section may help to further prioritize data.

The fourth example demonstrates such a workflow.

7.2.4 Reporting TP components

The TP components can be reported with the `report` function. This is done by setting the `components` function argument (i.e. equally to all other component types). The results will be displayed with a customized format that allows easy exploring of each parent with its TPs. In addition, the `TPs` argument can be set to include additional data such as transformation pathways.

```
report(fGroups, components = componTP, TPs = TPs)
```

7.3 Example workflows

The next subsections demonstrate several approaches to perform a TP screening workflow with `patRoön`. In all examples it is assumed that feature groups were already obtained (with the `findFeatures` and `groupFeatures` functions) and stored in the `fGroups` variable.

The workflows with `patRoön` are designed to be flexible, and the examples here are primarily meant to implement your own workflow. Furthermore, some of the techniques used in the examples can also be combined. For instance, the Fold change classification and MS/MS similarity filters applied in the fourth example could also be applied to any of the other examples.

7.3.1 Screen predicted TPs for targets

The first example is a simple workflow where TPs are predicted for a set of given parents with `BioTransformer` and subsequently screened. A `MetFrag` compound database is generated and used for annotation.

```

# predict TPs for a fixed list of parents
TPs <- generateTPs("biotransformer", parents = patRoonData::suspectsPos)

# screen for the TPs
suspectsTPs <- convertToSuspects(TPs, includeParents = FALSE)
fGroupsTPs <- screenSuspects(fGroups, suspectsTPs, adduct = "[M+H]+", onlyHits = TRUE)

# perform annotation of TPs
mslistsTPs <- generateMSPeakLists(fGroupsTPs)
convertToMFDB(TPs, "TP-database.csv", includeParents = FALSE) # generate MetFrag database
compoundsTPs <- generateCompounds(fGroupsTPs, mslistsTPs, "metfrag", adduct = "[M+H]+",
  ↪ database = "csv",
                                extraOpts = list(LocalDatabasePath =
  ↪ "TP-database.csv"))

```

7.3.2 Screening TPs from a library for suspects

In this example TPs of interest are obtained for the parents that surfaced from of a suspect screening. The steps of this workflow are:

1. Suspect screening parents.
2. Obtain TPs for the suspect hits from a library.
3. A second suspect screening is performed for TPs and the original parent screening results are amended.
Note that the parent data is needed for componentization.
4. Both parents and TPs are annotated using a database generated from their chemical structures.
5. Some prioritization is performed by
 - a. Only keeping candidate structures for which *in-silico* fragmentation resulted in at least one annotated MS/MS peak.
 - b. Only keeping suspect hits with an estimated identification level of 3 or better.
6. The TP components are made and only feature groups with parent/TP assignments are kept.
7. All results are reported.

```

# step 1
fGroupsScr <- screenSuspects(fGroups, patRoonData::suspectsPos, adduct = "[M+H]+")
# step 2
TPs <- generateTPs("library", parents = fGroupsScr)

# step 3
suspects <- convertToSuspects(TPs)
fGroupsScr <- screenSuspects(fGroupsScr, suspects, adduct = "[M+H]+", onlyHits = TRUE,
  ↪ amend = TRUE)

# step 4
mslistsScr <- generateMSPeakLists(fGroupsScr)
convertToMFDB(TPs, "TP-database.csv", includeParents = TRUE)
compoundsScr <- generateCompounds(fGroupsScr, mslistsScr, "metfrag", adduct = "[M+H]+",
  ↪ database = "csv",
                                extraOpts = list(LocalDatabasePath =
  ↪ "TP-database.csv"))

# step 5a

```

```

compoundsScr <- filter(compoundsScr, minExplainedPeaks = 1)

# step 5b
fGroupsScrAnn <- estimateIDConfidence(fGroupsScr, MSPeakLists = mslistsScr,
                                     compounds = compoundsScr)
fGroupsScrAnn <- filter(fGroupsScrAnn, maxLevel = 3, onlyHits = TRUE)

# step 6
componTP <- generateComponents(fGroupsScrAnn, "tp", TPs = TPs, MSPeakLists = mslistsScr,
                              compounds = compoundsScr)
fGroupsScrAnn <- fGroupsScrAnn[results = componTP]

# step 7
report(fGroupsScrAnn, MSPeakLists = mslistsScr, compounds = compoundsScr,
       components = componTP, TPs = TPs)

```

7.3.3 Non-target screening of predicted TPs

This example uses metabolic logic to calculate possible TPs for all feature groups from a complete non-target screening. This example demonstrates how a workflow can be performed when little is known about the identity of the parents. The steps of this workflow are:

1. Formula annotations are performed for all feature groups.
2. These results are then limited to the top 5 candidates, and only feature groups with annotations are kept.
3. The TPs are calculated for all remaining feature groups.
4. A suspect screening is performed to find the TPs. Unlike the previous example feature groups without hits are kept (discussed here).
5. The components are generated
6. The components are filtered:
 - a. The TPs must follow an expected retention time direction
 - b. The parent/TPs should have at least one candidate formula that fits with the transformation.
7. Only feature groups are kept with parent/TP assignments and all results are reported.

```

# steps 1-2
mslists <- generateMSPeakLists(fGroups)
formulas <- generateFormulas(fGroups, mslists, "genform", adduct = "[M+H]+")
formulas <- filter(formulas, topMost = 5)
fGroups <- fGroups[results = formulas]

# step 3
TPs <- generateTPs("logic", fGroups = fGroups, adduct = "[M+H]+")

# step 4
suspects <- convertToSuspects(TPs)
fGroupsScr <- screenSuspects(fGroups, suspects, adduct = "[M+H]+", onlyHits = FALSE)

# step 5
componTP <- generateComponents(fGroupsScr, "tp", TPs = TPs, MSPeakLists = mslists,
                              formulas = formulas)

```

```

# step 6
componTP <- filter(componTP, retDirMatch = TRUE, formulas = formulas)

# step 7
fGroupsScr <- fGroupsScr[results = componTP]
report(fGroupsScr, MSPeakLists = mslists, formulas = formulas, components = componTP)

```

7.3.4 Non-target screening of TPs by annotation similarities

This example shows a workflow where no TP data from a prediction or library is used. Instead, this workflow relies on statistics and MS/MS data to find feature groups which may potentially have a parent - TP relationship. The workflow is similar to that of the previous example. The steps of this workflow are:

1. Fold changes (FC) between two sample groups are calculated to classify which feature groups are decreasing (i.e. parents) or increasing (i.e. TPs).
2. Feature groups without classification are removed.
3. Formula annotations are performed like the previous example.
4. The componentization is performed and the FC classifications are used to specify which feature groups are to be considered parents or TPs.
5. Only TPs are kept that show a high MS/MS spectral similarity and share at least one fragment with their parent.
6. Only feature groups are kept with parent/TP assignments and all results are reported.

```

# step 1
tab <- as.data.table(fGroups, FCPParams = getFCParams(c("before", "after")))
groupsParents <- tab[classification == "decrease"]$group
groupsTPs <- tab[classification == "increase"]$group

# step 2
fGroups <- fGroups[, union(groupsParents, groupsTPs)]

# step 3
mslists <- generateMSPeakLists(fGroups)
formulas <- generateFormulas(fGroups, mslists, "genform", adduct = "[M+H]+")
formulas <- filter(formulas, topMost = 5)
fGroups <- fGroups[results = formulas]

# step 4
componTP <- generateComponents(algorithm = "tp",
                              fGroups = fGroups[, groupsParents],
                              fGroupsTPs = fGroups[, groupsTPs],
                              MSPeakLists = mslists, formulas = formulas)

# step 5
componTP <- filter(componTP, minSpecSimBoth = 0.75, minTotFragMatches = 1)

# step 6
fGroups <- fGroups[results = componTP]
report(fGroups, MSPeakLists = mslists, formulas = formulas, components = componTP)

```

8 Ion mobility spectrometry (IMS-HRMS) workflows

8.1 Introduction

This chapter describes workflows to process data from Ion Mobility Spectrometry coupled to high resolution mass spectrometry (IMS-HRMS) instruments in **patRoön**. These are simply referred to as *IMS workflows*.

IMS is increasingly used to improve non-target analysis, and has the potential to improve separation of isomeric and isobaric compounds, clean up HRMS data and use the collision cross section (CCS) to improve identification of compounds. **patRoön 3.0** adds IMS support throughout the complete workflow and data processing functionality to take advantage of the additional information provided by IMS.

In **patRoön** three types of IMS workflows can be distinguished:

1. **LC-MS workflows with IMS data:** These workflows are like regular (non-IMS) workflows, but work with IMS data.
2. **Direct mobility assignment:** Feature mobilities are assigned directly during feature detection.
3. **Post mobility assignment:** Feature mobilities are assigned after finding and grouping features.

The following sections further detail each of these workflows, including the required data and the steps involved.

NOTE IMS workflows are considerably more computationally demanding, as the raw IMS data typically contain several orders of magnitude more mass spectra. This is especially apparent during feature detection, but also subsequent steps such as creation of chromatograms and spectra will take up more time and RAM. The direct mobility assignment workflows are generally most demanding.

NOTE While IMS workflows are largely the same as non-IMS workflows, there are some important considerations (outlined in this chapter). It is highly recommended to experiment with the newProject tool and study the example workflow to get a feeling of how IMS workflows work.

NOTE IMS workflows have been primarily developed with Bruker TIMS and Agilent IMS data. Most of the functionality was designed to be generic and applicable to other IMS data as well, but data from other instruments likely need to be optimized. Furthermore, mobility and CCS conversion is not yet supported for other instruments. Contributions to test and extend support for other instruments are *very welcome!*

8.1.1 LC-MS workflows with IMS data

In this workflow the raw IMS data is ‘collapsed’ and is converted in such a way it looks like ‘regular’ LC-MS data. This is achieved by summing up the mass spectra in each IMS frame and subsequently replacing the IMS frames by the combined spectra. The workflow then proceeds as a regular LC-MS workflow. Hence, this type of workflow does not bring any of the benefits of IMS data, but allows full compatibility with software tools that do not support IMS data. Furthermore, this type of workflow is much less computationally intensive, which makes it suitable for e.g. initial exploration of data.

8.1.2 Direct mobility assignment

This workflow relies on a feature detection algorithm that separates features by ion mobility and assigns the feature mobilities directly during feature detection. In **patRoön** this is currently only supported by the **piek** and **greedy** algorithms or by importing feature data from other algorithms. Most of the remaining

workflow is the same as non-IMS workflows. However, it typically adds steps to calculate CCS values for features and matches these with (predicted) CCS values of suspects and/or compound annotation candidates. Furthermore, the IMS data is internally used to cleanup extracted ion chromatograms and mass spectra, which can improve their visualization and quality of feature annotation.

8.1.3 Post mobility assignment

Post mobility assignment workflows can be considered as a hybrid approach to the two previous workflow types: the workflow starts with feature detection and grouping as in a regular LC-MS workflow, followed by assigning IMS data to features and then proceeding much like the direct mobility assignment workflow.

The mobility assignment consists of the following steps:

1. Feature detection and grouping is performed with classical algorithms like regular LC-MS workflows.
2. Extracted ion mobilograms are generated for each feature and automatic peak detection is used to detect mobilities.
3. Each of the detected mobilities for a feature are used to generate a new set of *IMS features*. These features inherit their properties from the original features, which are referred to as *IMS precursors*. A link is formed between the IMS features and the IMS precursor.
4. The IMS feature data such as retention time and intensities is updated from mobility filtered extracted ion chromatograms. Any features that could not be detected from the filtered data are removed.
5. The feature groups are updated with IMS data and similar links between *IMS feature groups* and *IMS precursors* are formed.

All these steps are automatically performed by the `assignMobilities()` method function that will be discussed later.

An advantage of post mobility workflows is the compatibility with any of the feature detection and grouping algorithms supported in **patRoön**. In addition, post assignment is often less computationally intensive than direct mobility assignment workflows. Furthermore, the links between IMS features and IMS precursors facilitate the recognition of possible protomers (i.e. redundant features of a same compound with the same retention time and m/z but different mobility) and assists in recognizing IMS features across MS polarities in sets workflows. Finally, post mobility assignment workflows can selectively fallback to ‘regular’ LC-MS feature data from the IMS precursors in steps where this makes more sense, this will be discussed later. Potential disadvantages of post mobility assignment workflows are the reliance on raw data that is with and without IMS dimension, and the mobility assignment typically relies on two steps of peak detection (steps 2 and 4) which may fail with e.g. low intensity data. Nevertheless, this type of workflow is usually recommended for most IMS data processing projects.

8.1.4 Summary

The following table summarizes the differences between the three IMS workflows (some of these are introduced later in this chapter):

	LC-MS with IMS data	Direct mobility assignment	Post mobility assignment
Raw data requirements ¹	IMS collapsed	Raw IMS data	IMS and IMS collapsed raw data
Feature detection and grouping	Regular non-IMS (all algorithms)	Utilizes IMS separation (few algorithms)	Regular non-IMS (all algorithms)
Feature mobility assignment	No	During feature detection	After feature grouping
Feature CCS assignment	No	Yes	Yes

	LC-MS with IMS data	Direct mobility assignment	Post mobility assignment
IMS precursor links	No	No	Yes
Suspect & compound CCS matching	No	Yes	Yes
HRMS cleanup	No	Yes	Yes
non-IMS feature fallback	-	No	Yes
Supports sets workflows ²	Yes	Limited	Moderate

Notes:

1. If the `piek` feature detection algorithm is used then (only) IMS raw data may suffice, see more info [here](#).
2. Discussed further [here](#)

8.2 Performing IMS workflows

8.2.1 Parameter defaults

To effectively process IMS-HRMS data, it may be required to adjust the default numerical limits (e.g. mobility tolerances) and other settings. This is especially important since (1) different IMS instruments typically use different mobility units and (2) the data is often more ‘raw’, i.e. more noisy and without centroiding, which therefore require more instrument sensitive optimization. To simplify this, `patRoön` brings defaults for Bruker timsTOF and Agilent IMS instruments. The default instrument type and general numeric limits, are specified in the limits file. The instrument type that is defined there, also affects the default settings used for `piek`, mobilogram extraction and peak list generation (discussed later). To generate a limits file with the default settings for Agilent IMS data, the following code can be used:

```
genLimitsFile(IMS = "agilent")
```

A limits file is normally generated when using the `newProject()` function to create a new project. If no limits file is present, the defaults are used from Bruker instruments. See the limits advanced section for more details on the limits file and how to adjust it.

8.2.2 Raw data

The data conversion and pre-treatment section already discussed how raw data can be converted to make it suitable for processing. For the workflows that require raw data that has IMS data (see the Table of the previous section) there are two options:

1. Use the raw instrument directly. This is currently only supported for Bruker TIMS data, see details [here](#).
2. Convert the data to the `ims` type in the `mzML` file format, e.g. by using `convertMSFiles()` (explained [here](#)).

If IMS collapsed data is needed (see the Table of the previous section) then raw data of the type `raw` or `ims` should be converted to the type `centroid`.

The data conversion and pre-treatment section includes some examples on how to do these conversion steps.

8.2.3 Direct mobility assignment (DMA) workflows

8.2.3.1 Feature detection In direct mobility assignment (DMA) workflows the feature detection needs to be aware of the IMS dimension. In `patRoon` this is currently only supported by the `piek` algorithm. Alternatively, feature data from another IMS aware feature detection can also be imported.

The `piek` algorithm was already discussed before. By default, `piek` does not use IMS data for feature detection, even if processing IMS-HRMS data. To enable the use of IMS data, the `IMS` function argument be set to `TRUE`. For instance:

```
# comprehensive feature detection from 2 dimensional EIC bins
genEICParams <- getPiekEICParams(mzRange = c(100, 800), mobRange = c(0.5, 1.5), mzStep =
  ↳ 0.02, mobStep = 0.2)
peakParams <- getDefPeakParams("chrom", "piek")
fList <- findFeatures(anaInfo, "piek", IMS = TRUE, genEICParams = genEICParams,
  ↳ peakParams = peakParams)
```

8.2.3.1.1 Optimizing computational demands When `piek` is used to detect IMS resolved features, it will generate features from EIC bins that are created from the m/z **and** mobility dimension. This can result in a very large number of EIC bins, which can make feature detection computationally intensive. For this reason, it is recommended to have sufficient amount of RAM (e.g. ≥ 32 GB).

