Intro to RaMS

Welcome to RaMS! This vignette is designed to provide examples using the package at various levels of complexity. Let's jump right in.

If you have your own data, feel free to load it here. If not, there's a couple example small files you're welcome to use in the "inst" folder. I'll be using these throughout.

```
library(RaMS)
library(data.table)

# Locate the file directory
msdata_dir <- system.file("extdata", package = "RaMS")

# Identify the files of interest
data_files <- list.files(msdata_dir, pattern = "Full.*mzML", full.names = TRUE)

# Check that the files identified are the ones expected
basename(data_files)
#> [1] "FK180310_Full1.mzML.gz" "FK180310_Full2.mzML.gz" "FK180310_Full3.mzML.gz"
```

There's only one function to worry about in RaMS: the aptly named `grabMSdata`. This function has a couple arguments with sensible defaults, but you'll always need to tell it two things: one, which files you'd like to process; and two, the data you'd like to obtain from those files.

Let's start simple, with a single file and the most basic information about it.

Basic RaMS usage

TICs, BPCs, and metadata

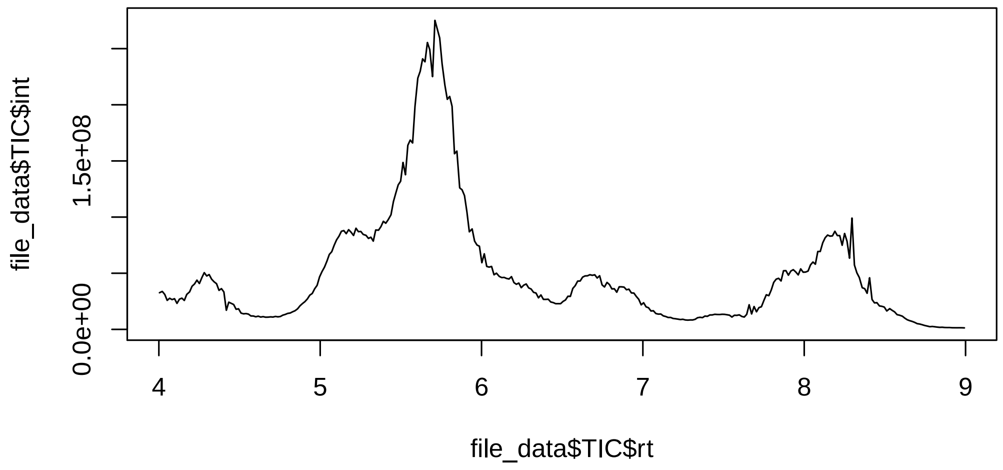
A TIC reports the total intensity measured by the mass analyzer during each scan, so the data is parsed into two columns: retention time (rt) and intensity (int). This makes it easy to read and simple to plot:

```
single_file <- data_files[1]

file_data <- grabMSdata(single_file, grab_what = "TIC")

knitr::kable(head(file_data$TIC, 3))
```

rt	int	filename
4.003672	32618050	FK180310_Full1.mzML.gz
4.021268	33817507	FK180310_Full1.mzML.gz
4.036503	31136263	FK180310_Full1.mzML.gz

Since we asked for a single thing, the TIC, our `file_data` object is a list with a single entry: the TIC. Let's plot that data:

```r
par(mar=c(4.1, 4.1, 0.1, 0.1))
plot(file_data$TIC$rt, file_data$TIC$int, type = "l")
```



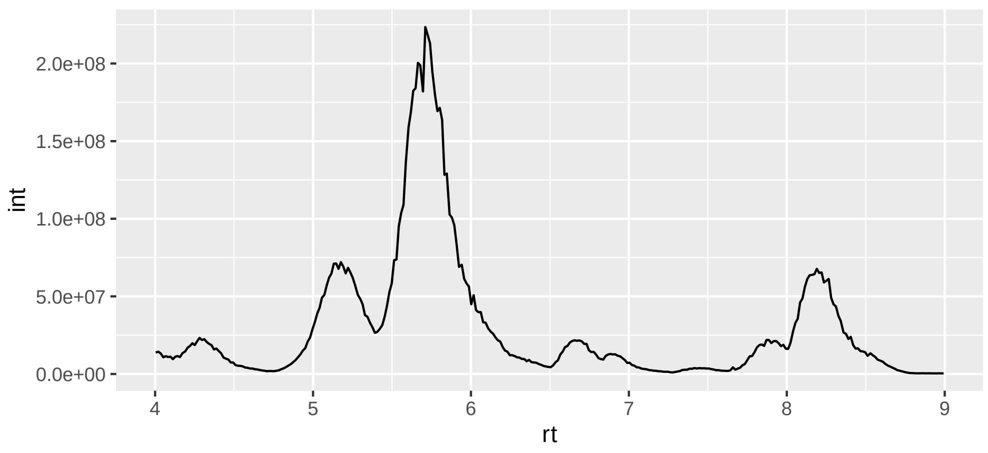
Simple enough!

