Cross-platform MS with RaMS

Updated 2021-01-31

The strength of `RaMS` is its simple data format. Table-like data structures are common in most programming languages, and they can always be converted to the nigh-universal matrix format. The goal of this vignette is to illustrate this strength by exporting MS data to several formats that can be used outside of R.

Standard export to CSV

As with all rectangular data, `RaMS` objects can be easily exported to CSV files with base R functions. This works best with a few chromatograms at a time, as the millions of data points found in most MS files can overwhelm common file readers.

```
library(RaMS)

# Locate an MS file
single_file <- system.file("extdata", "LB12HL_AB.mzML.gz", package = "RaMS")

# Grab the MS data
file_data <- grabMSdata(single_file, grab_what = "everything")

# Write out MS1 data to .csv file
write.csv(x = file_data$MS1, file = "MS1_data.csv")

# Clean up afterward
file.remove("MS1_data.csv")
```

```
## [1] TRUE
```

Fancier export to Excel

Excel workbooks are a common format because of their intuitive GUI and widespread adoption. They can also encode more information than CSV files due to their multiple "sheets" within a single workbook - perfect for encoding both MS1 and MS2 information in one place. This vignette uses the `openxlsx` package, although there are several alternatives with identical functionality.

```
library(openxlsx)

# Locate an MS2 file
MS2_file <- system.file("extdata", "DDApos_2.mzML.gz", package = "RaMS")

# Grab the MS1 and MS2 data
MS2_data <- grabMSdata(MS2_file, grab_what=c("MS1", "MS2"))

# Write out MS data to Excel file
```

```
# openxlsx writes each object in a list to a unique sheet
# Produces one sheet for MS1 and one for MS2
write.xlsx(MS2_data, file = "MS2_data.xlsx")

# Clean up afterward
file.remove("MS2_data.xlsx")
```

```
## [1] TRUE
```

Exporting to SQL database

For more robust data processing and storage, or to work with larger-than-memory data sets, SQL databases are an excellent choice. This vignette will demo the `RSQLite` package's engine, although several other database engines have similar functionality.

```
library(DBI)

# Create the sqlite database and connect to it
MSdb <- dbConnect(RSQLite::SQLite(), "MSdata.sqlite")

# Export MS1 and MS2 data to sqlite tables
dbWriteTable(MSdb, "MS1", MS2_data$MS1)
dbWriteTable(MSdb, "MS2", MS2_data$MS2)
dbListTables(MSdb)
```

```
## [1] "MS1" "MS2"
```

```
# Perform a simple query to ensure data was exported correctly
dbGetQuery(MSdb, 'SELECT * FROM MS1 LIMIT 3')
```

```
##        rt       mz       int         filename
## 1 240.051 80.05009 12057.776 DDApos_2.mzML.gz
## 2 240.051 80.26269  8178.767 DDApos_2.mzML.gz
## 3 240.051 80.94841 19075.213 DDApos_2.mzML.gz
```

```
# Perform EIC extraction in SQL rather than in R
EIC_query <- 'SELECT * FROM MS1 WHERE mz BETWEEN :lower_bound AND :upper_bound'
query_params <- list(lower_bound=118.086, upper_bound=118.087)
EIC <- dbGetQuery(MSdb, EIC_query, params = query_params)

# Disconnect after export
dbDisconnect(MSdb)

# Clean up afterward
unlink("MSdata.sqlite")
```

Interfacing with Python via `reticulate`

R and Python are commonly used together, and the `reticulate` package makes this even easier by enabling a Python interpreter within R. RStudio, in which this vignette was written, supports both R and Python code chunks as shown below.

R code chunk: {r}

```r
# Locate a couple MS files
data_dir <- system.file("extdata", package = "RaMS")
file_paths <- list.files(data_dir, pattern = "HL.*mzML", full.names = TRUE)

msdata <- grabMSdata(files = file_paths, grab_what = "BPC")$BPC
```

Python code chunk: {python}

```python
import seaborn as sns
import matplotlib.pyplot as plt

sns.relplot(data=r.msdata, kind="line", x="rt", y="int", hue="filename")
```

```
## <seaborn.axisgrid.FacetGrid object at 0x000000006591BC10>
```

```python
plt.show()
```