In addition, there are several advanced EIC parameters that can be tweaked to reduce computational demands (see `?getPiekEICParams` for a complete overview). For instance:

EIC parameter	Description
<code>mzRange</code> , <code>mobRange</code>	Limits the dimensions of the EIC bins. Reducing this can significantly speed up feature detection.
<code>retRange</code>	Limits the retention time range of the EIC bins. Useful to e.g. discard features from the dead volume.
<code>filter</code> , <code>filterIMS</code>	Applies filters which can considerably reduce the number of EIC bins. The <code>filter</code> parameter was discussed before. The <code>filterIMS</code> parameter is analogous to <code>filter</code> , but applies the given filter in the mobility dimension. <code>filterIMS</code> should be either <code>"none"</code> for no mobility filtering or equalling <code>"filter"</code> (i.e. <code>"suspects"</code> or <code>"ms2"</code>).
<code>minEICIntensity</code>	Discard EICs with a maximum intensity below the given intensity threshold.
<code>minEICAdjTime</code> , <code>minEICAdjPoints</code> , <code>minEICAdjIntensity</code>	Discard EICs without a clear continuous signal.
<code>topMostEICMob</code>	Only keep these top most intense EICs (based on maximum signal intensity).

Furthermore, the `EICBatchSize` function argument can be set to split the work in multiple batches, which is especially useful to reduce RAM usage. The following examples show how these parameters can be used to reduce computational needs:

```
# filter EIC bins by m/z and mobility data from detected PASEF MS2 precursors
genEICParams <- getPiekEICParams(filter = "ms2", filterIMS = "ms2", minTIC = 1000)
fList <- findFeatures(anaInfo, "piek", IMS = TRUE, genEICParams = genEICParams)

# filter by suspect m/z data, but don't filter mobility bins (e.g. if suspect list
  ↳ doesn't contain IMS data)
genEICParams <- getPiekEICParams(filter = "suspects", filterIMS = "none", mobRange =
  ↳ c(0.5, 1.5))
```

```
fList <- findFeatures(anaInfo, "piek", IMS = TRUE, genEICParams = genEICParams,
                     suspects = suspList, adduct = "[M+H]+")

# focus on low m/z and mobility features with high intensity
# don't load and process more than 10,000 EICs at a time
genEICParams <- getPiekEICParams(mzRange = c(100, 300), mobRange = c(0.5, 0.8),
  ↪ minEICIntensity = 1E5)
fList <- findFeatures(anaInfo, "piek", IMS = TRUE, genEICParams = genEICParams,
  ↪ EICBatchSize = 1E4)
```

8.2.3.1.2 m/z and mobility determination The raw m/z and mobility data is typically not or partially centroided. For this reason, **piek** automatically generates m/z and mobility *versus* intensity peak profiles for each feature, and determines the m/z and mobility data from these profiles. This process is largely similar to centroiding in HRMS workflows. For accurate m/z and mobility determination, it may be needed to tweak the default parameters that are used for profile generation:

EIC profile parameter	Description
sumWindowMZ, sumWindowMob	The retention window (+/- seconds) for summing up raw data to generate the profiles. Summing up multiple data points is especially needed for noisy data, e.g. generated with fast timsTOF methods. It's best to keep this smaller than the expected minimum LC peak width to avoid summing up data from different chromatographic peaks. Setting this to zero will disable summing.
smoothWindowMZ, smoothWindowMob, smoothExtMZ, smoothExtMob	Smoothing parameters. This is again primarily useful for noisy data. Setting these to zero will disable smoothing.

8.2.3.1.3 Instrument defaults To simplify the configuration of the previously discussed parameters, **piek** brings defaults for two different instrument types. The instrument is set by the **IMS** argument for **getPiekEICParams()** and should be set to "bruker" or "agilent", for Bruker timsTOF and Agilent IMS data, respectively. The default value for **IMS** is determined from the value determined from the previously discussed limits file.

8.2.3.2 Feature grouping The grouping of features in DMA workflows is currently only possible with the **greedy** algorithm or by importing data. The **greedy** algorithm was already briefly introduced before. This algorithm was introduced in **patRoom** 3.0 and is a simple and fast grouping algorithm, which also supports IMS data. Its **mobWindow** argument can be set to configure the mobility grouping tolerance. Some examples:

```
fGroups <- groupFeatures(fList, "greedy", mobWindow = 1)

# customize all tolerances and weights
fGroups <- groupFeatures(fList, "greedy",
  rtWindow = 5,
  mzWindow = 0.003,
  mobWindow = 0.5,
  scoreWeights = c(retention = 1, mz = 3, mobility = 3, intensity
  ↪ = 1))
```

8.2.4 Post mobility assignment (PMA) workflows

8.2.4.1 Feature detection and sample grouping In post mobility assignment (PMA) workflows, the feature detection and grouping is performed as in regular LC-MS workflows, i.e. without using the IMS data. Hence, the `findFeatures()` and `groupFeatures()` functions are used exactly like what was discussed before for HRMS workflows.

8.2.4.2 Assigning mobility values to features The `assignMobilities()` method function is used to assign mobilities to features. The most important function arguments are:

Argument	Description
<code>mobPeaksParams</code> , <code>chromPeaksParams</code>	Parameters for the peak detection in mobilograms and chromatograms, respectively. See <code>?getDefPeakParams()</code> for more details.
<code>EIMParams</code> , <code>EICParams</code>	Parameters for the extraction of ion mobilograms (EIMs) and chromatograms (EICs), respectively. See <code>?getDefEIMParams()</code> and <code>?getDefEICParams()</code> for more details.
<code>fallbackEIC</code>	Set to <code>TRUE</code> to use the raw EIC data to update properties of IMS features, in case peak detection fails in mobility filtered EICs (steps 3-4).
<code>fromSuspects</code>	<i>Suspect screening workflows:</i> Use suspect data for mobility assignment, this is discussed later.
<code>IMSMatchParams</code>	<i>Suspect screening workflows:</i> parameters to match IMS data between features and suspects, this is discussed later.

Note that mobility assignment is only performed if either `mobPeaksParams` or `fromSuspects` is set. The latter is discussed later. The `assignMobilities()` method function is executed after executing `groupFeatures()`, with often some intermediate steps in-between (e.g. filtering, componentization, discussed later in this chapter). A single call to `assignMobilities()` will perform all the mobility assignment steps that were introduced before.

The `EIMParams` parameter influences the construction of extracted ion mobilograms (EIMs), which are used to determine the feature mobilities. Importantly, for noisy data (e.g. obtained with fast timsTOF methods), the enabling if smoothing may be needed. The `IMS` argument to `getDefEIMParams()` can be set to `"bruker"` or `"agilent"` to use defaults for Bruker timsTOF and Agilent IMS data, respectively. The default value for `IMS` is determined from the value determined from the previously discussed limits file.

Some examples of how to use `assignMobilities()` for mobility assignment are given below:

```
fGroups <- assignMobilities(fGroups, mobPeaksParams = getDefPeakParams("bruker_ims",
  ↪ "piek"),
                           chromPeaksParams = getDefPeakParams("chrom", "piek"))

# customized peak detection parameters
fGroups <- assignMobilities(fGroups, mobPeaksParams = getDefPeakParams("agilent_ims",
  ↪ "xcms3", peakwidth = c(0.2, 3)),
                           chromPeaksParams = getDefPeakParams("chrom", "openms",
  ↪ gaussWidth = 20))

# use (aggressive) smoothing for noisy EIM data to improve mobility determination
fGroups <- assignMobilities(fGroups, mobPeaksParams = getDefPeakParams("bruker_ims",
  ↪ "piek"),
                           chromPeaksParams = getDefPeakParams("chrom", "piek"),
```

```

EIMParams = getDefEIMParams(smooth = "sg", smLength = 25))

# like above, setting changing instrument defaults for Agilent data
fGroups <- assignMobilities(fGroups, ..., EIMParams = getDefEIMParams(IMS = "agilent"))

```

Further details are found in the reference manual (`?assignMobilities_feat`).

8.2.5 Mobility and CCS conversion

Conversion of mobility and CCS values in DMA and PMA workflows is performed with the `assignMobilities()` function introduced in the previous section, and is activated if its `CCSPParams` function argument is set. This argument should be assigned with a `list` that contains the parameters to configure the conversions process. The parameters are instrument specific and the parameter list can be obtained with the `getCCSPParams()` function. The most important arguments for `getCCSPParams()` are:

Function argument	Description
<code>method</code>	Specifies the conversion method. Valid values are: "bruker", "mason-schamp_k" or "mason-schamp_1/k" or "agilent".
<code>calibrant</code>	Specifies the calibrant file or parameters if <code>method="agilent"</code> .

The function currently only supports Bruker TIMS and Agilent IMS data (contributions to Waters data and information are *very welcome!*). For Bruker TIMS instruments, the `method` should be set to "bruker" or "mason-schamp_1/k". The former relies on the Bruker TDF-SDK (see the Installation chapter to install it). The latter doesn't, but may give very minor differences in the results compared to how Bruker software performs its calculations. For Agilent IMS data, the `method` should be set to "agilent" and the `calibrant` argument should be set to configure the calibration settings. The value for `calibrant` is either a file path to the `.d` or `OverrideImsCal.xml` file that should be used for re-calibration, or a `list` with the elements `massGas`, `Tfix` and `beta`. It is recommended to read the reference manual for the `getCCSPParams()` function for more details (`?getCCSPParams`).

Some examples of how to use `assignMobilities()` for mobility and CCS conversion are given below:

```

# conversion of Bruker TIMS data using the TDF-SDK
fGroups <- assignMobilities(fGroups, CCSPParams = getCCSPParams("bruker"))

# conversion of Agilent IMS data
fGroups <- assignMobilities(fGroups, CCSPParams = getCCSPParams("agilent", calibrant =
  ↪ "path/to/Calibrant.d"))

# combine mobility assignment and CCS conversion into one step
fGroups <- assignMobilities(fGroups, mobPeaksParams = getDefPeakParams("bruker_ims",
  ↪ "piek"),
                           chromPeaksParams = getDefPeakParams("chrom", "piek"),
                           CCSPParams = getCCSPParams("bruker"))

```

8.2.6 Suspect screening

Suspect screening in IMS workflows is mostly the same as in non-IMS workflows. However, the suspect list can be extended with IMS properties to improve suspect matching, and performing the screening step may be slightly different for PMA workflows.

8.2.6.1 Suspect list The suspect list can contain the following additional columns (all optional):

- **mobility**: The mobility of the suspect.
- **CCS**: The collision cross section (CCS) of the suspect.
- **mobility_<adduct>**, **CCS_<adduct>**: The mobility and CCS of the suspect for a specific adduct. The <adduct> should be replaced with the adduct name, e.g. **mobility_[H+Na]+** or **CCS_[M-H]-**.

Multiple suspect reference values can be specified in each of the columns by separating them with a semicolon (;). If data for both the adduct and non-adduct specific columns is available (and not NA), then the latter gets precedence. It is recommended to include the adduct specific data whenever possible. During the suspect screening step to correct data is automatically chosen based on the adduct assigned to the feature (or assigned to the **adduct** function argument).

8.2.6.2 Prediction of mobility and CCS data The **assignMobilities()** function that was discussed before also has a method specifically for suspect lists. It can be used for the prediction (or library matching) of IMS data for suspects and convert CCS values to mobilities (or vice versa). The common function arguments include:

Function argument	Description
from	Specifies from where IMS data is added. See below for more details.
adducts	Specifies for which adducts IMS data is added or converted. A character vector with multiple values is allowed. Include NA to also consider the usage of non-adduct specific data. Any adducts specified in the suspect list (adduct column) are always considered.
CCSParams	Parameters for the mobility and CCS conversions. See its previous introduction.
overwrite	Set to TRUE to overwrite any existing IMS data in the suspect list.

The following options exist for the **from** function argument:

- **from=NULL** (default): No IMS data is added (but mobility and CCS conversion may still occur).
- **from="pubchemlite"**: IMS data is matched from the PubChemLite database with CCS data. See the Installation chapter for details on how to install the database.
- **from="c3sdb"**: Uses C3SDB to predict CCS values for suspects. See the Installation chapter for installation details.
- a **data.table** or **data.frame**: A custom library with IMS data.

Some examples:

```
susplist <- patRoonData::suspectsPos[1:4, ]

# adds CCS values from PubChemLite and converts them to mobilities
susplist <- assignMobilities(susplist, from = "pubchemlite", adducts = c("[M+H]+",
  ↪ "[M+Na]+", "[M+H]-"),
                           CCSParams = getCCSParams("mason-schamp_1/k"))
susplist[, grepl("name|InChIKey|mobility|CCS", colnames(susplist))]
```

```
#>           name           InChIKey      InChIKey1 CCS_[M+H]+ CCS_[M+Na]+ mobility_[M+H]+ mo
#> 1      DEET MMOXZBCLCQITDF-UHFFFAOYSA-N MMOXZBCLCQITDF      143.7      150.2      0.6707548
#> 2    Diglyme SBZXBUIDTXKZTM-UHFFFAOYSA-N SBZXBUIDTXKZTM      127.5      134.6      0.5797684
#> 3 Dimethametryn IKYICRRUVNIHPP-QMMMGPOBSA-N IKYICRRUVNIHPP      159.6      166.1      0.7571080
#> 4      Irgarol HDHLIWCXDDZUFH-UHFFFAOYSA-N HDHLIWCXDDZUFH      162.0      171.2      0.7681929
```

```

# overwrites data from custom library
IMSLib <- data.frame(
  name = c("DEET", "Diglyme"), # (name column is not mandatory)
  InChIKey1 = c("MMOXZBCLCQITDF", "SBZXBUIDTXKZTM"), # NOTE: matching happens by
  ↪ first-block IK by default
  "CCS_[M+H]+" = c(150, 160),
  check.names = FALSE # NOTE: need this to allow special characters in column names (ie
  ↪ in CCS column)
)
suspList <- assignMobilities(suspList, from = IMSLib, overwrite = TRUE,
  adducts = "[M+H]+", CCSParams =
  ↪ getCCSParams("mason-schamp_1/k"))
suspList[, grepl("name|InChIKey|mobility|CCS", colnames(suspList))]

```

```

#>           name           InChIKey      InChIKey1 CCS_[M+H]+ CCS_[M+Na]+ mobility_[M+H]+ mol
#> 1      DEET MMOXZBCLCQITDF-UHFFFAOYSA-N MMOXZBCLCQITDF      150.0      150.2      0.7001616
#> 2    Diglyme SBZXBUIDTXKZTM-UHFFFAOYSA-N SBZXBUIDTXKZTM      160.0      134.6      0.7275526
#> 3 Dimethametryn IKYICRRUVNIHPP-QMMMGOBSA-N IKYICRRUVNIHPP      159.6      166.1      0.7571080
#> 4      Irgarol HDHLIWCXDDZUFH-UHFFFAOYSA-N HDHLIWCXDDZUFH      162.0      171.2      0.7681929

```

8.2.6.3 Performing suspect screening The `screenSuspects()` method function introduced before is also used to perform the suspect screening in IMS workflows. The `IMSMatchParams` argument configures how IMS data is used to match suspects and features. See `?getIMSMatchParams` and the examples below for more details.

In post mobility assignment workflows, it is possible to perform the suspect screening *before* `assignMobilities()` is executed to assign the feature mobilities. In this case the `IMSMatchParams` argument can also be passed to `assignMobilities()` to filter out any suspect hits with deviating IMS data after feature mobilities have been assigned. This may be useful to e.g. first prioritize features with suspect screening and other steps before assigning mobilities to the remaining features.