A BPC is just like a TIC except that it records the *maximum* intensity measured, rather than the sum of all intensities. This data is also collected by the mass analyzer and doesn't need to be calculated.

```r
file_data <- grabMSdata(single_file, grab_what = "BPC")
```

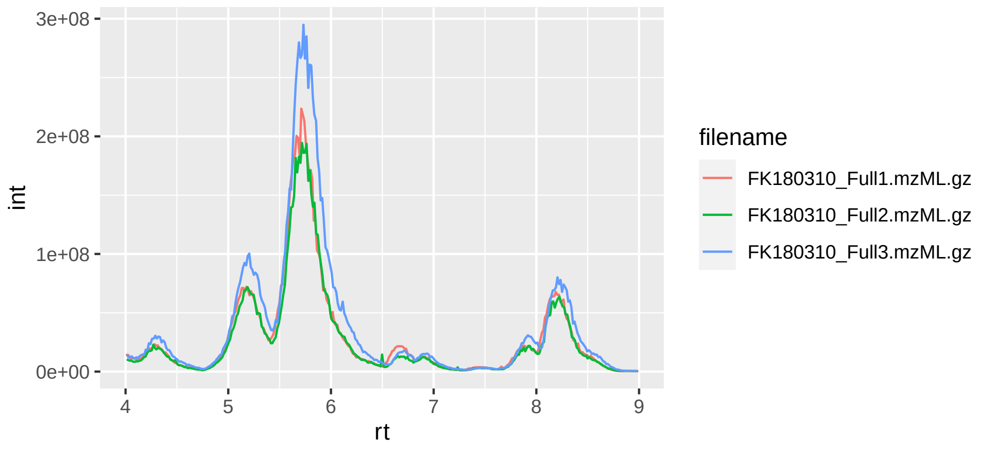
Since the data is parsed in a "tidy" format, it plays nicely with popular packages such as `ggplot2`. Let's use that to plot our BPC instead of the base R plotting system:

```r
library(tidyverse)
#> -- Attaching packages ------------------------------------------------------------------- tidyverse
#> v ggplot2 3.3.2     v purrr   0.3.4
#> v tibble  3.0.3     v dplyr   1.0.1
#> v tidyr   1.1.1     v stringr 1.4.0
#> v readr   1.3.1     v forcats 0.5.0
#> -- Conflicts ---------------------------------------------------------------------- tidyverse_confli
#> x dplyr::between()   masks data.table::between()
#> x dplyr::filter()    masks stats::filter()
#> x dplyr::first()     masks data.table::first()
#> x dplyr::lag()       masks stats::lag()
#> x dplyr::last()      masks data.table::last()
#> x purrr::transpose() masks data.table::transpose()
ggplot(file_data$BPC) + geom_line(aes(x=rt, y=int))
```



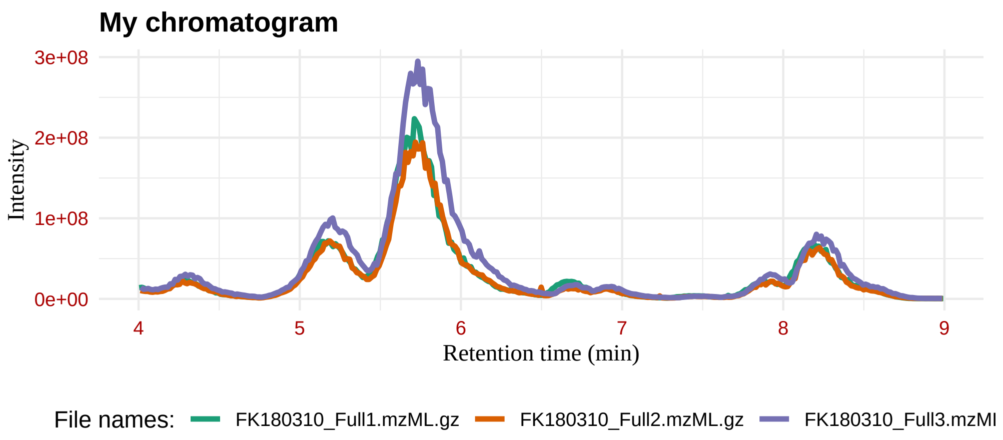
The advantages of tidy data and `ggplot` become clear when we load more than one file at a time because we can group and color by the third column, the name of the file from which the data was read. Here I've also enabled a progress bar with the argument `verbosity="minimal"` because it's nice to see progress when multiple files are being read.

```
file_data <- grabMSdata(data_files, grab_what = "BPC", verbosity = "minimal")
#>   |                                                                          |
#> Total time: 0.33  s

ggplot(file_data$BPC) + geom_line(aes(x=rt, y=int, color=filename))
```



