Cardinal 2.0: User's Guide

Kylie A. Bemis

February 21, 2019

Contents

1 Introduction and installation

Cardinal 2.0 provides new classes and methods for the manipulation, transformation, visualization, and analysis of imaging experiments–specifically mass spectrometry (MS) imaging experiments.

Classes and methods from older versions of Cardinal will continue to be supported; however, new development should focus on the new classes implemented in Cardinal 2.0.

New features include:

- New imaging data classes such as `ImagingExperiment`, `SparseImagingExperiment`, and `MSImagingExperiment` which will provide better support for larger-than-memory datasets
- New imaging metadata classes such as `PositionDataFrame` and `MassDataFrame` which make it easier to manipulate experimental runs, pixel coordinates, and m/z-values by storing them as separate slots rather than ordinary columns
- New `plot()` and `image()` visualization methods that can handle non-gridded pixel coordinates and allow assigning the resulting plot (and data) to a variable for later re-plotting
- Support for writing imzML in addition to reading; improved support and options for importing larger-than-memory imzML for both 'continuous' and 'processed' formats
- Data manipulation and summarization verbs borrowed from *dplyr* such as `select()`, `filter()`, `mutate()`, and `summarize()` for easier transformation, subsetting, and summarization of imaging datasets
- Delayed batch processing via a new `process()` method that allows queueing of preprocessing methods such as `normalize()` and `peakPick()` for later parallel execution

This document provides an introduction to several new and existing features of *Cardinal*.

Cardinal can be installed via the *BiocManager* package.

```
> install.packages("BiocManager")
> BiocManager::install("Cardinal")
```

The same function can be used to update *Cardinal* and other Bioconductor packages.

2 Data import and export

In order to be imported into R by *Cardinal*, input data must be in either imzML or Analyze 7.5 format. Data can be loaded into memory, or it can be attached on-disk using the `attach.only=TRUE` argument to the data import function.

2.1 imzML

To import or attach imzML, use the `readImzML()` function. Both files must be present in the same folder and have the same name (except for the file extension).

```
> name <- "common name of your .imzML and .ibd files"
> folder <- "/path/to/the/folder/containing/the/files"
> data <- readImzML(name, folder, as="MSImagingExperiment")
```

The imzML format is an open standard designed specifically for interchange of mass spectrometry imaging datasets [1]. Many other formats can be converted to imzML with the help of free applications available online. See http://www.imzml.org for more information and links to free converters.

The imzML format uses two files with extensions '.imzML' and '.ibd' to store data. The former is an XML-based human-readable text file that stores the metadata about the MS imaging experiment. The latter is a binary file storing the m/z and intensity data.

The `as="MSImagingExperiment"` argument is optional and may be used to specify whether to load the data using new Cardinal 2.0 classes (`MSImagingExperiment`) or legacy Cardinal classes (`MSImageSet`). Its defaults may change in future versions of Cardinal.

```
> # import large datasets without loading them into memory
> data <- readImzML(name, folder, attach.only=TRUE, as="MSImagingExperiment")
```

Large imzML files can be attached on-disk without fully loading them into memory by using the `attach.only=TRUE` option. Not all *Cardinal* features are supported for on-disk datasets.

Both 'continuous' and 'processed' imzML format are supported. When using on-disk data with `attach.only=TRUE`, accessing images of 'processed' data may be slow (but accessing spectra should remain fast). This is due to the way the data is stored in the imzML file.

```
> # import 'processed' data between m/z 500 - 600
> data <- readImzML(name, folder, mass.range=c(500,600), as="MSImagingExperiment")
> # import 'processed' data binned to 100 ppm
> data <- readImzML(name, folder, resolution=100, units="ppm", as="MSImagingExperiment")
> # import 'processed' data binned to 1 m/z
> data <- readImzML(name, folder, resolution=1, units="mz", as="MSImagingExperiment")
```

For 'processed' imzML files, there are additional options. The `resolution` and `units` arguments determine how the data is binned. If the data is imported using the new classes (`MSImagingExperiment`), this can be changed and the spectra re-binned on-the-fly later. Additionally, the `mass.range` argument allows specifying the mass range, which can be computationally efficient. This avoids having to parse this information from the data (which can potentially take a long time for very large datasets).

```
> writeImzML(data, name, folder, mz.type="64-bit float", intensity.type="32-bit float")
```

The `writeImzML()` function can be used to write an MS imaging dataset to an imzML file. For 'processed' data, re-importing the same dataset may result in the spectra being binned differently; however, the underlying data is preserved. Any metadata columns will not be written to the file.