Some examples are shown below:

```

# suspList is a suspect list with CCS and mobility data (see previous subsections)

# screen with default CCS matching parameters
fGroupsScr <- screenSuspects(fGroups, suspects = suspList, adduct = "[M+H]+",
  IMSMatchParams = getIMSMatchParams("CCS"))

# match suspects on mobility with a +/- 0.1 tolerance
fGroupsScr <- screenSuspects(fGroups, suspects = suspList, adduct = "[M+H]+",
  IMSMatchParams = getIMSMatchParams("mobility", window = 0.1,
  ↪ relative = FALSE))

# match suspects on CCS with a +/- 6% tolerance
fGroupsScr <- screenSuspects(fGroups, suspects = suspList, adduct = "[M+H]+",
  IMSMatchParams = getIMSMatchParams("CCS", window = 0.06,
  ↪ relative = TRUE))

# post mobility assignment workflow with prior suspect screening
# regular non-IMS suspect screening
fGroupsScr <- screenSuspects(fGroups, suspects = suspList, adduct = "[M+H]+", onlyHits =
  ↪ TRUE)

```



```

# ... do other things such as more feature prioritization
# ... and assign mobilities to the remaining features, match already screened suspects by
  ↪ CCS
fGroupsScr <- assignMobilities(fGroupsScr, mobPeaksParams =
  ↪ getDefPeakParams("bruker_ims", "piek"),
      chromPeaksParams = getDefPeakParams("chrom", "piek"),
      IMSMatchParams = getIMSMatchParams("CCS"))

```

8.2.6.4 Assigning feature mobilities from suspect data In post mobility assignment workflows, the `assignMobilities()` method function can also be used to assign feature mobilities from suspect data by setting the `fromSuspects` argument to `TRUE`. This replaces the mobility assignment step from mobilograms (step 2), and copies the assigned reference value for a suspect directly to feature data. If both `fromSuspects` and `mobPeaksParams` are set, then the mobility detection from mobilograms is performed for features without a suspect hit. If a feature has multiple hits, it is unclear which suspect value should be used, and therefore *no* suspect data will be used for mobility assignment in this case. The `IMSRange` function argument is used to derive the mobility range for the feature. Assigning mobilities from suspect data is only supported if the suspect screening was already performed prior to calling `assignMobilities()`.

Bypassing the need for peak detection for mobility assignment may be advantageous for low intensity features. However, the mobility values assigned to features are not verified against the raw data. The feature mobility boundaries are also not derived from experimental data, hence, the assigned mobility range to the feature may not cover the actual experimental range (i.e. the peak width in a mobilogram). Thus, when intensities are updated for IMS features (step 4), these can be superficially low (i.e. partial removal of signal) or high (i.e. inclusion of data of neighboring mobility peaks). With these limitations in mind, this approach is primarily intended for the following scenarios:

1. The mobility for a suspect is accurately known in advance, and no additional peak detection is needed. In this case the `mobWindow` argument is typically set to a narrow tolerance window.
2. There is no interest for accurate mobility (and CCS) assignment, and IMS data should only be used as a rough filtering step. In this scenario the `mobWindow` argument is typically set to a relatively large tolerance window.

Below are some example of how to use `assignMobilities()` to assign feature mobilities from suspect data:

```

# suspect screening must be performed in advance
# suspList is a suspect list with IMS data (see previous subsections)
fGroups <- screenSuspects(fGroups, suspects = suspList, adduct = "[M+H]+", onlyHits =
  ↪ TRUE)

# scenario #1: assume suspect data is accurate
fGroups <- assignMobilities(fGroups, fromSuspects = TRUE, mobWindow = 0.01,
      chromPeakParams = getDefPeakParams("chrom", "piek"))

# scenario #2: wide tolerance window, no interest in accurate mobility assignment
# add fallback to mobility detection for features with no (or >1) suspect hit or if no
  ↪ suspect mobility data is available
# set IMSMatchParams to also filter out suspect hits with deviating mobilities
fGroups <- assignMobilities(fGroups, fromSuspects = TRUE, mobWindow = 0.1,
      mobPeaksParams = getDefPeakParams("bruker_ims", "piek"),
      chromPeakParams = getDefPeakParams("chrom", "piek"),
      IMSMatchParams = getIMSMatchParams("CCS"))

```

For more details see the reference manual (`?assignMobilities_feat`).

8.2.7 Componentization

The algorithms that use componentization to detect adducts, isotopes, homologous series etc. (e.g. CAMERA, cliqueMS and nontarget) currently do not support IMS data. They are not supported in direct mobility assignment workflows. However, in post mobility assignment workflows they can operate on the IMS precursors (either before or after calling `assignMobilities()`).

The other componentization algorithms (e.g. intensity clustering and transformation products) optionally include IMS features and IMS precursors during the componentization. This is controlled by the `IMS` function argument to the `generateComponents()` function, and can be set as following:

1. `IMS=FALSE`: Do not consider any IMS features (only supported in post mobility assignment workflows).
2. `IMS=TRUE`: Only consider IMS features and ignore any IMS precursors.
3. `IMS="both"`: Consider both IMS features and IMS precursors.
4. `IMS="maybe"`: Consider IMS precursors if present, and IMS features otherwise. This is the default value.

In the case that IMS features are not considered (i.e. `IMS=FALSE` or `IMS="maybe"`), then the componentization data from the IMS precursors is directly copied to the IMS features. This is useful when the componentization results for IMS precursors and IMS features are expected to be similar, which is often the case for the algorithms that support the `IMS` argument.

In post mobility assignment workflows, it is possible to perform the componentization prior to calling `assignMobilities()`. In this case the componentization results can be ‘expanded’ by copying the IMS precursor results to the IMS features afterwards. This is performed by the `expandForIMS()` method function:

```
# perform componentization
components <- generateComponents(fGroups, "intclust")
# assign mobilities
fGroups <- assignMobilities(fGroups, mobPeaksParams = getDefPeakParams("bruker_ims",
  ↪ "piek"),
                           chromPeaksParams = getDefPeakParams("chrom", "piek"))
# expand componentization results for IMS features
components <- expandForIMS(components, fGroups)
```

8.2.8 Annotation

IMS workflows bring several advantages to feature annotation. The `generateMSPeakLists()` function automatically filters MS and MS/MS spectra by the mobility range assigned to the feature, which can considerably improve the quality of peak lists. Furthermore, CCS data can be assigned to compound annotation candidates and used to eliminate unlikely candidates.

NOTE In post mobility assignment workflows it is important that the `assignMobilities()` method function is called *prior* to any feature annotation steps in order to take advantage IMS data during feature annotation.

8.2.8.1 MS peak lists The `generateMSPeakLists()` function automatically filters MS and MS/MS spectra by mobility data assigned to features, and is therefore used similarly like non-IMS workflows. The function supports Bruker PASEF experiments, and can therefore assign m/z and mobility filtered MS/MS data to features. In post mobility assignment workflows, the MS and MS/MS data that will be assigned to IMS precursors are based on combined spectra for each IMS frame.

The m/z values in IMS-HRMS data are typically not (e.g. Agilent) or partially (e.g. Bruker) centroided. Since feature annotation algorithms are usually limited to centroided data, `generateMSPeakLists()` performs an additional centroiding step for IMS-HRMS data. The centroiding parameters, and some other advanced IMS specific parameters, are configured through the averaging parameters given to `generateMSPeakLists()` (`avgFeatParams` arguments). The default parameters can be changed with the `IMS` argument to `getDefAvgPListParams()`, which can be set to "bruker" or "agilent" for Bruker and Agilent IMS data, respectively. The default value for `IMS` is determined from the value determined from the previously discussed limits file. For example:

```
# generate peak lists with default parameters for Agilent data
mslists <- generateMSPeakLists(fGroups, avgFeatParams = getDefAvgPListParams(IMS =
  ↪ "agilent"))
```

see the reference manual for more details (`?getDefAvgPListParams`).

8.2.8.2 Compound annotation Candidates from compound annotations can be matched with IMS data for features, similarly as was discussed for candidates from suspect screening. The `assignMobilities()` method function works similarly to assign IMS data to candidates and perform mobility and CCS conversions. The `filter()` method function can subsequently be used to eliminate candidates that match poorly with features (discussed later).

The `assignMobilities()` function has similar arguments as the method for suspect screening:

Function argument	Description
<code>fGroups</code>	The feature groups for which the <code>compounds</code> were generated.
<code>from</code>	Specifies from where IMS data is added. See the suspects section for more details.
<code>adduct</code>	Should match the value of the <code>adduct</code> argument that was passed to <code>generateCompounds()</code> . Set to <code>NULL</code> if adduct annotations are available for features or in sets workflows.
<code>CCSParams</code>	Parameters for the mobility and CCS conversions. See the previous discussion.
<code>overwrite</code>	Set to <code>TRUE</code> to overwrite any existing IMS data in the candidate list.
<code>IMS</code>	For which type of features IMS data should be added to the candidates: <code>FALSE</code> (only IMS precursors), <code>TRUE</code> (only IMS features, default), <code>"both"</code> (both IMS precursors and IMS features) or <code>"maybe"</code> (IMS precursors if available, otherwise IMS features).

NOTE If MetFrag with the PubChemLite database with CCS is used for compound annotation, then the CCS data is copied from the database and there is no need to set the `from` argument.

Some examples:

```
# generate compound candidates
compounds <- generateCompounds(fGroups, mslists, "metfrag", adduct = "[M+H]+", database =
  ↪ "comptox")
# perform CCS prediction with C3SDB and convert the values to mobilities
compounds <- assignMobilities(compounds, fGroups, from = "c3sdb", adduct = "[M+H]+",
  CCSParams = getCCSParams("bruker"))

# generate compound candidates
# NOTE: if PubChemLite with CCS data is installed, then CCS data is automatically added
compounds <- generateCompounds(fGroups, mslists, "metfrag", adduct = "[M+H]+", database =
  ↪ "pubchemlite")
```

```

# overwrite CCS data from a custom library (candidates not in the library will be
  ↳ untouched)
IMSLib <- data.frame(
  name = c("Caffeine", "Acetaminophen"), # name column is not mandatory
  InChIKey = c("RYYVLZVUVIJVGH-UHFFFAOYSA-N", "RZVAJINKPMORJF-UHFFFAOYSA-N"),
  CCS = c(150, 160)
))
compounds <- assignMobilities(compounds, fGroups, from = IMSLib, adduct = "[M+H]+",
  overwrite = TRUE)

```

For more details see the reference manual (`?assignMobilities_comp`).

8.2.8.3 Copying annotation data to IMS features with similar MS/MS data The number of features in post mobility assignment workflows can be very large, considering that both IMS features and their IMS precursors are present. Since formula and compound annotation is primarily influenced by MS/MS data, it is possible to skip annotation for IMS features with highly similar MS/MS data as their IMS precursor, and copy the annotations from the IMS precursor result instead. The `minIMSSpecSim` argument to `generateFormulas()` and `generateCompounds()` sets the minimum spectral similarity threshold for when MS/MS data is considered similar.

```

# generate formulas: copy IMS precursor results when the spectral similarity is at least
  ↳ 0.9
formulas <- generateFormulas(fGroups, mslists, "genform", adduct = "[M+H]+",
  ↳ minIMSSpecSim = 0.9)

```

8.2.9 Sets workflows

There is basic functionality to combine IMS workflows and sets workflows. However, an important limitation currently is that there is no way to group IMS features across the different sets. This is because mobility values among polarities (e.g. $[M+H]^+$ and $[M-H]^-$) typically differ slightly and there is not a straightforward way to normalize for this. However, in post mobility assignment workflows grouping of IMS precursors across sets still occurs (similar to non-IMS workflows). Hence, the links between IMS features and IMS precursors can still give some indication which IMS features across sets are related. Thus, combined IMS and sets workflows are therefore mainly useful to e.g. process positive and negative sample analyses in one workflow, but lack other advantages of non-IMS sets workflows, such as easily relating features among sets and improving feature annotation with combined positive/negative MS/MS data.

Like non-IMS workflows, the `makeSet()` function is used to initialize the sets workflow.

8.3 Processing data

The data processing functionality of `patRoan` that was discussed before equally applies to IMS workflows. In addition, some extra functionality exists to inspect and filter the data. These are discussed in the next subsections.

8.3.1 Updating feature group properties

The `updateGroups()` function to update feature group properties (retention time and m/z) was briefly introduced before. This function can also be used to update the assigned mobilities and CCS values to e.g. improve their accuracy after eliminating unwanted features:

```
fGroups <- filter(fGroups, ...) # perform filtering steps

fGroups <- updateGroups(fGroups) # update all by default: retention time, m/z, mobility
  ↳ and CCS (if present)
fGroups <- updateGroups(fGroups, what = c("mobility", "CCS")) # only update mobility and
  ↳ CCS
```

8.3.2 Inspecting and plotting data

The `plotMobilograms()` method function is used to plot extracted ion mobilograms (EIMs), and works very similar as the `plotChroms()` function introduced before. The function arguments are mostly equal, and the `EIMParams` argument is similarly used to configure advanced parameters (see `?getDefEIMParams` for details).

Most other plotting functions have an `IMS` argument that controls the inclusion of IMS features and IMS precursors in the plots. This argument is similar as discussed before for e.g. componentization and feature annotation and supports the following values:

- `IMS=FALSE`: Do not plot any IMS features (only supported in post mobility assignment workflows).
- `IMS=TRUE`: Only plot IMS features and ignore any IMS precursors.
- `IMS="both"`: Plot both IMS features and IMS precursors.
- `IMS="maybe"`: Plot IMS precursors if present, and IMS features otherwise. This is the default value.

The default is "maybe" since most plotting functions primarily use LC-MS data and therefore produce similar plots for IMS features and their IMS precursors.

The `spectrumSimilarityIMS()` method function calculates the spectrum similarity between MS (or MS/MS) data of a IMS feature and its IMS precursor. This function is internally used by feature annotation if the `minIMSSpecSim` argument is set, and uses the `spectrumSimilarity()` function introduced earlier.

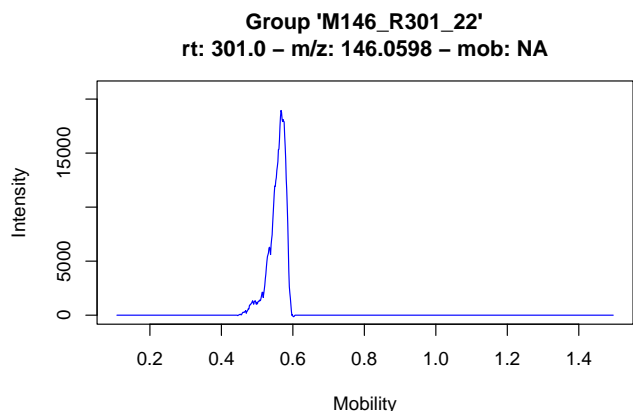
The reporting functionality of `patRoön` contains various functionality that facilitates processing of IMS data.