And of course, this means that all of `ggplot`'s aesthetic power can be brought to your chromatograms as well, so customize away!

```
ggplot(file_data$BPC) +
  geom_line(aes(x=rt, y=int, color=filename), lwd=1.2) +
  theme_minimal() +
  theme(legend.position = "bottom",
        axis.text = element_text(color = "#AA0000"),
        axis.title = element_text(family = "serif"),
        plot.title = element_text(face = "bold")) +
  scale_colour_brewer(palette = "Dark2") +
  labs(x="Retention time (min)", y="Intensity",
       title = "My chromatogram", color="File names:")
```



RaMS also provides some basic file metadata extraction capability, although the focus for this package is on the actual data and other MS packages handle file metadata much more elegantly. This is one area where there are major differences between mzML and mzXML file types - the mzXML file type simply doesn't encode as much metadata as the mzML filetype, so RaMS can't extract it.

```
# Since the minification process strips a lot of useful data, I use DDA here
metadata_files <- list.files(msdata_dir, pattern = "DDA", full.names = TRUE)
grabMSdata(metadata_files, grab_what = "metadata")
#> $metadata
#>                                    source_file
#> 1: 200709_Poo_TruePooFK180310_DDApos100.raw
#> 2: 200709_Poo_TruePooFK180310_DDApos100.raw
#>                                                                                      inst_data
#> 1:                                                                                   Q Exactive
#> 2: Thermo Scientific,Q Exactive,electrospray ionization,quadrupole,inductive detector
#>          config_data           timestamp   mslevels polarity
#> 1: <data.frame[4x3]> 2020-07-09 22:57:28 MS 1, MS 2 positive
#> 2:                                  <NA>   MS1, MS2        +
#>                       filename
#> 1:  FK180310_DDApos100.mzML.gz
#> 2: FK180310_DDApos100.mzXML.gz
```

Adding a column: MS1 data

MS1 data can be extracted just as easily, by supplying "MS1" to the grab_what argument of grabMSdata function.

```
file_data <- grabMSdata(data_files, grab_what = "MS1")

knitr::kable(head(file_data$MS1, 3))
```

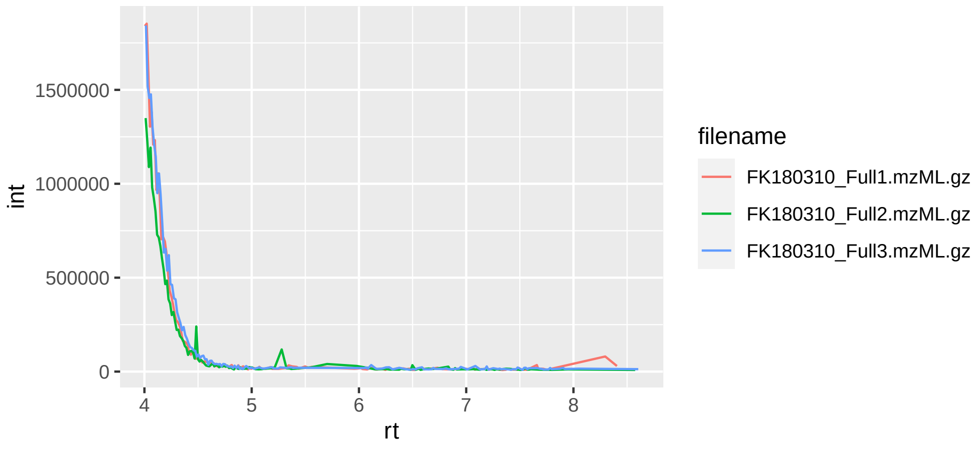
rt	mz	int	filename
4.003672	60.08151	8744.915	FK180310_Full1.mzML.gz
4.003672	90.05549	53195.234	FK180310_Full1.mzML.gz
4.003672	104.07107	639768.500	FK180310_Full1.mzML.gz

So we've now got the *mz* column, corresponding to the mass-to-charge ratio (m/z) of an ion. This means that we can now filter our data for specific masses and separate out molecules with different masses.