For more information on reading and writing imzML files, see `?readImzML` and `?writeImzML`.

2.2 Analyze 7.5

Originally designed for MRI data by the Mayo Clinic, Analyze 7.5 is another common format used for mass spectrometry imaging data.

The Analyze format uses a collection of three files with extensions '.hdr', '.img', and '.t2m' to store data. To read datasets stored in the Analyze format, use the `readAnalyze()` function. All three files must be present in the same folder and have the same name (except for the file extension) for the data to be read properly.

```
> name <- "common name of your .hdr, .img, and .t2m files"
> folder <- "/path/to/the/folder/containing/the/files"
> data <- readAnalyze(name, folder, as="MSImagingExperiment")
```

The `as="MSImagingExperiment"` argument is optional may be used to specify whether to load the data using new Cardinal 2.0 classes (`MSImagingExperiment`) or legacy Cardinal classes (`MSImageSet`). Its defaults may change in future versions of Cardinal.

```
> # import large datasets without loading them into memory
> data <- readAnalyze(name, folder, attach.only=TRUE, as="MSImagingExperiment")
```

Large Analyze files can be attached on-disk without fully loading them into memory by using the `attach.only=TRUE` option. Not all *Cardinal* features are supported for on-disk datasets.

```
> writeAnalyze(data, name, folder, intensity.type="16-bit integer")
```

The `writeAnalyze()` function can be used to write an MS imaging dataset to an Analyze 7.5 file. Any metadata columns will not be written to the file.

For more information on reading Analyze files, see `?readAnalyze` and `?writeAnalyze`.

2.3 Using `readMSIData()` and `writeMSIData()`

```
> file <- "/path/to/an/imaging/data/file.extension"
> data <- readMSIData(file, as="MSImagingExperiment")
```

Cardinal also provides the convenience functions `readMSIData()` and `writeMSIData()`, which will attempt to automatically infer the correct function to use from the provided file extensions. The same rules for naming conventions apply as described above, but one need only provide the path to any of the data files. For example, to read/write an Analyze file, providing a path containing any of '.hdr', '.img', or '.t2m'. Likewise, to read or write imzML, provide a path containing either '.imzML' or '.ibd'.

Any additional arguments will be passed to the appropriate underlying read/write functions.

3 Components of an imaging experiment

In *Cardinal*, imaging experiment datasets are composed of multiple sets of metadata, in addition to the actual experimental data. These are (1) pixel metadata, (2) feature (m/z) metadata, (3) the actual imaging data, and (4) a class that holds all of these and represents the experiment as a whole.

Unlike many software packages designed for analysis of MS imaging experiments, *Cardinal* is designed to work with multiple datasets and incorporate all aspects of experimental design and metadata.

This section will discuss in detail each component of an imaging dataset in *Cardinal*, assuming the context of MS. If you would like to jump straight into working with a dataset, you may skip ahead to the next section and return here when you have questions on particular aspects of an imaging dataset.

3.1 Metadata with `XDataFrame`

For storing metadata related to pixels (i.e., individual mass spectra, their coordinates, etc.), and features (i.e., m/z-values, peaks, proteins, lipids, etc.), *Cardinal* extends the `DataFrame` object from the *S4Vectors* package.

The `XDataFrame` class is a data frame class with eXtra "slot-columns" for storing additional metadata columns separately from ordinary columns. These additional "slot-columns" may be required to obey special rules not required of other columns.

```
> xdf <- XDataFrame(a=1:10, b=letters[1:10])
> xdf

XDataFrame with 10 rows and 2 columns
            a           b
   <integer> <character>
1           1           a
2           2           b
3           3           c
4           4           d
5           5           e
6           6           f
7           7           g
8           8           h
9           9           i
10         10           j
```

An ordinary `XDataFrame` behaves the same as an ordinary data frame. It exists primarily to allow shared behavior for its sub-classes `PositionDataFrame` and `MassDataFrame`.