Some examples are shown below:

```
# plot mobilogram for first IMS precursor
plotMobilograms(fGroups[1, "M146_R301_22"])
```

```
#> Using 'mstoolkit' backend for reading MS data.
```

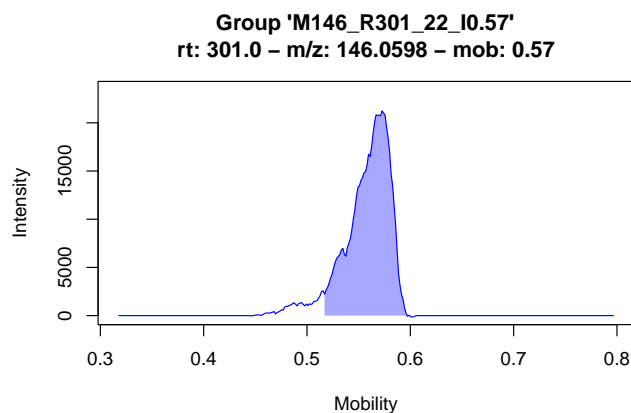
```
#> =====
```



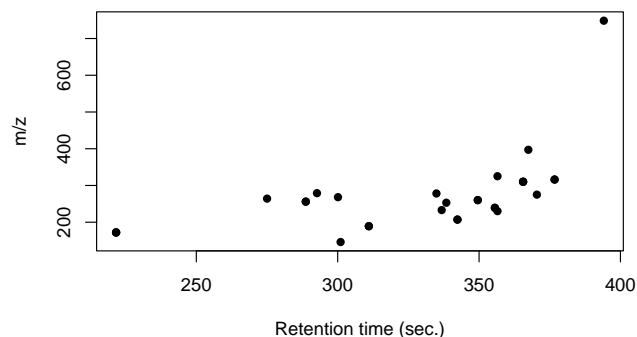
```
# plot mobilogram for an IMS feature: this highlights the mobility (range) assigned to
↳ the feature
# NOTE: the mobility is appended to the feature group name
plotMobilograms(fGroups[1, "M146_R301_22_I0.57"], showPeakArea = TRUE, showFGroupRect =
↳ FALSE)
```

```
#> Using 'mstoolkit' backend for reading MS data.
```

```
#> =====
```



```
plot(fGroups, IMS = TRUE) # make a scatter plot with only IMS features
```



```
# returns a table with spectral similarities between IMS features and their IMS
↳ precursors
spectrumSimilarityIMS(mslists, fGroups)
```

```
#>
#>          group ims_precursor_group similarity
#>          <char>          <char>          <num>
#> 1:  M146_R301_22_I0.57      M146_R301_22  0.4591192
#> 2:  M172_R222_60_I0.60      M172_R222_60  0.7959959
#> 3:  M172_R222_60_I0.62      M172_R222_60  0.8223180
#> 4:  M172_R222_60_I0.64      M172_R222_60  0.8559733
#> 5:  M189_R311_73_I0.65      M189_R311_73  0.2274158
#> ---
#> 26: M316_R377_214_I0.83      M316_R377_214  0.7105095
#> 27: M316_R377_214_I1.26      M316_R377_214  0.1660779
#> 28: M325_R357_234_I0.87      M325_R357_234  0.5189992
#> 29: M397_R367_277_I0.85      M397_R367_277  0.7907177
#> 30: M748_R394_360_I1.26      M748_R394_360  0.3859687
```

8.3.3 Subsetting and filtering data

The following filters are specifically for IMS workflows available:

Filter	Classes	Remarks
IMS	featureGroups	Only keep IMS precursors (IMS=FALSE), IMS features (IMS=TRUE), both (IMS="both") or IMS precursors if available and IMS features otherwise (IMS="maybe"). The IMS argument is also available for the <code>[]</code> operator.
withIMSPrecursor	featureGroups	Only keep IMS features that have an IMS precursor.
applyIMS	featureGroups	Only apply other filters to a subset of features. Should be set like the IMS argument.
IMSRangeParams	features, featureGroups, compounds	Only keep features/annotations within the specified mobility range. The range is configured with the <code>getIMSRangeParams()</code> function.
IMSMatchParams	compounds	Only keep candidates that match reference IMS data. See its description in suspect screening workflows for more details.

Some examples are shown below:

```
# keep only IMS features
# NOTE: this effectively results in data that is equal to a direct IMS workflow
fGroupsMob <- filter(fGroups, IMS = TRUE)
fGroupsMob <- fGroups[IMS = TRUE] # same as above

# only keep IMS precursors (post mobility assignment workflows only)
# NOTE: this effectively results in data that is equal to before `assignMobilities()` was
↳ called
fGroupsIMSPar <- filter(fGroups, IMS = FALSE)

# perform various filtering steps and ensure that only IMS features with an IMS precursor
↳ are kept
fGroupsF <- filter(fGroups, absMinIntensity = 1E5, relMinReplicateAbundance = 1,
                  withIMSPrecursor = TRUE)

# perform various filtering steps, but not on IMS features
fGroupsF <- filter(fGroups, absMinIntensity = 1E5, relMinReplicateAbundance = 1,
                  applyIMS = FALSE)

# only keep compound candidates within a defined CCS range
compoundsF <- filter(compounds, IMSRangeParams = getIMSRangeParams("CCS", 150, 200))

# only keep compound candidates with matching CCS (6% tolerance)
compoundsF <- filter(compounds, IMSMatchParams = getIMSMatchParams("CCS", window = 0.06,
↳ relative = TRUE))
```

8.4 Example workflow

The example below shows a complete IMS workflow which uses post mobility assignment for features, and includes matching of suspect and compound annotation candidates by CCS.

NOTE The newProject tool supports the creation of IMS workflows, and makes it easy to create and explore different types of IMS workflows.

```
# anaInfo should be a data.frame with the analysis information

# convert raw data to ims and to centroid data
convertMSFiles(anaInfo, typeFrom = "raw", formatFrom = "bruker_ims", typeTo = "ims",
  ↪ formatTo = "mzML",
    algorithm = "pwiz")
convertMSFiles(anaInfo, typeFrom = "ims", formatFrom = "mzML", typeTo = "centroid",
  ↪ formatTo = "mzML",
    algorithm = "imscollapse")

# perform feature detection and grouping
fList <- findFeatures(anaInfo, "openms")
fGroups <- groupFeatures(fList, "openms")

# perform basic filtering
fGroups <- filter(fGroups, absMinIntensity = 1E4, relMinReplicateAbundance = 1)

# assign mobilities and CCS values
CCSParams <- getCCSParams("mason-schamp_1/k")
fGroups <- assignMobilities(fGroups, mobPeaksParams = getDefPeakParams("bruker_ims",
  ↪ "piek"),
    chromPeaksParams = getDefPeakParams("chrom", "piek"),
    CCSParams = CCSParams)

# suspect screening
susplist <- read.csv("suspects.csv")
# add CCS data from PubChemLite and convert them to mobilities
susplist <- assignMobilities(susplist, from = "pubchemlite", adducts = "[M+H]+",
    CCSParams = CCSParams)
# screen suspects and only keep fair matches (3% CCS tolerance)
IMSMatchParams <- getIMSMatchParams("CCS", window = 0.03, relative = TRUE)
fGroups <- screenSuspects(fGroups, suspects = susplist, adduct = "[M+H]+",
    IMSMatchParams = IMSMatchParams, onlyHits = TRUE)

# perform feature annotation
mslists <- generateMSPeakLists(fGroups)
formulas <- generateFormulas(fGroups, mslists, "genform", adduct = "[M+H]+")
# skip compound annotation for IMS features with similar MS/MS data as their IMS
  ↪ precursor
compounds <- generateCompounds(fGroups, mslists, "metfrag", adduct = "[M+H]+", database =
  ↪ "pubchemlite",
    minIMSSpecSim = 0.9)
# NOTE: assume that PubChemLite with CCS data is installed, so only mobility conversions
  ↪ needs to be applied
compounds <- assignMobilities(compounds, fGroups, adduct = "[M+H]+", CCSParams =
  ↪ CCSParams)
# filter out deviating candidates
compounds <- filter(compounds, IMSMatchParams = IMSMatchParams)

report(fGroups, mslists, formulas, compounds)
```

The patRoonaDataIMS package contains example data and can be used to run this workflow. The data is

already converted to various formats, therefore, the `convertMSFiles()` steps can be skipped when using this data. To use its data, simply assign the `anaInfo` variable:

```
anaInfo <- patRoonDataIMS::exampleAnalysisInfo("positive") # positive IMS-HRMS data
```

9 Advanced usage

9.1 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``).

9.2 Default numeric limits and tolerances

The functions in `patRoan` use various numeric limits and tolerances to configure e.g. tolerances applied to match retention times and m/z values. Since `patRoan` 3.0 the defaults for these limits are centralized, which simplifies configuration and switching defaults between different HRMS instruments.

The configuration is done through the `limits.yml` file. This is an easily editable YAML text file, and by default looks like this:

```
general:
  version: 1
  IMS: bruker
retention:
  very_narrow: 2
  narrow: 6
  medium: 12
  wide: 30
mz:
  very_narrow: 0.001
  narrow: 0.002
  medium: 0.005
  wide: 0.02
  narrow_rel: 5
  medium_rel: 10
mobility_bruker:
  very_narrow: 0.01
  narrow: 0.02
  medium: 0.04
  wide: 0.2
  medium_rel: 0.05
mobility_agilent:
  very_narrow: 0.1
  narrow: 0.2
  medium: 0.4
  wide: 2
  medium_rel: 0.05
CCS:
  medium: 10
  medium_rel: 0.05
```

The `IMS` setting under `general` specifies which IMS defaults are applied: this should be `bruker` or `agilent`. Depending on this setting, defaults are either taken from the `mobility_bruker` or `mobility_agilent` setting blocks.

If you used the `newProject` tool to create a new project, the `limits.yml` file is automatically copied to the project directory. Otherwise, it can be created by the `genLimitsFile()` function:

```
genLimitsFile() # copy default limits.yml file to current working directory
genLimitsFile("my/path/limits.yml", IMS = "agilent") # generate `my/path/limits.yml` with
↳ Agilent IMS defaults
```

More information on the limits can be found in the reference manual (`?limits`).

9.3 Feature intensity normalization

Feature intensities are often compared between sample analyses, for instance, to evaluate trends between sample points. However, matrix effects, varying detector sensitivity and differences in analysed sample amount may complicate such comparison. For this reason, it may be desired to *normalize* the feature intensities.

The `normInts()` function is used to normalize feature intensities (peak heights and areas). Two different types are supported:

1. **Feature normalization:** normalization occurs by intensities within the same sample analysis
2. **Group normalization:** normalization occurs by intensities among features within the same group

Both normalization types can be combined.

9.3.1 Feature normalization

Feature normalization itself supports the following normalization methods:

Method	Usage	Description
TIC	<code>normInts(featsNorm = "tic", ...)</code>	Normalizes by the combined intensity of all features, also known as the Total Ion Current (TIC).
Internal Standard	<code>normInts(featsNorm = "istd", ...)</code>	Uses internal standards (IS) to normalize feature intensities.
Concentration	<code>normInts(featsNorm = "conc", ...)</code>	Normalizes feature intensities of a sample analysis by its <i>normalization concentration</i> (explained below).
None	<code>normInts(featsNorm = "none", ...)</code>	Performs no feature normalization. Set this if you only want to perform group normalization (discussed in the next section).

9.3.1.1 Normalization concentration All methods (except "none") are influenced by the *normalization concentration*, which is a property set for each sample analysis. For IS normalization, this should equal the concentration of the IS present in the sample. Otherwise the normalization concentration resembles the injected sample amount. The normalization concentration is defined in the `norm_conc` column of the analysis information. For example:

```
# obtain analysis information as usual, but add normalization concentrations.
# The blanks are set to NA, and will therefore not be normalized.
generateAnalysisInfo(fromCentroid = patRoofData::exampleDataPath(),
  replicate = c(rep("solvent", 3), rep("standard", 3)),
  blank = "solvent",
  norm_conc = c(NA, NA, NA, 2, 2, 1))
```

```
#>      analysis                                path_centroid path_raw path_profile path_im
#> 1 solvent-pos-1 /usr/local/lib/R/site-library/patRoofData/extdata/pos
#> 2 solvent-pos-2 /usr/local/lib/R/site-library/patRoofData/extdata/pos
#> 3 solvent-pos-3 /usr/local/lib/R/site-library/patRoofData/extdata/pos
#> 4 standard-pos-1 /usr/local/lib/R/site-library/patRoofData/extdata/pos
#> 5 standard-pos-2 /usr/local/lib/R/site-library/patRoofData/extdata/pos
#> 6 standard-pos-3 /usr/local/lib/R/site-library/patRoofData/extdata/pos
```

The normalization concentration does not need to be an absolute value. In the end, what matters are the relative numbers between the sample analyses. For example, if the concentrations for two analyses are `c(1, 2)` or `c(1.5, 3.0)` the normalization occurs the same. Setting the concentration to `NA` (or `0`) will skip normalization for an analysis. If the normalization concentration is absent from the analysis information it will be defaulted to `1`.

9.3.1.2 Internal standard normalization For IS normalization an internal standard list should be specified with the properties of the internal standards. Essentially, the format of this list is exactly the same as a suspect list. Example lists can be found in the `patRoonaData` package:

```
patRoonaData::ISTDListPos[1:5, ]
```

```
#>           name          formula    rt
#> 1 1H-benzotriazole-D4      C6[2]H4HN3 268.1
#> 2      Atenolol-D7      C14[2]H7H15N2O3 213.5
#> 3      Atrazine-D5       C8[2]H5H9C1N5 336.5
#> 4  Bezafibrate-D6      C19[2]H6H14C1N04 351.7
#> 5      Climbazole-D4    C15[2]H4H13C1N2O2 359.1
```

As can be seen from above, labelled isotopes can be specified with square brackets, *e.g.* `[2]H` for deuterium. The next step is to perform the normalization with `normInts()`:

```
fGroupsNorm <- normInts(fGroups, featNorm = "istd", standards = patRoonaData::ISTDListPos,
  ↪ adduct = "[M+H]+",
  ISTDRTWindow = 20, ISTDmZWindow = 200, minISTDs = 2)
```

This will do the following:

- Perform a suspect screening to find the specified IS (`standards` argument).
- Remove the IS candidates which are absent in one or more of the analyses to be normalized.
- Select IS candidates for each feature group, based on close retention time (`ISTDRTWindow` argument), m/z (`ISTDMZWindow` argument) and a minimum number (`minISTDs`). If the number of IS candidates within specified retention time and m/z windows is below `minISTDs`, the close(st) candidate(s) outside these windows are additionally chosen.
- Normalization of features is performed with the combined IS intensities.

To evaluate the assignments for a particular feature group, the `internalStandardAssignments()` function and `plotGraph()` functions can be used:

```
fg <- names(fGroupsNorm)[2]
internalStandardAssignments(fGroupsNorm)[[fg]] # IS assignments for 2nd feature group
```

```
#> [1] "M221_R336_292" "M284_R323_569" "M213_R340_263"
```

```
plotGraph(fGroupsNorm) # interactively explore assignments
```

Explore connections by dragging/zooming/selecting.
Smaller retention time differences have wider edges.

Select by feat group ▼

Select by ISTD ▼

9.3.1.3 Other methods Like IS normalization, other feature normalization methods also occurs with `normInts()`:

```
fGroupsNorm <- normInts(fGroups, featNorm = "tic") # TIC normalization
fGroupsNorm <- normInts(fGroups, featNorm = "conc") # Concentration normalization
```

9.3.2 Group normalization

Normalizing feature intensities among group member is easily performed by setting `groupNorm=TRUE`:

```
# only perform group normalization
fGroupsNorm <- normInts(fGroups, featNorm = "none", groupNorm = TRUE)
# first perform TIC feature normalization and then group normalization
fGroupsNorm <- normInts(fGroups, featNorm = "tic", groupNorm = TRUE)
```

9.3.3 Using normalized intensities

The normalized intensity (peak heigh/area) values can easily be obtained with `as.data.table()`:

```
as.data.table(fGroupsNorm, normalized = TRUE)[1:5]
```

```
#>      group      ret      mz standard-pos-1_intensity standard-pos-2_intensity standard-pos-3_in
#>      <char>    <num>    <num>                <num>                <num>                <num>
#> 1: M109_R192_20 191.8717 109.0759                2.328459                2.1068991                0
#> 2: M111_R330_23 330.4078 111.0439                0.476554                0.4156571                0
#> 3: M114_R269_25 268.6906 114.0912                1.006808                1.1271519                0
#> 4: M116_R317_29 316.7334 116.0527                3.804086                3.8240928                2
#> 5: M120_R268_30 268.4078 120.0554                3.376374                3.0432604                1
```

```
# can be combined with other parameters
```

```
as.data.table(fGroupsNorm, normalized = TRUE, average = TRUE, areas = TRUE)[1:5]
```

```
#>      group      ret      mz standard_intensity                                ISTD_
#>      <char>    <num>    <num>                <num>                                <num>
#> 1: M109_R192_20 191.8717 109.0759                3.2597655                                M280_R212_561,M274_
#> 2: M111_R330_23 330.4078 111.0439                0.2753524                                M221_R336_292,M284_R323_569,M213_
#> 3: M114_R269_25 268.6906 114.0912                0.8325869                                M300_R262_608,M275_
#> 4: M116_R317_29 316.7334 116.0527                2.6500817 M284_R323_569,M198_R310_215,M285_R301_570,M221_
#> 5: M120_R268_30 268.4078 120.0554                1.8965138                                M300_R262_608,M275_
```

```
# feature values (no need to set normalized=TRUE)
```

```
as.data.table(fGroupsNorm, features = TRUE)[1:5, .(group, analysis, intensity_rel,
  ↪ area_rel)]
```

```
#>      group      analysis intensity_rel area_rel
#>      <char>    <char>        <num>    <num>
#> 1: M109_R192_20 standard-pos-1    2.3284588 3.9777827
#> 2: M109_R192_20 standard-pos-2    2.1068991 4.0198440
#> 3: M109_R192_20 standard-pos-3    0.9688233 1.7816697
#> 4: M111_R330_23 standard-pos-1    0.4765540 0.3352008
#> 5: M111_R330_23 standard-pos-2    0.4156571 0.3251259
```

Several other `patRoan` functions also accept the `normalized` argument to use normalized data, such as `plotInt()` (discussed here), `plotVolcano()` (discussed here) and `generateComponents()` with intensity clustering (discussed here).