Note that this also makes the data much larger in R's memory - so don't go loading hundreds of files simultaneously. If that's necessary, check out the section below on saving space.

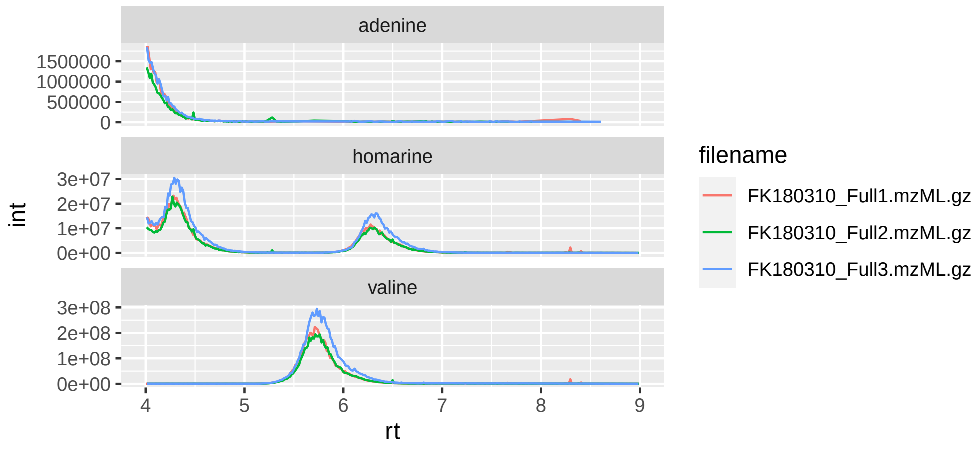
Because `RaMS` returns data.tables rather than normal `data.frames`, indexing is super-fast and a bit more intuitive than with base R. Below, I also use the `pmppm` function from `RaMS` to produce a mass range from an initial mass and spectrometer accuracy (here, 5 parts-per-million).

```r
adenine_mz <- 136.06232

adenine_data <- file_data$MS1[mz%between%pmppm(adenine_mz, ppm=5)]

ggplot(adenine_data) + geom_line(aes(x=rt, y=int, color=filename))
```



This makes it easy to grab the data for multiple compounds of interest with a simple loop, provided here by the `purrr` package of the tidyverse:

```r
masses_of_interest <- c(adenine=136.06232, valine=118.0865, homarine=138.055503)

mass_data <- imap(masses_of_interest, function(mz_i, name){
  cbind(file_data$MS1[mz%between%pmppm(mz_i, ppm=5)], name)
}) %>% rbindlist()

ggplot(mass_data) +
  geom_line(aes(x=rt, y=int, color=filename)) +
  facet_wrap(~name, ncol = 1, scales = "free_y")
```



Moving along: MS2 data

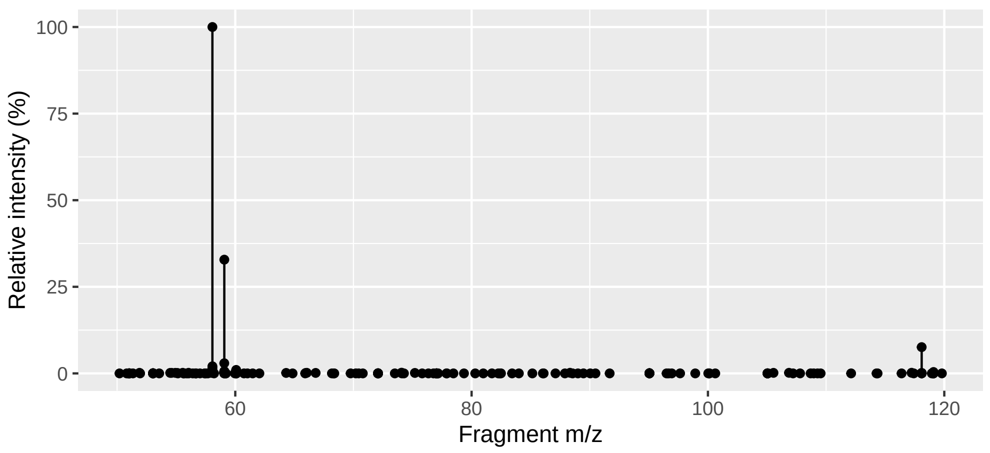
`RaMS` also handles MS2 data elegantly. Request it with the "MS2" option for `grab_what`, although it's often a good idea to grab the MS1 data alongside.

```
DDA_file <- list.files(msdata_dir, pattern = "DDA.*mzML", full.names = TRUE)
DDA_data <- grabMSdata(DDA_file, grab_what = c("MS2"))
knitr::kable(head(DDA_data$MS2, 3))
```