3.1.1 Pixel metadata with `PositionDataFrame`

The `PositionDataFrame` class provides extra columns for storing spatial coordinates and identifying experimental runs.

```
> coord <- expand.grid(x=1:9, y=1:9)
> run <- factor(rep("Run 1", nrow(coord)))
> pid <- seq_len(nrow(coord))
> pdata <- PositionDataFrame(run=run, coord=coord, pid=pid)
> pdata
```

```
PositionDataFrame with 81 rows and 1 column
       :run:   coord:x   coord:y       pid
    <factor> <integer> <integer> <integer>
1      Run 1         1         1         1
2      Run 1         2         1         2
3      Run 1         3         1         3
4      Run 1         4         1         4
5      Run 1         5         1         5
...      ...       ...       ...       ...
77     Run 1         5         9        77
78     Run 1         6         9        78
79     Run 1         7         9        79
80     Run 1         8         9        80
81     Run 1         9         9        81
```

A `PositionDataFrame` can be created with any number of ordinary columns (or none), but the `run` and `coord` must always be provided.

```
> head(run(pdata))
```

```
[1] Run 1 Run 1 Run 1 Run 1 Run 1 Run 1
Levels: Run 1
```

The `run` slot-column, accessed via `run()`, uniquely identifies experimental runs. It must be a `factor`. It is analogous to the "sample" column in the legacy `IAnnotatedDataFrame` class from older versions of Cardinal, but with a less-confusing name. For MS imaging experiments, this typically identifies data gathered from unique slides.

```
> coord(pdata)
```

```
DataFrame with 81 rows and 2 columns
            x         y
    <integer> <integer>
1           1         1
2           2         1
3           3         1
4           4         1
5           5         1
...       ...       ...
77          5         9
78          6         9
79          7         9
80          8         9
81          9         9
```

The `coord` slot-columns, accessed via `coord()`, provides the spatial coordinates of pixels in an imaging experiment. It must be a `DataFrame` with numeric columns. The spatial coordinates do not need to be unique; they do not need to be integers; they may also be redundant with other columns such as `run`.

```
> gridded(pdata)

[1] TRUE

> resolution(pdata)

x y
1 1
```

The `PositionDataFrame` object provides several additional methods useful for plotting and manipulating imaging data. The `gridded()` and `resolution()` return whether the pixels lie on a gridded raster and their spatial resolution, which are calculated automatically from `coord`.

See `?PositionDataFrame` for additional methods.

3.1.2 Feature metadata with `MassDataFrame`

The `MassDataFrame` class provides extra columns for storing the m/z-values associated with mass spectral features.

```
> mz <- seq(from=500, to=600, by=0.2)
> fid <- seq_along(mz)
> fdata <- MassDataFrame(mz=mz, fid=fid)
> fdata

MassDataFrame with 501 rows and 1 column
         :mz:       fid
    <numeric> <integer>
1         500         1
2       500.2         2
3       500.4         3
4       500.6         4
5       500.8         5
...       ...       ...
497     599.2       497
498     599.4       498
499     599.6       499
500     599.8       500
501       600       501
```

A `MassDataFrame` can be created with any number of ordinary columns (or none), but the `mz` column must always be provided.

```
> head(mz(fdata))

[1] 500.0 500.2 500.4 500.6 500.8 501.0

> resolution(fdata)
```

```
 mz
0.2
```

The m/z-values can be accessed via `mz()`. They must be a non-negative numeric vector sorted in increasing order. An approximate m/z resolution can be accessed via `resolution()`.