9.3.4 Default normalization

If normalized data is requested (`normalized=TRUE`, see previous section) and `normInts()` was *not* called on the feature group data, a *default normalization* will occur. This is nothing more than a group normalization (`normInts(groupNorm=TRUE, ...)`), and was mainly implemented to ensure backwards compatibility with previous `patRoan` versions.

9.3.5 IMS workflows

In IMS workflows with post mobility assignment `normInts()` will not consider any IMS features for the assignment of internal standards and calculation of TICs and the normalized intensity values for IMS features will be copied from IMS precursors.

9.4 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 `patRoan`. 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 functionality. Please see the reference manual for more information (e.g. `?feature-optimization`).

NOTE The SIRIUS and SAFD algorithms are currently not (yet) supported.

9.4.1 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 or KPIC2 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 and KPIC2) we need to set the grouping and retention alignment parameters in two different nested lists: these are `groupArgs/retcorArgs` (algorithm="xcms"), `groupParams/retAlignParams` (algorithm="xcms3") or `groupArgs/alignArgs` (algorithm="kpic2").

```
pSetFG <- list(groupParams = list(bw = c(20, 30))) # xcms3
pSetFG <- list(retcorArgs = list(gapInit = c(0, 7))) # xcms
pSetFG <- list(groupArgs = list(mz_weight = c(0.3, 0.9))) # kpic2
```

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`).

9.4.2 Processing optimization 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] 4
#>
#> $mzPPM
#> [1] 13.5
#>
#> $minFWHM
#> [1] 1.6
#>
#> $maxFWHM
#> [1] 36
```

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

```

#> A featuresOpenMS object
#> Hierarchy:
#> features
#>      |-- featuresOpenMS
#> ---
#> Object size (indication): 639.6 kB
#> Algorithm: openms
#> Total feature count: 3948
#> Average feature count/analysis: 3948
#> Has IMS data: no
#> Analyses: solvent-pos-1 (1 total)
#> Replicates: solvent-pos (1 total)
#> Replicates used as blank: solvent-pos (1 total)

```

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

```

optimizedParameters(ftOpt, 1) # best settings for first parameter set

```

```

#> $chromFWHM
#> [1] 4
#>
#> $mzPPM
#> [1] 13.5
#>
#> $minFWHM
#> [1] 1.6
#>
#> $maxFWHM
#> [1] 36

```

```

optimizedParameters(ftOpt, 1, 1) # best settings for first parameter set and experiment
  ↳ iteration

```

```

#> $chromFWHM
#> [1] 5
#>
#> $mzPPM
#> [1] 10
#>
#> $minFWHM
#> [1] 1
#>
#> $maxFWHM
#> [1] 35

```

```

optimizedObject(ftOpt, 1) # features object with best settings for first parameter set

```

```

#> A featuresOpenMS object
#> Hierarchy:
#> features
#>      |-- featuresOpenMS

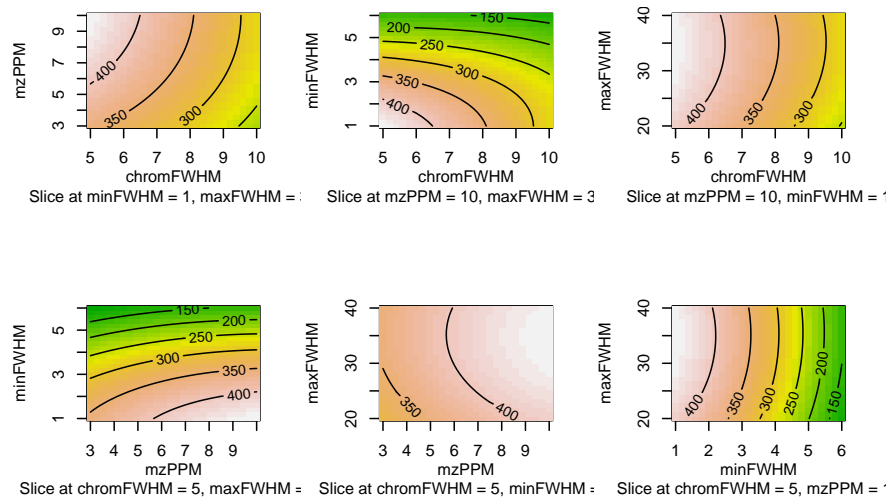
```



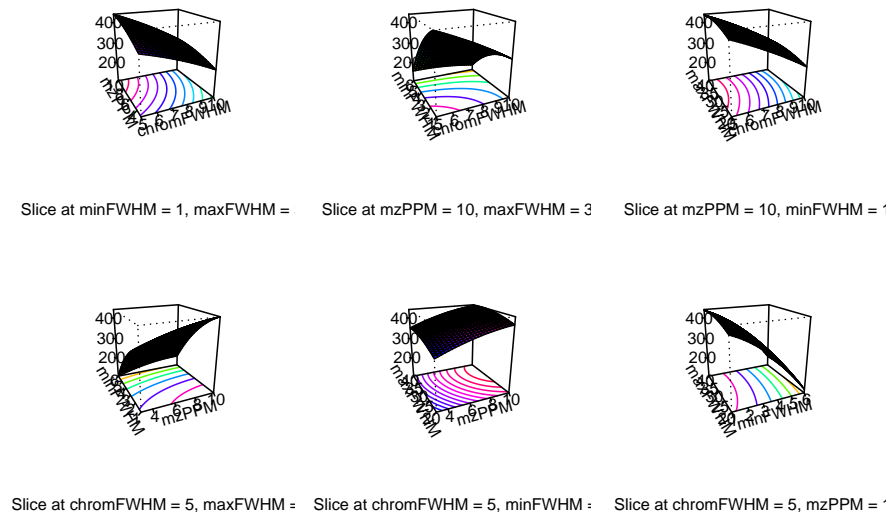
```
#> ---
#> Object size (indication): 639.6 kB
#> Algorithm: openms
#> Total feature count: 3948
#> Average feature count/analysis: 3948
#> Has IMS data: no
#> Analyses: solvent-pos-1 (1 total)
#> Replicates: solvent-pos (1 total)
#> Replicates used as blank: solvent-pos (1 total)
```

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`).

9.5 Chromatographic peak qualities

The algorithms used by `findFeatures` detect chromatographic peaks automatically to find the features. However, it is common that not all detected features have ‘proper’ chromatographic peaks, and some features could be just noise. The MetaClean R package supports various quality measures for chromatographic peaks. The quality measures include Gaussian fit, symmetry, sharpness and others. In addition, MetaClean averages all feature data for each feature group and adds a few additional group specific quality measures (e.g. retention time consistency). Please see Chetnik et al. (2020) for more details. The calculations are integrated into `patRoan`, and are easily performed with the `calculatePeakQualities()` generic function.

```
fList <- calculatePeakQualities(fList) # calculate for all features
fGroups <- calculatePeakQualities(fGroups) # calculate for all features and groups
```

Most often the `featureGroups` method is only used, unless you want to filter features (discussed below) prior to grouping.

An extension in `patRoan` is that the qualities are used to calculate *peak scores*. The score for each quality measure is calculated by normalizing and scaling the values into a 0–1 range, where zero is the worst and one the best. Note that most scores are relative, hence, the values should only be used to compare features among each other. Finally, a `totalScore` is calculated which sums all individual scores and serves as a rough overall score indicator for a feature (group).

The qualities and scores are easily obtained with the `as.data.table()` function.

```
# (limit rows/columns for clarity)
as.data.table(fList)[1:5, 26:30]
```

```
#>      GaussianSimilarityScore SharpnessScore TPASRScore ZigZagScore totalScore
#>           <num>           <num>           <num>           <num>           <num>
#> 1:      0.6314046    3.443351e-02    0.9951840    0.9103221    6.301669
#> 2:      0.9633994    9.900530e-10    0.9938460    0.3565674    6.512552
#> 3:      0.3613087    7.565147e-10    0.7770017    0.9999449    5.627723
#> 4:      0.9758176    8.600747e-03    0.6635416    0.9963211    5.519801
#> 5:      0.3676624    1.000000e+00    0.9896699    0.8435805    5.824171
```

```
# the qualities argument is necessary to include the scores.
# valid values are: "quality", "score" or "both"
as.data.table(fGroups, qualities = "both")[1:5, 25:29]
```

```
#>      TPASRScore ZigZagScore ElutionShiftScore RetentionTimeCorrelationScore totalScore
#>           <num>           <num>           <num>           <num>           <num>
#> 1: 0.75433983    0.9961342           0.8985829           0.9941415    8.011216
#> 2: 0.09267299    0.9743659           0.9974944           0.7752439    6.123999
#> 3: 0.61668299    0.9170699           0.9333565           0.9768228    7.514189
#> 4: 0.83855430    0.8906915           0.9550289           0.9928579    8.478425
#> 5: 0.98761665    0.8666283           0.6912845           0.9985570    8.858657
```

The feature quality values can also be reviewed interactively with reports generated with `report` (see Reporting) and with `checkFeatures` (see here). The `filter` function can be used filter out low scoring features and feature groups:

```

# only keep features with at least 0.3 Modality score and 0.5 symmetry score
fList <- filter(fList, qualityRange = list(ModalityScore = c(0.3, Inf),
                                           SymmetryScore = c(0.5, Inf)))

# same as above
fGroups <- filter(fGroups, featQualityRange = list(ModalityScore = c(0.3, Inf),
                                                    SymmetryScore = c(0.5, Inf)))

# filter group averaged data
fGroups <- filter(fGroups, groupQualityRange = list(totalScore = c(0.5, Inf)))

```

9.5.1 Applying machine learning with MetaClean

An important feature of MetaClean is to use the quality measures to train a machine learning model to automatically recognize ‘good’ and ‘bad’ features. `patRoön` provides a few extensions to simplify training and using a model. Furthermore, while `MetaClean` was primarily designed to work with `XCMS`, the extensions of `patRoön` allow the usage of data from all the algorithms supported by `patRoön`.

The `getMCTrainData` function can be used to convert data from a feature check session to training data that can be used by MetaClean. This allows you to use interactively select good/bad peaks. The workflow looks like this:

```

# untick the 'keep' checkbox for all 'bad' feature groups
checkFeatures(fGroupsTrain, "train_session.yml")

# get train data. This gives comparable data as MetaClean::getPeakQualityMetrics()
trainData <- getMCTrainData(fGroupsTrain, "train_session.yml")

# use train data with MetaClean with MetaClean::runCrossValidation(),
# MetaClean::getEvaluationMeasures(), MetaClean::trainClassifier() etc
# --> see the MetaClean vignette for details

```

Once you have created a model with MetaClean it can be used with the `predictCheckFeaturesSession()` function:

```

predictCheckFeaturesSession(fGroups, "model_session.yml", model)

```

This will generate another *check session file*: all the feature groups that are considered good will be with a ‘keep’ state, the others without. As described elsewhere, the `checkFeatures` function is used to review the results from a session and the `filter` function can be used to remove unwanted feature groups. Note that `calculatePeakQualities()` must be called before `getMCTrainData`/`predictCheckFeaturesSession` can be used.

NOTE `MetaClean` only predicts at the feature group level. Thus, only the kept feature groups from a *feature check session* will be used for training, and any individual features that were marked as removed will be ignored.

9.6 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, and the `getPICSet()` function can be used to convert `features` to KPIC2 data.

NOTE None of the exporting and conversion functions support data from sets workflows or IMS workflows. For this reason, the `set` argument must be given to specify for which features should be exported in sets workflows, and any IMS features will *not* be exported.

Some examples for these functions are shown below.

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

# convert features to xcmsSet.
# NOTE: loadRawData should only be FALSE when the analysis data files cannot be
# loaded by the algorithm (e.g. when features were obtained with DataAnalysis and data
# was not exported to mz(X)ML)
xset <- getXCMSSet(fList, loadRawData = TRUE)
xsetg <- getXCMSSet(fGroups, loadRawData = TRUE) # get grouped xcmsSet

# using the new XCMS3 interface
xdata <- getXCMSnExp(fList)
xdata <- getXCMSnExp(fGroups)
xdataPos <- getXCMSnExp(fGroupsSet, set = "positive") # sets workflow

# KPIC2 conversion. Like XCMS it optionally loads the raw data.
picSet <- getPICSet(fList, loadRawData = TRUE)
```

9.7 Algorithm consensus

With `patRoan` 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 advantageous 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
<code>obj, ...</code>	All	Two or more objects (of the same type) that should be compared to generate the consensus.

Argument	Classes	Remarks
<code>compThreshold</code> , <code>relAbundance</code> , <code>absAbundance</code> , <code>formThreshold</code>	<code>compounds</code> , <code>formulas</code> , <code>featureGroupsComparison</code>	The minimum overlap (relative/absolute) for a result (feature, candidate) to be kept.
<code>uniqueFrom</code>	<code>compounds</code> , <code>formulas</code> , <code>transformationProductsStructure</code> , <code>featureGroupsComparison</code>	Only keep <i>unique</i> results from specified objects.
<code>uniqueOuter</code>	<code>compounds</code> , <code>formulas</code> , <code>transformationProductsStructure</code> , <code>featureGroupsComparison</code>	Should be combined with <code>uniqueFrom</code> . If TRUE then only results are kept which are <i>also</i> unique between the objects specified with <code>uniqueFrom</code> .

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 with plotting functions described previously for feature groups.

Some examples are shown below

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

TPsCons <- consensus(TPsLib, TPsBT) # combine library and BioTransformer TPs

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
```

9.8 Background removal in MS/MS data

Even if data-dependent acquisition (DDA) is used to acquire MS/MS data, there may still be a substantial number of background peaks present in the MS/MS spectra. The filtering section introduced several filters to remove these.

Another approach to remove background is to identify frequently occurring background peaks, which typically are a sign of contamination in the analytical system, and subsequently remove these. The `getBGMSMSPeaks()` function inspects all MS/MS data in a set of blank analyses and identifies frequently occurring peaks.

```
# get background MS/MS peaks from blank analyses
bgPeaks <- getBGMSMSPeaks(anaInfo, replicates = "solvent-pos")
```

```
#> Averaging the spectra for each of the 3 analyses
#> Using 'mzr' backend for reading MS data.
#> =====
#> Averaging analyses averaged spectra... Done!
```

```
bgPeaks
```

```
#>      mz intensity abundance_rel_ana abundance_abs_ana abundance_rel_spec abundance_abs_spec
#>      <num>      <num>          <num>          <num>          <num>          <num>
#> 1:  98.97532  6325.059              1              3          0.4669805          233.3333
#> 2: 100.11211  4400.579              1              3          0.2261462          113.0000
#> 3: 101.01768  3430.920              1              3          0.2321316          116.0000
#> 4: 102.99709  3728.637              1              3          0.3108626          155.3333
#> 5: 107.08547  4883.053              1              3          0.5610356          280.3333
#> ---
#> 19: 139.07513  4069.568              1              3          0.5043540          252.0000
#> 20: 145.00769 14196.250              1              3          0.2955198          147.6667
#> 21: 163.01816 14682.364              1              3          0.2988518          149.3333
#> 22: 169.08578  4610.031              1              3          0.2307982          115.3333
#> 23: 181.02875  3742.822              1              3          0.2401476          120.0000
```

The `replicates` function argument tells which analyses are used for blank subtraction, in the above example all the analyses with the replicate set to “solvent-pos” are used. It is also possible to specify multiple replicate names. Typically, you will choose sample analyses here that did not undergo any sample preparation, such as pure solvent blanks.

The `filter()` function is then used to remove these peaks in the peak list data.

```
# remove background peaks from MS/MS spectra
mslists <- filter(mslists, removeMZs = bgPeaks)
```

The `removeMZs` filter removes all the MS/MS peaks from the `mz` column. You can also specify a vector with m/z values to remove, for instance, if you identified background peaks by other means than the `getBGMSMSPeaks()` function.

For more details, see the reference manual (`?getBGMSMSPeaks`).

9.9 MS libraries

The `loadMSLibraries()` function is used to load MS spectral libraries, and was already briefly introduced for compound annotation. Currently, loading of MSP files and MoNA JSON files is supported, while loading of formula annotations for MS peaks is currently only supported for the latter. The underlying algorithms implement several optimizations to efficiently load large number of records. Furthermore, `loadMSLibraries()` automatically verifies record data such as formulas, adducts and masses, and automatically calculates missing or invalid data where possible.