rt	premz	fragmz	int	voltage	filename
4.01227	757.017	57.46079	6493.131	100	FK180310_DDApos100.mzML.gz
4.01227	757.017	59.75748	6986.794	100	FK180310_DDApos100.mzML.gz
4.01227	757.017	62.73330	7692.589	100	FK180310_DDApos100.mzML.gz

DDA data can be plotted nicely with `ggplot2` as well. Typically it makes sense to filter for a precursor mass, then render the fragments obtained.

```
betaine_mass <- 118.0865

betaine_MS2 <- DDA_data$MS2[premz%between%pmppm(betaine_mass, 5)]
betaine_MS2$int <- betaine_MS2$int/max(betaine_MS2$int)*100

ggplot(betaine_MS2) +
  geom_point(aes(x=fragmz, y=int)) +
  geom_segment(aes(x=fragmz, xend=fragmz, y=int, yend=0)) +
  labs(x="Fragment m/z", y="Relative intensity (%)")
```



This is also the perfect place to enable some interactivity with packages such as `plotly`, making data exploration not only simple but also enjoyable.

```r
## Not run to save space in the vignette:
library(plotly)
data_files <- list.files(msdata_dir, pattern = "mzML", full.names = TRUE)
file_data <- grabMSdata(data_files, grab_what = c("MS1", "MS2", "BPC"))

clean_MS2 <- file_data$MS2 %>%
  filter(premz%between%pmppm(betaine_mass)) %>%
  group_by(rt) %>%
  arrange(desc(int)) %>%
  summarise(frags=paste(
    paste(round(fragmz, digits = 3), round(int), sep = ": "), collapse = "\n"),
    .groups="drop"
  )
file_data$MS1 %>%
  filter(mz%between%pmppm(betaine_mass)) %>%
  filter(!str_detect(filename, "DDA")) %>%
  plot_ly() %>%
  add_trace(type="scatter", mode="lines", x=~rt, y=~int, color=~filename,
            hoverinfo="none") %>%
  add_trace(type="scatter", mode="markers", x=~rt, y=0,
            text=~frags, hoverinfo="text", showlegend=FALSE,
            marker=list(color="black"), data = clean_MS2) %>%
  layout(annotations=list(x=min(clean_MS2$rt), y=0,
                          text="Mouse over to see\nMSMS fragments"))
```

Easy access to MS2 data also allows us to rapidly perform simple operations such as searching for a specific fragment mass. For example, if we know that glycine betaine typically produces a fragment with a mass of 58.0660, we simply subset the MS2 data for fragments in a range around that mass:

```r
betaine_frag_mz <- 58.0660
knitr::kable(head(DDA_data$MS2[fragmz%between%pmppm(betaine_frag_mz, ppm = 5)]))
```

rt	premz	fragmz	int	voltage	filename
4.178221	118.0865	58.06591	904490.2	100	FK180310_DDApos100.mzML.gz
4.191554	212.0951	58.06590	334911.0	100	FK180310_DDApos100.mzML.gz
4.528248	212.0950	58.06588	306575.6	100	FK180310_DDApos100.mzML.gz
4.537131	118.0864	58.06588	788627.7	100	FK180310_DDApos100.mzML.gz
4.867593	212.0951	58.06590	305300.6	100	FK180310_DDApos100.mzML.gz
4.927637	118.0865	58.06589	759520.3	100	FK180310_DDApos100.mzML.gz

We find that there's not only glycine betaine that produces that fragment, but also another compounds with a mass of 212.0951. Fragments like this can then be searched manually in online databases or, since the data is already in R, passed to a script that automatically searches them.