3.2 Image data with `ImageArrayList`

The `ImageList` and `ImageArrayList` classes are list-like classes used to store the actual imaging data. `ImageList` allows elements of type, as long as they are array-like (i.e., have a `dim` attribute). `ImageArrayList` further imposes that each element has the same number of dimensions–specifically, the first two dimensions are treated as "rows" and "columns", and these two dimensions must be equal for all elements.

```
> set.seed(1)
> data0 <- generateSpectrum(nrow(pdata), range=c(500, 600), peaks=3,
+   baseline=3000, noise=0.01, sd=0.5, resolution=300, step=0.2)
> data1 <- generateSpectrum(nrow(pdata), range=c(500, 600), peaks=3,
+   baseline=3000, noise=0.01, sd=0.5, resolution=300, step=0.2)
> idata <- ImageArrayList(list(data0=data0$x, data1=data1$x))
> idata

Reference class object of class "SimpleImageArrayList"
Field "data":
List of length 2
names(2): data0 data1
```

An `ImageArrayList` can be created with a constructor of the same name. It takes a list of data elements.

```
> dim(idata[[1]])

[1] 501  81

> dim(idata[["data1"]])

[1] 501  81
```

The data elements of an `ImageArrayList` can be accessed via subsetting with double-brackets like an ordinary list.

3.3 MS imaging experiments with `MSImagingExperiment`

The `MSImagingExperiment` class represents MS imaging experiments as a whole, and brings together the pixel metadata, feature (m/z) metadata, and experimental imaging data.

In the long-term, it is intended to replace the legacy `MSImageSet` class from older versions of Cardinal. `MSImagingExperiment` is designed to be more flexible and provide better support for large datasets.

If needed, the `SparseImagingExperiment` superclass provides a more general interface for working with imaging experiments, not specific to MS.

```
> msdata <- MSImagingExperiment(imageData=idata, featureData=fdata, pixelData=pdata)
> msdata

An object of class 'MSImagingExperiment'
  <501 feature, 81 pixel> imaging dataset
    imageData(2): data0 data1
    featureData(1): fid
    pixelData(1): pid
    processing complete(0):
    processing pending(0):
    raster dimensions(2): x := 9, y := 9
    mass range: 500 to 600
    centroided: FALSE
```

The individual components can be accessed via the `pixelData()`, `featureData()`, and `imageData()` methods.

```
> imageData(msdata)

Reference class object of class "SimpleImageArrayList"
Field "data":
List of length 2
names(2): data0 data1

> pixelData(msdata)

PositionDataFrame with 81 rows and 1 column
       :run:   coord:x   coord:y       pid
    <factor> <integer> <integer> <integer>
1      Run 1         1         1         1
2      Run 1         2         1         2
3      Run 1         3         1         3
4      Run 1         4         1         4
5      Run 1         5         1         5
...      ...       ...       ...       ...
77     Run 1         5         9        77
78     Run 1         6         9        78
79     Run 1         7         9        79
80     Run 1         8         9        80
81     Run 1         9         9        81

> featureData(msdata)

MassDataFrame with 501 rows and 1 column
         :mz:       fid
    <numeric> <integer>
1         500         1
2       500.2         2
3       500.4         3
4       500.6         4
5       500.8         5
...       ...       ...
497     599.2       497
498     599.4       498
```

```
499     599.6       499
500     599.8       500
501       600       501
```

While `pData()` and `fData()` can be used as shortcuts for `pixelData()` and `featureData()`, as in older versions of Cardinal, the `iData()` function for accessing the imaging data works slightly differently.

```
> dim(iData(msdata))

[1] 501  81

> dim(iData(msdata, 1))

[1] 501  81

> dim(iData(msdata, "data0"))

[1] 501  81

> dim(spectra(msdata))

[1] 501  81
```

The `iData()` function directly accesses the data elements of `imageData()`. By default, it returns the first dataset. It can take an additional argument to specify which dataset to return. The `spectra()` function is an analog for it for `MSImagingExperiment` objects.

Cardinal provides a few subclasses of `MSImagingExperiment` that are specialized for particular kinds of data: `MSContinuousImagingExperiment` and `MSProcessedImagingExperiment`.

3.3.1 Continuous MS imaging experiments

The `MSContinuousImagingExperiment` subclass is specialized for MS imaging datasets where the spectra are stored in a dense matrix, either in-memory or on-disk. This includes MS experiments loaded from 'continuous' imzML files.