```
mslibraryMSP <- loadMSLibrary("MoNA-export-CASMI_2016.msp", "msp")
mslibraryJSON <- loadMSLibrary("MoNA-export-CASMI_2016.json", "json")
```

Several advanced parameters are available that influence the loading of MS library data, see the reference manual (`?loadMSLibrary`) for details.

Once loaded, the usual methods are available to inspect its data:

```
show(mslibraryMSP)
```

```
#> A MSLibrary object
#> Hierarchy:
#> workflowStep
#> |-- MSLibrary
#> ---
#> Object size (indication): 101.6 kB
#> Algorithm: msp
#> Total records: 26
#> Total peaks: 318
#> Total annotated peaks: 0 (0.00%)
```

```
mslibraryMSP[[1]] # MS/MS spectrum for first candidate
```

```
#>           mz  intensity
#>      <num>    <num>
#> 1: 135.0441  1.001001
#> 2: 161.0594  0.500501
#> 3: 163.0379  0.600601
#> 4: 173.0590  0.200200
#> 5: 176.0699  0.200200
#> ---
#> 44: 353.1191  1.201201
#> 45: 354.1323 100.000000
#> 46: 355.1351  20.820821
#> 47: 356.1374  2.702703
#> 48: 357.1401  0.300300
```

```
mslibraryJSON[["SM801601"]] # a record with annotations
```

```
#>           mz  intensity annotation
#>      <num>    <num>    <char>
#> 1:  65.0388  0.100228      C5H5
#> 2:  91.0541  0.922448      C7H7
#> 3:  93.0573  5.489900      C6H7N
#> 4: 106.0651  0.101855      C7H8N
#> 5: 108.0807 100.000000      C7H10N
#> 6: 109.0648  2.004170      C7H9O
#> 7: 132.0807  0.926004      C9H10N
#> 8: 150.0913 76.554515      C9H12NO
```

```
# overview of all metadata (select few columns for readability)
records(mslibraryJSON)[, .(DB_ID, Name, InChIKey, formula)]
```

#>	DB_ID	Name	InChIKey	formula
#>	<char>	<char>	<char>	<char>
#>	1: SM800003	1,2,3-Triazole	QWENRTYMTSOGBR-UHFFFAOYSA-N	C2H3N3
#>	2: SM800201	1-Naphthylamine	RUFPHBVGCFYCNW-UHFFFAOYSA-N	C10H9N
#>	3: SM800553	2,3-Dihydroxybiphenyl	YKOQAAJBYBTSBS-UHFFFAOYSA-N	C12H10O2

```
#> 4: SM800653 2,4-Dibromophenol FAXWFCTVSHEODL-UHFFFAOYSA-N C6H4Br2O
#> 5: SM800802 2-Aminoanthracene YCSBALJAGZKWFF-UHFFFAOYSA-N C14H11N
#> ---
#> 618: SM884401 Anthranilic acid RWZYAGGXGHYGB-UHFFFAOYSA-N C7H7NO2
#> 619: SM884552 Fipronil sulfide FQXWEKADCSXYOC-UHFFFAOYSA-N C12H4Cl2F6N4S
#> 620: SM884652 Fipronil sulfone LGHZJDKSVUTELU-UHFFFAOYSA-N C12H4Cl2F6N4O2S
#> 621: SM884701 N-Cyclohexyl-2-benzothiazol-amine UPWPIFMHSFSVLE-UHFFFAOYSA-N C13H16N2S
#> 622: SM884952 Fipronil desulfinyl JWKXVHLIRTVXLD-UHFFFAOYSA-N C12H4Cl2F6N4
```

```
# convert all data to a data.table (may be huge!)
as.data.table(mslibraryMSP)[, .(DB_ID, SMILES, formula, mz, intensity)]
```

```
#> Key: <DB_ID>
#>      DB_ID      SMILES      formula      mz inter
#>      <char>      <char>      <char>      <num>
#> 1: SMI00001 CN1CC2=C(C=CC3=C2OC03)[C@@H]4[C@H]1C5=CC6=C(C=C5C[C@@H]4O)OC06 C20H19NO5 135.0441 1.0
#> 2: SMI00001 CN1CC2=C(C=CC3=C2OC03)[C@@H]4[C@H]1C5=CC6=C(C=C5C[C@@H]4O)OC06 C20H19NO5 161.0594 0.5
#> 3: SMI00001 CN1CC2=C(C=CC3=C2OC03)[C@@H]4[C@H]1C5=CC6=C(C=C5C[C@@H]4O)OC06 C20H19NO5 163.0379 0.6
#> 4: SMI00001 CN1CC2=C(C=CC3=C2OC03)[C@@H]4[C@H]1C5=CC6=C(C=C5C[C@@H]4O)OC06 C20H19NO5 173.0590 0.2
#> 5: SMI00001 CN1CC2=C(C=CC3=C2OC03)[C@@H]4[C@H]1C5=CC6=C(C=C5C[C@@H]4O)OC06 C20H19NO5 176.0699 0.2
#> ---
#> 314: SMI00172 C1=CC=C(C=C1)NN=CC2=CC=CC=C2N C13H13N3 120.0678 22.1
#> 315: SMI00172 C1=CC=C(C=C1)NN=CC2=CC=CC=C2N C13H13N3 121.0756 6.5
#> 316: SMI00172 C1=CC=C(C=C1)NN=CC2=CC=CC=C2N C13H13N3 167.0729 28.0
#> 317: SMI00172 C1=CC=C(C=C1)NN=CC2=CC=CC=C2N C13H13N3 168.0810 13.5
#> 318: SMI00172 C1=CC=C(C=C1)NN=CC2=CC=CC=C2N C13H13N3 195.0917 8.2
```

Furthermore, like many other objects in `patRoan`, the MS library objects can be subset and filtered:

```
mslibrarySub <- mslibrary[1:100] # only keep first 100 records

# only keep records a neutral mass of 100-200
mslibraryF <- filter(mslibrary, massRange = c(100, 200))
# remove records with neutral mass below 100
mslibraryF <- filter(mslibrary, massRange = c(0, 100), negate = TRUE)
# only keep mass peaks with m/z 100-500
mslibraryF <- filter(mslibrary, mzRangeSpec = c(100, 500))
# remove low intensity peaks (<1%) and only keep top 10
mslibraryF <- filter(mslibrary, relMinIntensity = 0.01, topMost = 10)
# only keep mass peak with annotations
mslibraryF <- filter(mslibraryJSON, onlyAnnotated = TRUE)
```

In addition, the `properties` filter may be useful to tailor the library data. The library properties can be obtained as following:

```
names(records(mslibrary)) # get all property names
```

```
#> [1] "Name"          "Synon"          "DB_ID"          "InChIKey"
#> [5] "InChI"         "Precursor_type" "Spectrum_type"  "PrecursorMZ"
#> [9] "Instrument_type" "Instrument"      "Ion_mode"       "Collision_energy"
#> [13] "formula"       "MW"             "neutralMass"    "Comments"
#> [17] "SMILES"        "SPLASH"         "CAS"            "PubChemCID"
#> [21] "ChemSpiderID"  "Ionization"     "Resolution"
```



```
unique(records(mslibrary)[["Instrument_type"]]) # Get the available instrument types
```

```
#> [1] "LC-ESI-QTOF" "LC-APCI-ITFT" "APCI-ITFT"
```

Then to filter the MS library:

```
# only keep APCI instrument types
mslibraryF <- filter(mslibrary, properties = list(Instrument_type = c("LC-APCI-ITFT",
  ↪ "APCI-ITFT")))
# remove Q-TOF by negation
mslibraryF <- filter(mslibrary, properties = list(Instrument_type = "LC-ESI-QTOF"),
  ↪ negate = TRUE)
```

More advanced filtering can be performed with the `delete()` generic function, see the reference manual for details (`?MSLibrary`).

Finally, functionality exists to convert, export and merge MS libraries:

```
# Convert the MS library to a suspect list.
# By setting collapse to TRUE, all records with the same first block InChIKey
# are collapsed and mass peaks are averaged.
suspl <- convertToSuspects(mslibrary, adduct = "[M+H]+", collapse = TRUE)
# Amend custom suspect list with library data (fragments_mz column)
suspl <- convertToSuspects(mslibrary, adduct = "[M+H]+", suspects =
  ↪ patRoomData::suspectsPos)

export(mslibrary, out = "myMSLib.msp") # export to a new MSP library

mslibraryM <- merge(mslibraryMSP, mslibraryJSON) # merge two libraries
```

9.10 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 `rdck`.

```
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 M192_R355_191
plotStructure(compsClust, groupName = "M192_R355_191", 1)

# plot dendrogram
plot(compsClust, groupName = "M192_R355_191")

# re-assign clusters for a feature group
compsClust <- treeCut(compsClust, k = 5, groupName = "M192_R355_191")
# ditto, but automatic cluster determination
compsClust <- treeCutDynamic(compsClust, minModuleSize = 3, groupName = "M192_R355_191")
```

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

9.11 Feature regression analysis

Basic support in `patRoan` is available to perform simple linear regression (using $y=ax+b$ with a the slope b the intercept) on feature intensities vs given x values. The x values typically are experimental conditions such as sampling time or initial concentration of a parent in a degradation experiment. The linearity could then be used as a way to prioritize features (as performed in Helmus et al. (2025)). Originally, this functionality was implemented as a very basic method to perform rough calculations of concentrations. However, the next section describes a much better way by using the `MS2Quant` package.

The x -values should be specified as metadata in the analysis information, for instance:

```
# create analysis information: for demonstrative purposes we just base it on the example
↪ data
anaInfoRegr <- anaInfo
anaInfoRegr$experiment <- c("UV", "UV", "UV", "H2O2", "H2O2", "H2O2")
anaInfoRegr$exposure <- c(0, 30, 60, 0, 30, 60) # time in minutes
anaInfoRegr[, c("analysis", "experiment", "exposure")]
```

```
#>      analysis experiment exposure
#> 1 solvent-pos-1      UV         0
#> 2 solvent-pos-2      UV        30
#> 3 solvent-pos-3      UV        60
#> 4 standard-pos-1    H2O2         0
#> 5 standard-pos-2    H2O2        30
#> 6 standard-pos-3    H2O2        60
```

The x -values can be set to `NA` in case no experimental conditions are available; the regression properties will be calculated from all non-`NA` data.

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

```
# use areas for regression calculation and make sure that feature data is reported
# the exposure metadata is used as x values, the experiment metadata is used to calculate
# the regression for analyses groups
# (only relevant columns are shown for clarity)
as.data.table(fGroupsRegr, areas = TRUE, features = TRUE,
              regression = "exposure", regressionBy = "experiment")[, .(analysis, group,
                               x_reg, RSQ, intercept, slope, p)]
```

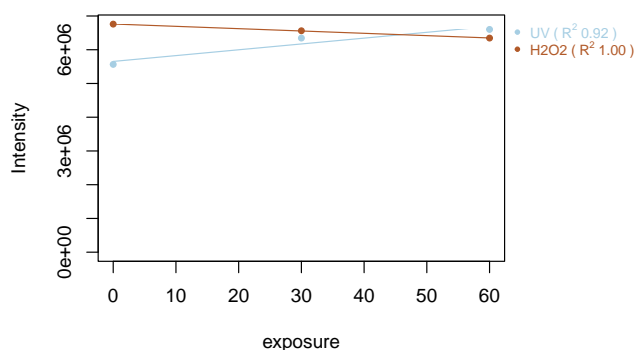
```
#>      analysis      group      x_reg      RSQ intercept      slope      p
#>      <char>      <char>      <num>      <num>      <num>      <num>      <num>
#> 1: solvent-pos-1 M99_R14_1 -765.0052 0.99525396 4351423 5176.367 0.04389244
#> 2: solvent-pos-2 M99_R14_1 -775.8074 0.99525396 4351423 5176.367 0.04389244
#> 3: solvent-pos-3 M99_R14_1 -778.7245 0.99525396 4351423 5176.367 0.04389244
#> 4: standard-pos-1 M99_R14_1 710.7070 0.90164426 4814974 -6310.333 0.20308172
#> 5: standard-pos-2 M99_R14_1 716.6610 0.90164426 4814974 -6310.333 0.20308172
#> ---
#> 2922: solvent-pos-2 M433_R10_680 -538.6387 0.99048530 2595803 3329.317 0.06219692
#> 2923: solvent-pos-3 M433_R10_680 -534.8049 0.99048530 2595803 3329.317 0.06219692
#> 2924: standard-pos-1 M433_R10_680 1635.3435 0.06340353 3078297 -1303.883 0.83795459
#> 2925: standard-pos-2 M433_R10_680 1716.0164 0.06340353 3078297 -1303.883 0.83795459
#> 2926: standard-pos-3 M433_R10_680 1660.7139 0.06340353 3078297 -1303.883 0.83795459
```

The `regressionBy` argument in the above example is set to ensure that regression is calculated for each experiment separately. In sets workflows you can set `regressionBy="set"` to perform the calculations per set. The `regressionBy` argument can be omitted if the regression should be calculated for all analyses together. The `x_reg` output column stores the x values calculated by the regression model (only present if `features=TRUE`). Other regression properties such as the RSQ, slope and p can be used as a basic trend analysis for prioritization:

```
fGroupsTab <- as.data.table(fGroupsRegr, areas = TRUE, features = TRUE,
                           regression = "exposure", regressionBy = "experiment")
# subset features that appear to increase in intensity with longer UV exposure
increasingFGroups <- fGroupsRegr[, fGroupsTab[RSQ >= 0.8 & p < 0.05 & slope > 0, group]]
```

The `plotInt()` function can also work with regression data and is helpful for visualization purposes.