Similarly, we can easily search instead for neutral losses with this method. If we suspect other molecules are producing a similar neutral loss as glycine betaine:

```
betaine_mass <- 118.0865
betaine_neutral_loss <- betaine_mass - betaine_frag_mz

file_data$MS2 <- mutate(DDA_data$MS2, neutral_loss=premz-fragmz) %>%
  select("rt", "premz", "fragmz", "neutral_loss", "int", "voltage", "filename")
file_data$MS2[neutral_loss%between%pmppm(betaine_neutral_loss, ppm = 5)] %>%
  head() %>% knitr::kable()
```

rt	premz	fragmz	neutral_loss	int	voltage	filename
4.178221	118.0865	58.06591	60.02055	904490.2	100	FK180310_DDApos100.mzML.gz
4.537131	118.0864	58.06588	60.02054	788627.7	100	FK180310_DDApos100.mzML.gz
4.927637	118.0865	58.06589	60.02058	759520.3	100	FK180310_DDApos100.mzML.gz
5.281570	118.0864	58.06588	60.02053	708404.8	100	FK180310_DDApos100.mzML.gz
6.142964	138.0549	78.03445	60.02043	2307126.8	100	FK180310_DDApos100.mzML.gz
6.477283	138.0549	78.03446	60.02040	2133855.0	100	FK180310_DDApos100.mzML.gz

We can again confirm our suspicions that there's another molecule with a similar neutral loss: one with a mass of 138.0549.

Advanced RaMS usage

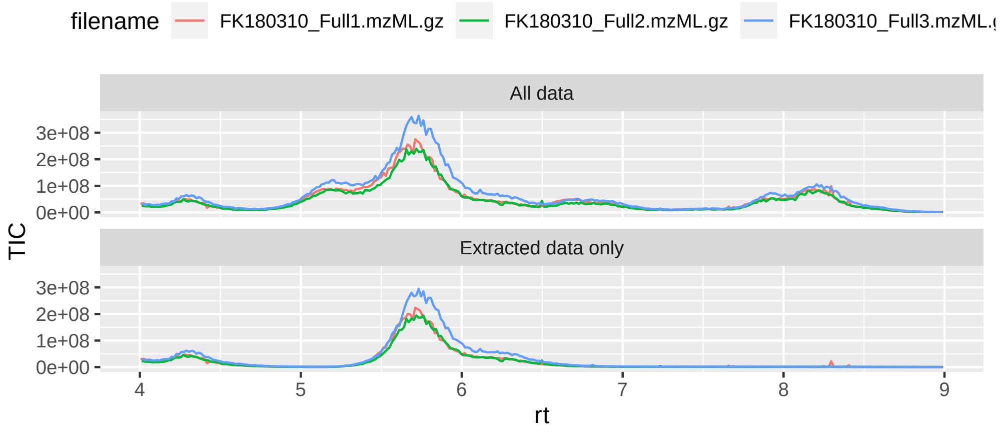
Saving space: EICs and rtrange

Mass-spec files are typically tens or hundreds of megabytes in size, which means that the simultaneous analysis of many files can easily exceed a computer's memory. Since `RaMS` stores all data in R's working memory, this can become a problem for large analyses.

However, much of the usage envisioned for `RaMS` on this scale doesn't require access to the entire file, the entire time. Instead, users are typically interested in a few masses of interest or a specific time window. This means that while each file still needs to be read into R in full to find the data of interest, extraneous data can be discarded before the next file is loaded.

This functionality can be enabled by passing "EIC" and/or "EIC_MS2" to the `grab_what` argument of `grabMSdata`, along with a vector of masses to extract (mz) and the instrument's ppm accuracy. When this is enabled, files are read into R's memory sequentially, the mass window is extracted, and the rest of the data is discarded.

```r
data_files <- list.files(msdata_dir, pattern = "mzML", full.names = TRUE)
all_data <- grabMSdata(data_files, grab_what = c("MS1", "MS2"))

masses_of_interest <- c(adenine=136.06232, valine=118.0865, homarine=138.055503)
small_data <- grabMSdata(data_files, grab_what = c("EIC", "EIC_MS2"),
                         mz=masses_of_interest, ppm = 5)

all_data$MS1 %>%
  mutate(type="All data") %>%
  rbind(small_data$EIC %>% mutate(type="Extracted data only")) %>%
  filter(!str_detect(filename, "DDA")) %>%
  group_by(rt, filename, type) %>%
  summarise(TIC=sum(int), .groups="drop") %>%
  ggplot() +
  geom_line(aes(x=rt, y=TIC, color=filename)) +
  facet_wrap(~type, ncol = 1) +
  theme(legend.position = "top")
```