An `MSContinuousImagingExperiment` object is created automatically from the `MSImagingExperiment()` constructor function when provided a dense R matrix or `matter_matc` matrix.

```
> msdata0 <- MSImagingExperiment(imageData=data0$x, featureData=fdata, pixelData=pdata)
> msdata0

An object of class 'MSContinuousImagingExperiment'
  <501 feature, 81 pixel> imaging dataset
    imageData(1): intensity
    featureData(1): fid
    pixelData(1): pid
    processing complete(0):
    processing pending(0):
    raster dimensions(2): x := 9, y := 9
    mass range: 500 to 600
    centroided: FALSE
```

This class works nearly identically to `MSImagingExperiment`, but methods written for it can assume that the data is stored densely, and both spectra and images can be accessed relatively quickly and efficiently.

3.3.2 Processed MS imaging experiments

The `MSProcessedImagingExperiment` subclass is specialized for MS imaging datasets where the spectra are stored sparsely, either in-memory or on-disk. This includes MS experiments loaded from 'processed' imzML files.

An `MSProcessedImagingExperiment` object is created automatically from the `MSImagingExperiment()` constructor function when provided a `sparse_matc` matrix.

```
> t <- matter::rep_vt(list(data1$t), ncol(data1$x))
> x <- lapply(1:ncol(data1$x), function(i) data1$x[,i])
> data1b <- matter::sparse_mat(data=list(keys=t, values=x),
+   nrow=length(t[[1]]), ncol=length(x), keys=t[[1]])
```

A `sparse_matc` matrix is a sparse matrix as implemented by the *matter* package. Each column is sparse and are stored as key-value pairs, which may either be an R list in-memory, or a `matter_list` stored on-disk.

```
> msdata1 <- MSImagingExperiment(imageData=data1b, featureData=fdata, pixelData=pdata)
> msdata1

An object of class 'MSProcessedImagingExperiment'
  <501 feature, 81 pixel> imaging dataset
    imageData(1): intensity
    featureData(1): fid
    pixelData(1): pid
    processing complete(0):
    processing pending(0):
    raster dimensions(2): x := 9, y := 9
    mass range: 500 to 600
    centroided: FALSE
```

This class works very similarly to `MSImagingExperiment`, but methods written for it should assume that the data is stored sparsely, so spectra can be accessed relatively quickly and efficiently, but images may be accessed more slowly.

Additionally, the original, observed m/z-values are stored as the keys for reconstructing the spectra. They may not match the canonical vector of m/z-values accessed via `mz()`. This means that spectra are binned on-the-fly.

The data accessed via `iData()` and `spectra()` are the binned data. The original, observed m/z-values and intensities can be accessed via `mzData()` and `peakData()`.

```
> head(mzData(msdata1)[[1]]) # m/z of spectrum 1

[1] 500.0 500.2 500.4 500.6 500.8 501.0

> head(peakData(msdata1)[[1]]) # intensities of spectrum 1

[1] 3.890952 4.577313 4.223219 4.362255 4.348984 3.061082

> head(mzData(msdata1)[[2]]) # m/z of spectrum 2
```

```
[1] 500.0 500.2 500.4 500.6 500.8 501.0

> head(peakData(msdata1)[[2]]) # intensities of spectrum 2

[1] 5.586714 4.036220 4.926360 4.775202 4.862122 4.199646
```

Parameters for how the data are binned for `MSProcessedImagingExperiment` objects can be changed with the `tolerance()` method.

4 Data manipulation and transformation

Cardinal provides a number of methods for manipulation and transformation of imaging datasets. They are described below.

4.1 Subsetting and combining imaging experiments

`MSImagingExperiment` objects can be subset as a whole using the standard semantics of R, where the "rows" are the mass features, and the "columns" are the pixels.

```
> msdata[1:10,]

An object of class 'MSImagingExperiment'
  <10 feature, 81 pixel> imaging dataset
    imageData(2): data0 data1
    featureData(1): fid
    pixelData(1): pid
    processing complete(0):
    processing pending(0):
    raster dimensions(2): x := 9, y := 9
    mass range: 500.0 to 501.8
    centroided: FALSE

> msdata[,1:10]

An object of class 'MSImagingExperiment'
  <501 feature, 10 pixel> imaging dataset
    imageData(2): data0 data1
    featureData(1): fid
    pixelData(1): pid
    processing complete(0):
    processing pending(0):
    raster dimensions(2): x := 9, y := 2
    mass range: 500 to 600
    centroided: FALSE
```