```
# plot regression for a specific feature group
plotInt(fGroupsRegr[, 13], areas = TRUE, regression = TRUE, xBy = "exposure",
        groupBy = "experiment", showLegend = TRUE)
```



For more details, see the reference manual for the feature groups methods for `as.data.table()` and `plotInt()` (`?as.data.table` and `?plotInt`).

9.12 Predicting toxicities and concentrations (MS2Tox and MS2Quant integration)

The MS2Tox and MS2Quant R packages predict toxicities and feature concentrations using a machine learning approach. The predictions are performed with either SMILES data or fingerprints calculated from MS/MS data with SIRIUS+CSI:FingerID. While using SMILES data is generally more accurate, using MS/MS fingerprints is generally faster and may be more suitable for features without known or suspected structure.

In `patRoan` the predictions are done in two steps:

1. The LC50 values (toxicity prediction) or response factors (concentration prediction) are calculated for given SMILES or MS/MS fingerprint data using MS2Tox/MS2Quant. This step is performed by the `predictTox()/predictConcs()` method function.
2. The predicted LC50 values are used to assign toxicities/concentrations to feature data. This is performed by the `calculateTox()/calculateConcs()` method function.

Various workflow data can be used to perform the predictions for step 1:

- a. Suspect hits that were obtained with `screenSuspects` (see suspect screening).
- b. Formula annotations obtained with SIRIUS+CSI:FingerID.
- c. Compound annotations obtained with SIRIUS+CSI:FingerID.
- d. Compound annotations obtained with other algorithms, e.g. `MetFrag`.

For *a* and *d* SMILES is used to perform the calculations, for *b* MS/MS fingerprints are used and for *c* either can be used.

NOTE For option *b*, make sure that `getFingerprints=TRUE` and SIRIUS logins are handled when running `generateFormulas()` in order to obtain fingerprints.

9.12.1 Predicting toxicities

Some example workflows are shown below:

```
# Calculate toxicity for suspect hits.
fGroupsSuspTox <- predictTox(fGroupsSusp)
fGroupsSuspTox <- calculateTox(fGroupsSuspTox)

# Calculate toxicity for compound hits. Limit to the top 5 to reduce calculation time.
compoundsTop5 <- filter(compounds, topMost = 5)
compoundsTox <- predictTox(compoundsTop5)
fGroupsTox <- calculateTox(fGroups, compoundsTox)
```

It is also possible to calculate toxicities from multiple workflow objects:

```
fGroupsSuspTox <- predictTox(fGroupsSusp) # as above

# Predict toxicities from compound candidates, using both SMILES and MS/MS fingerprints
# compoundsSuspSIR is an object produced with generateCompounds() with algorithm="sirius"
compoundsSuspSIRTox <- predictTox(compoundsSuspSIR, type = "both")

# Assign toxicities to feature groups from both suspect hits and SIRIUS annotations
fGroupsSuspTox <- calculateTox(fGroupsSuspTox, compoundsSuspSIRTox)
```

More details are in the reference manual: `?`pred-tox``.

9.12.2 Predicting concentrations

The workflow to predict concentrations is quite similar to predicting toxicities. However, before we can start we first have to specify the *calibrants* and the LC gradient elution program.

The calibrant data is used by `MS2Quant` to convert predicted ionization efficiencies to actual response factors, which are specific to the used LC instrument and methodology. For this purpose, several mixtures with known concentrations (i.e. standards) should be measured alongside your samples. The calibrants should be specified as a `data.frame`, for instance:

name	SMILES	intensity	conc	rt
Atrazine	<chem>CCNc1nc(nc(n1)Cl)NC(C)C</chem>	32708	1	336.6
Atrazine	<chem>CCNc1nc(nc(n1)Cl)NC(C)C</chem>	66880	2	336.6
Atrazine	<chem>CCNc1nc(nc(n1)Cl)NC(C)C</chem>	174087	5	336.6
Atrazine	<chem>CCNc1nc(nc(n1)Cl)NC(C)C</chem>	371192	10	336.6
Atrazine	<chem>CCNc1nc(nc(n1)Cl)NC(C)C</chem>	806749	25	336.6
Atrazine	<chem>CCNc1nc(nc(n1)Cl)NC(C)C</chem>	1852591	50	336.6
Carbamazepine	<chem>c1ccc2c(c1)C=Cc3ccccc3N2C(=N)O</chem>	25231	1	349.2
Carbamazepine	<chem>c1ccc2c(c1)C=Cc3ccccc3N2C(=N)O</chem>	47831	2	349.2
Carbamazepine	<chem>c1ccc2c(c1)C=Cc3ccccc3N2C(=N)O</chem>	118843	5	349.2
Carbamazepine	<chem>c1ccc2c(c1)C=Cc3ccccc3N2C(=N)O</chem>	211395	10	349.2
Carbamazepine	<chem>c1ccc2c(c1)C=Cc3ccccc3N2C(=N)O</chem>	545192	25	349.2
Carbamazepine	<chem>c1ccc2c(c1)C=Cc3ccccc3N2C(=N)O</chem>	1083568	50	349.2
DEET	<chem>CCN(CC)C(=O)c1cccc(c1)C</chem>	45061	1	355.8
DEET	<chem>CCN(CC)C(=O)c1cccc(c1)C</chem>	84859	2	355.8
DEET	<chem>CCN(CC)C(=O)c1cccc(c1)C</chem>	228902	5	355.8
DEET	<chem>CCN(CC)C(=O)c1cccc(c1)C</chem>	434161	10	355.8
DEET	<chem>CCN(CC)C(=O)c1cccc(c1)C</chem>	1133166	25	355.8
DEET	<chem>CCN(CC)C(=O)c1cccc(c1)C</chem>	2385472	50	355.8
Venlafaxine	<chem>CN(C)CC(C1=CC=C(C=C1)OC)C2(CCCCC2)O</chem>	41465	1	324.0
Venlafaxine	<chem>CN(C)CC(C1=CC=C(C=C1)OC)C2(CCCCC2)O</chem>	89684	2	324.0
Venlafaxine	<chem>CN(C)CC(C1=CC=C(C=C1)OC)C2(CCCCC2)O</chem>	230890	5	324.0
Venlafaxine	<chem>CN(C)CC(C1=CC=C(C=C1)OC)C2(CCCCC2)O</chem>	400385	10	324.0
Venlafaxine	<chem>CN(C)CC(C1=CC=C(C=C1)OC)C2(CCCCC2)O</chem>	1094329	25	324.0
Venlafaxine	<chem>CN(C)CC(C1=CC=C(C=C1)OC)C2(CCCCC2)O</chem>	1965139	50	324.0

The intensity column should contain either the peak intensity (height) or area. Note that some feature detection algorithms can sometimes produce inaccurate peak areas, and the area determination methodology is often different among algorithms. For this reason, using peak intensities may be more reliable, however, it is worth testing this with your data.

It is also possible to use the `getQuantCalibFromScreening()` function to automatically create the calibrant table from feature group data:

```
calibList <- data.frame(...) # this should be a suspect list with your calibrants
fGroups <- screenSuspects(fGroups, calibList) # screen for the calibrants
concs <- data.frame(...) # concentration data for each calibrant compound, see below
calibrants <- getQuantCalibFromScreening(fGroups, concs)
calibrants <- getQuantCalibFromScreening(fGroups, concs, areas = TRUE) # obtain feature
↳ areas instead of intensities
```

The first step is to perform a screening for the calibrant compounds. Please ensure that this list contains SMILES data, and to ensure correct feature assignment it is highly recommended to include retention times. The second requirement for `getQuantCalibFromScreening()` is a table with concentrations for each calibrant compound, e.g.:

```

concs <- data.frame(
  name = c("DEET", "1h-benzotriazole", "Caffeine", "Atrazine", "Carbamazepine",
    ↪ "Venlafaxine"),
  standard_1 = c(1.00, 1.05, 1.10, 0.99, 1.01, 1.12),
  standard_2 = c(2.00, 2.15, 2.20, 1.98, 2.02, 1.82),
  standard_5 = c(5.01, 5.05, 5.22, 5.00, 4.88, 4.65),
  standard_10 = c(10.2, 10.11, 10.23, 11.77, 11.75, 12.13),
  standard_25 = c(25.3, 25.12, 25.34, 24.89, 24.78, 24.68),
  standard_50 = c(50.34, 50.05, 50.10, 49.97, 49.71, 50.52)
)
concs

```

```

#>           name standard_1 standard_2 standard_5 standard_10 standard_25 standard_50
#> 1          DEET         1.00         2.00         5.01         10.20         25.30         50.34
#> 2 1h-benzotriazole         1.05         2.15         5.05         10.11         25.12         50.05
#> 3          Caffeine         1.10         2.20         5.22         10.23         25.34         50.10
#> 4          Atrazine         0.99         1.98         5.00         11.77         24.89         49.97
#> 5   Carbamazepine         1.01         2.02         4.88         11.75         24.78         49.71
#> 6     Venlafaxine         1.12         1.82         4.65         12.13         24.68         50.52

```

The concentrations are specified separately for each calibrant compound. The column names should follow the names of the replicate assigned to the standards. The concentration unit is $\mu\text{g/l}$ by default. The next section describes how this can be changed.

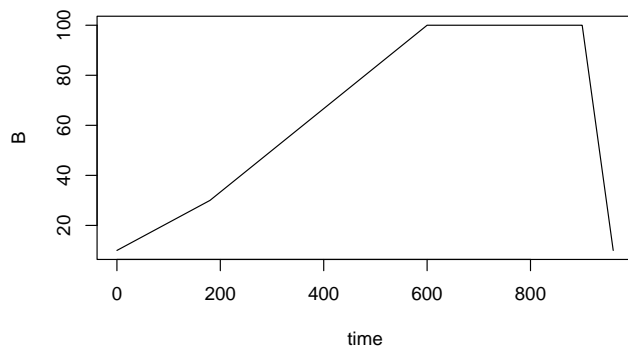
NOTE The `getQuantCalibFromScreening()` function ignores any IMS features in IMS workflows with post mobility assignment, as the calibration procedure does not take any differences due to IMS separation into account.

The gradient elution program is also specified by a `data.frame`, which for every time point (in seconds!) describes the percentage of 'B'. In this case, 'B' represents the total amount of organic modifier.

```

eluent <- data.frame(
  time = c(0, 180, 600, 900, 960),
  B = c(10, 30, 100, 100, 10)
)
plot(eluent, type = "l")

```



```
eluent
```

```
#>   time   B
#> 1     0  10
#> 2    180  30
#> 3    600 100
#> 4    900 100
#> 5    960  10
```

Then, the workflow to predict concentrations is very similar then predicting toxicities (previous section):

```
# Calculate concentrations for suspect hits.
fGroupsSuspConc <- predictRespFactors(
  fGroupsSusp,
  calibrants = calibrants, eluent = eluent,
  organicModifier = "MeCN", # organic modifier: MeOH or MeCN
  pHaq = 4 # pH of the aqueous part of the mobile phase
)
# set areas to TRUE if the calibrant table contains areas
fGroupsSuspConc <- calculateConcs(fGroupsSuspConc, areas = FALSE)
```

As was shown for toxicities it is possible to use different data sources (e.g. compound annotations, suspects) for predictions.

More details are in the reference manual: `?`pred-quant``.

9.12.3 Toxicity and concentration units

The default unit for toxicity and concentration data is $\mu\text{g/l}$. However, this can be configured when calling the `predictTox()/predictRespFactors()` functions:

```
fGroupsSuspTox <- predictTox(fGroupsSusp) # default unit: ug/l
fGroupsSuspTox <- predictTox(fGroupsSusp, concUnit = "ug/l") # same as above
fGroupsSuspTox <- predictTox(fGroupsSusp, concUnit = "mM") # millimolar
fGroupsSuspTox <- predictTox(fGroupsSusp, concUnit = "log mM") # unit used by MS2Tox

# calculated concentrations are ng/l, calibrants are specified in ug/l
# (by default calibConcUnit=concUnit)
fGroupsSuspConc <- predictRespFactors(
  fGroupsSusp, calibrants = calibrants, eluent = eluent,
  organicModifier = "MeCN", pHaq = 4,
  concUnit = "ng/l", calibConcUnit = "ug/l"
)
```

See the reference manuals (`?`pred-tox`/`?`pred-quant``) For more details on which units can be specified

9.12.4 Inspecting predicted values

The raw toxicity and concentration data assigned to feature groups can be retrieved with the `toxicities()` and `concentrations()` method functions, respectively.

```
toxicities(fGroupsSuspTox)
```

```
#>      group    type      candidate      candidate_name      LC50
#>      <char> <char>      <char>      <char>      <num>
#> 1: M120_R268_30 suspect [nH]1nnc2ccccc12 1H-benzotriazole 24327.64
#> 2: M137_R249_53 suspect NC(=O)Nc1ccccc1 N-Phenyl urea 68490.93
#> 3: M146_R225_70 suspect Oc1ccc2ccccc2n1 2-Hydroxyquinoline 11815.10
#> 4: M146_R248_69 suspect Oc1ccc2ccccc2n1 2-Hydroxyquinoline 11815.10
#> 5: M146_R309_68 suspect Oc1ccc2ccccc2n1 2-Hydroxyquinoline 11815.10
```

```
concentrations(fGroupsSuspConc)
```

```
#>      group    type      candidate      candidate_name standard-pos-1 standard-pos-2 standard-p
#>      <char> <char>      <char>      <char>      <num>      <num>      <num>
#> 1: M120_R268_30 suspect [nH]1nnc2ccccc12 1H-benzotriazole 43.070773 39.905306 35.2
#> 2: M137_R249_53 suspect NC(=O)Nc1ccccc1 N-Phenyl urea 18.485430 19.864756 17.3
#> 3: M146_R225_70 suspect Oc1ccc2ccccc2n1 2-Hydroxyquinoline 15.700200 17.662215 18.4
#> 4: M146_R248_69 suspect Oc1ccc2ccccc2n1 2-Hydroxyquinoline 19.030263 20.207821 19.5
#> 5: M146_R309_68 suspect Oc1ccc2ccccc2n1 2-Hydroxyquinoline 7.978394 8.646156 8.6
```

If there were multiple candidates for a single feature group then these are split over the table rows:

```
toxicities(fGroupsTox)
```

```
#>      group    type      candidate      candidate
#>      <char> <char>      <char>      <char>
#> 1: M120_R268_30 compound C1=CC2=C(C=NN2)N=C1 1H-pyrazolo[4,5-b]pyr
#> 2: M120_R268_30 compound C1=CC2=C(N=C1)N=CN2 1H-imidazo[4,5-b]pyr
#> 3: M120_R268_30 compound C1=CC2=NNN=C2C=C1 2H-benzotri
#> 4: M120_R268_30 compound C1=CN2C(=CC=N2)N=C1 pyrazolo[1,5-a]pyrim
#> 5: M120_R268_30 compound C1=CNC2=CN=CN=C21 5H-pyrrolo[3,2-d]pyrim
#> ---
#> 16: M192_R355_191 compound CCN(CC)C(=O)C1=CC=C(C=C1)C N,N-diethyl-4-methylbenza
#> 17: M192_R355_191 compound CCN(CC)C(=O)C1=CC=CC(=C1)C N,N-diethyl-3-methylbenza
#> 18: M192_R355_191 compound CCN(CC)C(=O)CC1=CC=CC=C1 N,N-diethyl-2-phenylaceta
#> 19: M192_R355_191 compound CCNC(C)C(=O)C1=CC=C(C=C1)C 2-(ethylamino)-1-(4-methylphenyl)propan-
#> 20: M192_R355_191 compound C[C@H]1[C@@H](OCCN1C)C2=CC=CC=C2 (2S,3S)-3,4-dimethyl-2-phenylmorph
```

The `as.data.table()` method function, which was discussed previously, can be used to summarize toxicity and concentration values.

```
# NOTE: NA values are filtered and columns are subset for readability
as.data.table(fGroupsTox)[!is.na(LC50), c("group", "LC50", "LC50_types")]
```

```
#>      group      LC50 LC50_types
#>      <char>      <num>      <char>
#> 1: M120_R268_30 69343.39 compound
#> 2: M137_R249_53 265983.07 compound
#> 3: M146_R309_68 7442.69 compound
#> 4: M192_R355_191 44602.85 compound
```



```
concCols <- c("group", paste0(analyses(fGroupsSuspConc), "_conc"), "conc_types")
as.data.table(fGroupsSuspConc)[!is.na(conc_types), concCols, with = FALSE]
```

```
#>           group standard-pos-1_conc standard-pos-2_conc standard-pos-3_conc conc_types
#>           <char>           <num>           <num>           <num>           <char>
#> 1: M120_R268_30          43.070773          39.905306          35.21956        suspect
#> 2: M137_R249_53          18.485430          19.864756          17.36268        suspect
#> 3: M146_R309_68           7.978394           8.646156           8.67157        suspect
#> 4: M146_R248_69          19.030263          20.207821          19.54181        suspect
#> 5: M146_R225_70          15.700200          17.662215          18.49330        suspect
```

The `as.data.table()` method function *aggregates* the data for a feature group in case multiple candidates were assigned to it. By default the values are mean averaged, but this can be changed with the `toxAggrParams/concAggrParams` arguments, for instance:

```
# as above, but aggregate by taking maximum values
as.data.table(fGroupsTox, toxAggrParams = getDefPredAggrParams(max)) [!is.na(LC50),
  ↪ c("group", "LC50", "LC50_types")]
```

```
#>           group      LC50 LC50_types
#>           <char>      <num>      <char>
#> 1: M120_R268_30 135239.04    compound
#> 2: M137_R249_53 937420.55    compound
#> 3: M146_R309_68 11936.93     compound
#> 4: M192_R355_191 81974.51    compound
```

If the `as.data.table()` method is used on suspect screening results, and predictions were performed directly for suspect hits, then predicted values can be reported for individual suspect match instead of aggregating them per feature group:

```
# Reports predicted values for each suspect separately. If multiple suspects are assigned
  ↪ to a feature group,
# then each suspect match is split into a different row.
as.data.table(fGroupsSuspTox, collapseSuspects = NULL)
```

Finally, the reporting functionality can be used to overview all predicted values, both aggregated and raw.

9.12.5 Using predicted values to prioritize data

The `filter()` method function that was introduced before can also be used to filter data based on predicted toxicities, response factors and concentrations. For instance, this allows you to remove annotation candidates which are unlikely to be toxic or sensitive enough to be detected or any features with very low concentrations. Some examples are shown below.