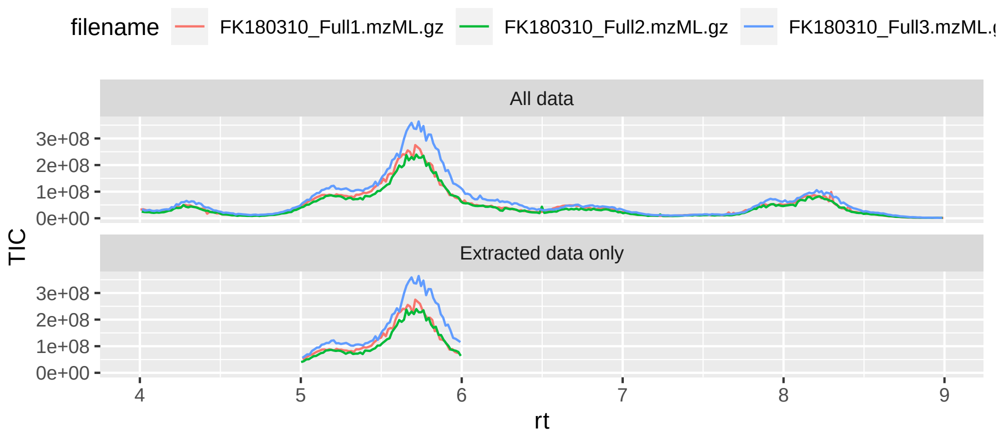


```r
# Size reduction factor:
as.numeric(object.size(all_data)/object.size(small_data))
#> [1] 44.19984
```

As expected, the size of the `small_data` object is much smaller than the `all_data` object, here by a factor of nearly 30x. For files that haven't already been "minified", that size reduction will be even more significant. Of course, this comes with the cost of needing to re-load the data a second time if a new mass feature becomes of interest but this shrinkage is expecially valuable for targeted analyses where the analytes of interest are known in advance.

A second way of reducing file size is to constrain the retention time dimension rather than the m/z dimension. This can be done with the `rtrange` argument, which expects a length-two vector corresponding to the upper and lower bounds on retention time. This is useful when a small time window is of interest, and only the data between those bounds is relevant.

```r
small_data <- grabMSdata(data_files, grab_what = c("MS1", "MS2"), rtrange = c(5, 6))

all_data$MS1 %>%
  mutate(type="All data") %>%
  rbind(small_data$MS1 %>% mutate(type="Extracted data only")) %>%
  filter(!str_detect(filename, "DDA")) %>%
  group_by(rt, filename, type) %>%
  summarise(TIC=sum(int), .groups="drop") %>%
  ggplot() +
  geom_line(aes(x=rt, y=TIC, color=filename)) +
  facet_wrap(~type, ncol = 1) +
  theme(legend.position = "top")
```



```
# Size reduction factor:
as.numeric(object.size(all_data)/object.size(small_data))
#> [1] 4.697372
```

The savings are more modest here, but constraining the retention time actually speeds up data retrieval slightly. Since m/z and intensity data is encoded in mzML and mzXML files while retention time information is not, eliminating scans with a retention time window removes the need to decode the intensity and m/z information in those scans.

However, decoding is rarely the rate-limiting step and for more information about speeding things up, continue to the next section.

Speeding things up

So, `RaMS` isn't fast enough for you? Let's see what we can do to improve that. The first step in speeding things up is discovering what's slow. Typically, this is the process of reading in the mzML/mzXML file rather than any processing that occurs afterward, but this is not always the case. To examine bottlenecks, `RaMS` includes timing information that's produced if the `verbosity` argument is set to "very".

```
all_data <- grabMSdata(data_files, grab_what = c("MS1", "MS2"), verbosity = "very")
#>   |                                                                      |
#> Reading file FK180310_DDApos100.mzML.gz... 0.324651 s
#> Reading MS1 data...0.28424 s
#> Reading MS2 data...0.1166899 s
#>   |                                                                      |===================
#> Reading file FK180310_Full1.mzML.gz... 0.05886507 s
#> Reading MS1 data...0.06582499 s
#> Reading MS2 data...0.008974791 s
#>   |                                                                                 |===================
#> Reading file FK180310_Full2.mzML.gz... 0.06482506 s
#> Reading MS1 data...0.06387115 s
#> Reading MS2 data...0.009995937 s
#>   |                                                                                 |===================
#> Reading file FK180310_Full3.mzML.gz... 0.05784392 s
#> Reading MS1 data...0.06937099 s
#> Reading MS2 data...0.008976936 s
#>   |                                                                                 |===================
#> Total time: 1.14  s
```

These minified demo files are pretty quick to load, but it looks like reading the file in takes about as long as extracting MS1 information, with MS2 extraction taking a fraction of that time.

In general, slow file read times can be improved by compressing the data. mzML files are highly compressible, with options to compress both the data itself using the `zlib` method and the files as a whole using `gzip`.