```
# compoundsSuspSIRTox is an object with predicted toxicities (LC50 values) for each
  ↪ candidate
# we can use the common scoreLimits filter to select a range of allowed values (min/max)
compoundsSuspSIRToxF <- filter(compoundsSuspSIRTox, scoreLimits = list(LC50_SMILES = c(0,
  ↪ 1E4)))
```

```

# for suspects with predicted toxicities/response factors there are dedicated filters
fGroupsSuspConcF <- filter(fGroupsSuspConc, minRF = 5E4) # remove suspect hits with
  ↳ response factor <5E4
fGroupsSuspToxF <- filter(ffGroupsSuspTox, maxLC50 = 100) # remove suspect hits with LC50
  ↳ values > 100

# similarly, for feature data there are dedicated filters.
# note that these aggregate data prior to filtering (see previous section)
fGroupsConcF <- filter(fGroupsConc, absMinConc = 0.02)
# only keep features with concentrations that are at least 1% of their toxicity
# note that both concentrations/toxicity values should have been calculated with
  ↳ calculateConcs()/calculateTox()
fGroupsConcToxF <- filter(fGroupsConcTox, absMinConcTox = 0.01)

# also get rid of features without concentrations (these are ignored by default)
fGroupsConcF <- filter(fGroupsConc, absMinConc = 0.02, removeNA = TRUE)
# like as.data.table we can configure how values are aggregated
# here the minimum is used instead of the default mean
fGroupsToxF <- filter(fGroupsTox, absMaxTox = 5E3, predAggrParams =
  ↳ getDefPredAggrParams(min))

```

More details are found in the reference manual (`?feature-filtering`).

9.13 Fold changes

A specific statistical way to prioritize feature data is by Fold changes (FC). This is a relative simple method to quickly identify (significant) changes between two sample groups. A typical use case is to compare the feature intensities before and after an experiment.

To perform FC calculations we first need to specify its parameters. This is best achieved with the `getFCParams()` function:

```

getFCParams(c("before", "after"))

#> $replicates
#> [1] "before" "after"
#>
#> $thresholdFC
#> [1] 0.25
#>
#> $thresholdPV
#> [1] 0.05
#>
#> $zeroMethod
#> [1] "add"
#>
#> $zeroValue
#> [1] 0.01
#>
#> $PVTestFunc
#> function (x, y)
#> t.test(x, y, paired = TRUE)$p.value

```

```
#> <bytecode: 0x560d6f3660a0>
#> <environment: 0x560d327de320>
#>
#> $PVAdjFunc
#> function (pv)
#> p.adjust(pv, "BH")
#> <bytecode: 0x560d6f3663b0>
#> <environment: 0x560d327de320>
```

In this example we generate a list with parameters in order to make a comparison between two replicates: before and after. Several advanced parameters are available to tweak the calculation process. These are explained in the reference manual (`?featureGroups`).

The `as.data.table` function for feature groups is used to perform the FC calculations.

```
myFCParams <- getFCParams(c("solvent-pos", "standard-pos")) # compare solvent/standard
as.data.table(fGroups, FCParams = myFCParams)[, c("group", "FC", "FC_log", "PV",
  ↪ "PV_log", "classification")]
```

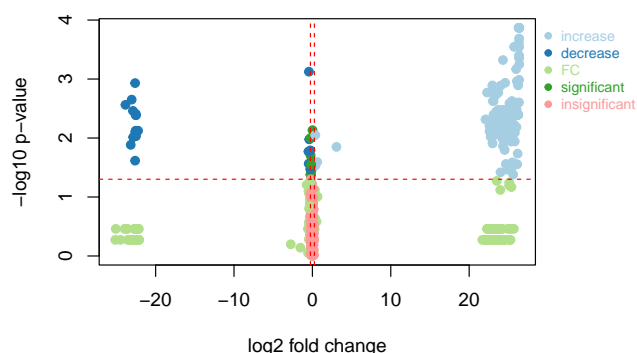
```
#>           group      FC      FC_log      PV      PV_log classification
#>      <char>      <num>      <num>      <num>      <num>      <char>
#> 1:  M99_R14_1 8.837494e-01 -0.17829070 0.223506802 0.65070926 insignificant
#> 2:  M99_R4_2 8.500464e-01 -0.23438649 0.778488444 0.10874783 insignificant
#> 3:  M100_R7_3 8.009186e-01 -0.32027248 0.804751489 0.09433821          FC
#> 4:  M100_R5_4 4.140000e+06 21.98119934 0.533213018 0.27309926          FC
#> 5:  M100_R28_5 9.594972e-01 -0.05964952 0.975712373 0.01067819 insignificant
#> ---
#> 676: M425_R319_676 2.149907e+07 24.35777069 0.009681742 2.01404652          increase
#> 677: M427_R10_677 1.059937e+00 0.08397893 0.371260940 0.43032074 insignificant
#> 678: M427_R319_678 7.776800e+06 22.89074521 0.533213018 0.27309926          FC
#> 679: M432_R383_679 9.816400e+06 23.22676261 0.347009089 0.45965915          FC
#> 680: M433_R10_680 1.132909e+00 0.18003240 0.293217996 0.53280938 insignificant
```

The classification column allows you to easily identify if and how a feature changes between the two sample groups. This can also be used to prioritize feature groups:

```
tab <- as.data.table(fGroups, FCParams = myFCParams)
# only keep feature groups that significantly increase or decrease
fGroupsChanged <- fGroups[, tab[classification %in% c("increase", "decrease")]]$group]
```

The `plotVolcano` function can be used to visually the FC data:

```
plotVolcano(fGroups, myFCParams)
```



9.14 Caching

In **patRoön** 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(patRoön.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 **patRoön** 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:
```

```
#> - EICAllTimes (12 rows)
#> - EICData (3 rows)
#> - EICs (3749 rows)
#> - EIMs (413 rows)
#> - LC50_SMILES (23 rows)
#> - MS2QMD (1 rows)
#> - MSLibraryJSON (1 rows)
#> - MSLibraryMSP (1 rows)
#> - MSPeakLists (234 rows)
#> - MSPeakListsAvg (5 rows)
#> - MSPeakListsSetAvg (2 rows)
#> - MSReadBackendMetadata (21 rows)
#> - MSReadBackendMobilities (3 rows)
#> - MSReadBackendMzR (18 rows)
#> - RF_SMILES (5 rows)
#> - TPsLib (1 rows)
#> - TPsLogP (1 rows)
#> - assignFeatureMobilitiesPeaks (1 rows)
#> - assignMobilitiesDT (2 rows)
#> - avgBGMSMS (3 rows)
#> - calculatePeakQualities (3 rows)
#> - componentsCAMERA (1 rows)
#> - componentsNontarget (1 rows)
#> - componentsTPs (1 rows)
#> - compoundsCluster (1 rows)
#> - compoundsMetFrag (79 rows)
```

```

#> - estimateIDConfidenceCompounds (2 rows)
#> - estimateIDConfidenceFormulas (2 rows)
#> - estimateIDConfidenceScr (1 rows)
#> - featureGroupsOpenMS (6 rows)
#> - featuresOpenMS (75 rows)
#> - filterFGroups_IMS_selection (1 rows)
#> - filterFGroups_blank (4 rows)
#> - filterFGroups_intensity (11 rows)
#> - filterFGroups_minAnalyses (1 rows)
#> - filterFGroups_minReplicates (83 rows)
#> - filterFGroups_minSets (1 rows)
#> - filterFGroups_replicateAbundance (8 rows)
#> - filterFGroups_replicate_group (6 rows)
#> - filterFGroups_retention (3 rows)
#> - filterMSPeakLists (4 rows)
#> - formulasFGroupConsensus (1 rows)
#> - formulasGenForm (39 rows)
#> - formulasSIRIUS (5 rows)
#> - generateTPsBT (74 rows)
#> - loadIntensities (75 rows)
#> - peaksEIC (3 rows)
#> - reintegrateIMSFeatures (1 rows)
#> - reportHTMLCompounds (1 rows)
#> - reportHTMLFormulas (1 rows)
#> - screenSuspects (8 rows)
#> - screenSuspectsPrepList (9 rows)
#> - updateFGroupsForMobilities (1 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

```

9.15 Parallelization

Some steps in the non-target screening workflow are inherently computationally intensive. To reduce computational times **patRoön** is able to perform *parallelization* for most of the important functionality. This is especially useful if you have a modern system with multiple CPU cores and sufficient RAM.

For various technical reasons several parallelization techniques are used, these can be categorized as *multithreading*, *parallelization of R functions* and *multiprocessing*. The next sections describe these parallelization approaches in order to let you optimize the workflow.

9.15.1 Multithreading

Multithreading is the most efficient form of parallelization, but is limited to C/C++ code. In **patRoön** this is realized through the OpenMP API. Multithreading is currently only used by the msdata interface to load raw (IMS-)HRMS data and by the **piek** algorithm for peak detection in chromatograms and mobilograms.

The maximum number of threads used for parallelization is configured through the `patRoan.threads` option. By default it is set to the number of available CPU cores, which results usually in the best performance. However, this could be lowered, for instance, to keep your computer more responsive or use less RAM while processing data.

```
options(patRoan.threads = 2) # do not use more than two threads
```

9.15.2 Parellization of R functions

Several functions of `patRoan` support parallelization.

Function	Purpose	Remarks
<code>findFeatures</code>	Obtain feature data	Only <code>envipick</code> and <code>kpic2</code> algorithms.
<code>generateComponents</code>	Generate components	Only <code>cliquems</code> algorithm.
<code>estimateIDConfidence</code>	Estimate ID confidence	Only for <code>formulas</code> and <code>compounds</code> .
<code>report</code>	Reporting data	
<code>assignMobilities</code>	Assign IMS data	Only for <code>featureGroups</code> .
<code>generateTPs</code>	Obtain transformation products	Only <code>cts</code> , <code>ann_form</code> and <code>ann_comp</code> algorithms.
<code>optimizeFeatureFinding</code> , <code>optimizeFeatureGrouping</code>	Optimize feature finding/grouping parameters	Discussed here.
<code>calculatePeakQualities</code>	Calculate feature (group) qualities	Discussed here.
<code>predictTox</code> / <code>predictRespFactors</code>	Prediction of toxicities/concentrations	Only <code>compounds</code> methods. Discussed here.

The parallelization is achieved with the `future` and `future.apply` R packages. To enable parallelization of these functions the `parallel` argument must be set to `TRUE` and the future framework must be properly configured in advance. For example:

```
# setup three workers to run in parallel
future::plan("multisession", workers = 3)

# find features with envipick in parallel
fList <- findFeatures(anaInfo, "envipick", parallel = TRUE)
```

It is important to properly configure the right future plan. With more advanced configurations it is also possible to perform the the parallelization on one or more external computers. The future R package supports many approaches, such as cluster computing via multiple networked computers and more advanced HPC approaches such as `slurm` via the `future.batchtools` R package. For example, the following configuration uses four nodes on an external computer to perform data processing operations:

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

Please see the documentation of the respective packages (e.g. `future` and `future.batchtools`) for more details on how to configure the workers.

9.15.3 Multiprocessing

`patRoön` relies on several external (command-line) tools to generate workflow data. These commands may be executed in *parallel* to reduce computational times ('multiprocessing'). The table below outlines the tools that are executed in parallel.

Tool	Used by	Notes
<code>msConvert</code>	<code>convertMSFiles(algorithm="pwiz", ...)</code>	
<code>FileConverter</code>	<code>convertMSFiles(algorithm="openms", ...)</code>	
<code>TIMSCONVERT</code>	<code>convertMSFiles(algorithm="timsconvert", ...)</code>	
<code>FeatureFinderMetabo</code>	<code>findFeatures(algorithm="openms", ...)</code>	
<code>julia</code>	<code>findFeatures(algorithm="safd", ...)</code>	
<code>SIRIUS</code>	<code>findFeatures(algorithm="sirius", ...)</code>	
<code>MetaboliteAdductDech</code>	<code>generateComponents(algorithm="openms", ...)</code>	
<code>GenForm</code>	<code>generateFormulas(algorithm="genform", ...)</code>	
<code>SIRIUS</code>	<code>generateFormulas(algorithm="sirius", ...),</code> <code>generateCompounds(algorithm="sirius", ...)</code>	Only if <code>splitBatches=TRUE</code>
<code>MetFrag</code>	<code>generateCompounds(algorithm="metfrag", ...)</code>	
<code>BioTransformer</code>	<code>generateTPs(algorithm = "biotransformer")</code>	Disabled by default (see <code>?generateTPs</code> for details).
<code>OpenBabel</code>	various	Used to convert and verify chemical data, such as formulae and SMILES

Multiprocessing is either performed by executing processes in the background with the `processx` R package (*classic interface*) or by futures, which were introduced in the previous section. An overview of the characteristics of both parallelization techniques is shown below.

<code>classic</code>	<code>future</code>
requires little or no configuration	configuration needed to setup
works with all tools	slower with <code>GenForm</code>
only supports parallelization on the local computer	allows both local and cluster computing

Which method is used is controlled by the `patRoön.MP.method` package option.

9.15.3.1 Classic multiprocessing interface The classic interface is the 'original' method implemented in `patRoön`, 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(patRoön.MP.method = "classic")
```

The number of parallel processes is configured through the `patRoön.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(patRoan.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 with the `withOpt()` function.

```
# do not execute more than two tools in parallel.
options(patRoan.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 `patRoan..`

9.15.3.2 Multiprocessing with futures The primary goal of the “future” method is to use the future R package to allow parallel processing on one or more external computers, as discussed before. For example:

```
options(patRoan.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))
```

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

```
# default to future parallelization
options(patRoan.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
```

9.15.3.3 Logging Most tools that are executed in parallel will log their output to text files. These files may contain valuable information, for instance, when 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 `patRoan.MP.logPath` option. If you set this option to `FALSE` then no logging occurs.

9.15.4 Notes when using parallelization with futures

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

- `GenForm` currently performs less optimal with future multiprocessing to 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 `patRoön.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 `patRoön` is present and with the same version on all computing hosts.
- Make sure that any external dependencies used by multiprocessing, such as `MetFrag` and `SIRIUS`, and local compound databases, such as `PubChemLite`, are also with the same version and are configured properly. See the Installation section for more details.
- If you encounter errors then it may be handy to switch to `future::plan("sequential")` and see if it works or you get more descriptive error messages.
- In order to restart the nodes, for instance after re-configuring `patRoön`, 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 to find problems.
- If external parallel workers are used (e.g. clusters) with `patRoön` functionality that produces output files, then make sure that the output directory is shared between the workers and that the workers have write access to it. This is currently mainly relevant for `convertMSFiles()` and `report()`.

10 References

- Chetnik, Kelsey, Lauren Petrick, and Gaurav Pandey. 2020. "MetaClean: A Machine Learning-Based Classifier for Reduced False Positive Peak Detection in Untargeted LC-MS Metabolomics Data." *Metabolomics* 16 (11). <https://doi.org/10.1007/s11306-020-01738-3>.
- Dietrich, Christian, Arne Wick, and Thomas A. Ternes. 2021. "Open-Source Feature Detection for Non-Target LC-MS Analytics." *Rapid Communications in Mass Spectrometry* 36 (2). <https://doi.org/10.1002/rcm.9206>.
- Helmus, Rick, Ingrida Bagdonaite, Pim de Voogt, et al. 2025. "Comprehensive Mass Spectrometry Workflows to Systematically Elucidate Transformation Processes of Organic Micropollutants: A Case Study on the Photodegradation of Four Pharmaceuticals." *Environmental Science & Technology* 59 (7): 3723–36. <https://doi.org/10.1021/acs.est.4c09121>.
- Schollee, Jennifer E., Emma L. Schymanski, Sven E. Avak, Martin Loos, and Juliane Hollender. 2015. "Prioritizing Unknown Transformation Products from Biologically-Treated Wastewater Using High-Resolution Mass Spectrometry, Multivariate Statistics, and Metabolic Logic." *Analytical Chemistry* 87 (24): 12121–29. <https://doi.org/10.1021/acs.analchem.5b02905>.
- Schymanski, Emma L., Junho Jeon, Rebekka Gulde, et al. 2014. "Identifying Small Molecules via High Resolution Mass Spectrometry: Communicating Confidence." *Environmental Science and Technology* 48 (4): 2097–98. <https://doi.org/10.1021/es5002105>.