`gzip` is the simplest one to use, as a plethora of methods exist to compress files this way, including online sites. `RaMS` can read the data directly from a gzipped file, no decompression necessary, so this is an easy way to reduce file size and read times. Sharp-eyed users will have noticed that the demo files are already gzipped, which is part of the reason they are so small.

`zlib` compression is slightly trickier, and is most often performed with tools such as Proteowizard's `msconvert` tool with the option "-z".

Read times may also be slow if files are being accessed via the Internet, either through a VPN or network drive. If your files are stored elsewhere, consider first moving those files somewhere more local before reading data from them.

If the bottleneck appears when reading MS1 data, consider restricting the retention time range with the `rtrange` argument or using more detailed profiling tools such as RStudio's "Profile" options or the `profvis` package. Pull requests that improve data processing speed are always welcome!

While the package is not currently set up for parallel processing, this is a potential future feature if a strong need is demonstrated.

Finer control: grabMzmlData, grabMzxmlData

While there's only one main function (`grabMSdata`) in `RaMS`, you may have noticed that two other functions have been exposed that perform similar tasks: `grabMzmlData` and `grabMzxmlData`. The main function `grabMSdata` serves as a wrapper around these two functions, which detects the file type, adds the "filename" column to the data, and loops over multiple files if provided. However, there's often reason to use these internal functions separately.

For one, the objects themselves are smaller because they don't have the filename column attached yet. You as the user will need to keep track of which data belongs to which files in this case.

Another use case might be applying functions to each file individually, perhaps aligning to a reference chromatogram or identifying peaks. Rather than spending the time to bind the files together and immediately separate them again, these functions have been exposed to skip that step.

Finally, these functions are useful for parallelization. Because iterating over each mass-spec file is often the largest reasonable chunk, these functions can be passed directly to parallel processes like `mclapply` or `doParallel`. However, parallelization is a beast best handled by individual users because its actual implementation often differs wildly and its utility depends strongly on individual setups (remember that parallelization won't help with slow I/O times, so it may not always improve data processing speed.)

```r
## Not run:
data_files <- list.files(msdata_dir, pattern = "mzML", full.names = TRUE)

library(parallel)
output_data <- mclapply(data_files, grabMzmlData, mc.cores = detectCores()-1)

library(foreach)
library(doParallel)
registerDoParallel(numCores)
output_data <- foreach (i=data_files) %dopar% {
  grabMzmlData(i)
}
stopImplicitCluster()
```

The nitty-gritty details

Wow, you really made it this far through the vignette? Well, your reward is some very dry information about what `RaMS` does under the hood.

`RaMS` is possible because mzML and mzXML documents are fundamentally XML-based. This means that we can leverage speedy and robust XML parsing packages such as `xml2` to extract the data. Fundamentally, `RaMS` relies on XPath navigation to collect various bits of mass-spec data, and the format of mzML and mzXML files provides the tags necessary. That means a lot of `RaMS` code consists of lines like the following:

```r
## Not run:
data_nodes <- xml2::xml_find_all(mzML_nodes, xpath="//d1:precursorMz")
raw_data <- xml2::xml_attr(data_nodes, "value")
```

…plus a lot of data handling to get the output into the tidy format.

The other tricky bit of data extraction is converting the (possibly compressed) binary data into R-friendly objects. This is usually handled with code like that shown below. Many of the settings can be deduced from the file, but sometimes compression types need to be guessed at and will throw a warning if so.

```r
## Not run:
decoded_binary <- base64enc::base64decode(binary)
raw_binary <- as.raw(decoded_binary)
decomp_binary <- memDecompress(raw_binary, type = file_metadata$compression)
final_binary <- readBin(decomp_binary, what = "double",
                        n=length(decomp_binary)/file_metadata$mz_precision,
                        size = file_metadata$mz_precision)

# See https://github.com/ProteoWizard/pwiz/issues/1301
```

Fundamentally, mass-spectrometry data is formatted as a ragged array, with an unknown number of m/z and intensity values for a given scan. This makes it difficult to encode neatly without interpolating, but tidy data provides a solution by stacking those arrays rather than aligning them into some sort of matrix.

This ragged shape is also the reason that subsetting mass-spec data by retention time is trivial - grab the scans that correspond to the desired retention times and you're done. Subsetting by mass, on the other hand, requires decoding each and every scan's m/z and intensity data. If you're reading a book and only want a couple chapters, it's easy to flip to those sections. If you're looking instead for every time a specific word shows up, you've gotta read the whole thing.

For more information about the mzML data format and its history, check out the specification at http://www.psidev.info/mzML.