Datasets can likewise be combined using `rbind()` and `cbind()`, where the "rows" are the mass features, and the "columns" are the pixels.

```
> cbind(msdata0, msdata1)
```

```
An object of class 'MSContinuousImagingExperiment'
  <501 feature, 162 pixel> imaging dataset
    imageData(1): intensity
    featureData(2): fid fid
    pixelData(1): pid
    processing complete(0):
    processing pending(0):
    raster dimensions(2): x := 9, y := 9
    mass range: 500 to 600
    centroided: FALSE
```

For `cbind()`, the mass features (i.e., the m/z-values) must match between all of the datasets. For `rbind()`, the run information and spatial coordinates must match.

Several data manipulation verbs are borrowed from the *dplyr* package for subsetting imaging experiments as well.

```
> select(msdata, x < 4, y < 4) # select based on pixels

An object of class 'MSImagingExperiment'
  <501 feature, 9 pixel> imaging dataset
    imageData(2): data0 data1
    featureData(1): fid
    pixelData(1): pid
    processing complete(0):
    processing pending(0):
    raster dimensions(2): x := 3, y := 3
    mass range: 500 to 600
    centroided: FALSE

> filter(msdata, mz < 550) # filter based on m/z features

An object of class 'MSImagingExperiment'
  <250 feature, 81 pixel> imaging dataset
    imageData(2): data0 data1
    featureData(1): fid
    pixelData(1): pid
    processing complete(0):
    processing pending(0):
    raster dimensions(2): x := 9, y := 9
    mass range: 500.0 to 549.8
    centroided: FALSE
```

The `select()` function subsets based on pixels ("columns") of the imaging dataset.

The `filter()` function subsets based on the mass features ("rows") of the imaging dataset.

Similar to those functions from *dplyr*, the names of metadata columns such as 'mz' can be used literally. They can also be chained together via the `%>%` operator.

```
> msdata %>%
+   select(x < 5, y < 5) %>%
+   filter(mz > 525)

An object of class 'MSImagingExperiment'
```

```
<375 feature, 16 pixel> imaging dataset
  imageData(2): data0 data1
  featureData(1): fid
  pixelData(1): pid
  processing complete(0):
  processing pending(0):
  raster dimensions(2): x := 4, y := 4
  mass range: 525.2 to 600.0
  centroided: FALSE
```

4.2 Using `pixelApply()` and `featureApply()`

The `pixelApply()` and `featureApply()` functions apply functions over pixels (i.e., mass spectra) or over features (i.e., flattened images). When applied to new *Cardinal* 2.0 classes, these can be executed in parallel.

```
> tic <- pixelApply(msdata, sum, BPPARAM=SerialParam()) # calculate TIC
> head(tic)

[1] 2662.5428  780.9577 3296.9571 4113.9654 3984.4219  774.6736

> ms <- featureApply(msdata, mean, BPPARAM=SerialParam()) # calculate mean spectrum
> head(ms)

[1] 4.616381 4.586164 4.306445 4.535474 4.544348 4.519470
```

Both functions take an argument `BPPARAM` which will be passed to the `bplapply()` function. By default, the registered parallel backend will be used. Otherwise, the specified backend will be used.

Note that for true parallel backends (i.e., not `SerialParam()`), *Cardinal* cannot print a progress bar to the console. A progress bar must be specified as part of the parallel backend, which will be updated according to the number of workers (rather than the number of iterations).

See `?pixelApply` for more details on how the function is applied, and other avaialble options to these functions. See `?bplapply` for more information on the parallel backends.

4.3 Summarization of imaging experiments

The `summarize()` function allows efficient summarization over an imaging dataset. Summarization can be applied over either pixels or features. Internally, it is implemented using `pixelApply()` and `featureApply()`.

```
> summarize(msdata, sum, .by="pixel") # calculate TIC

PositionDataFrame with 81 rows and 1 column
       :run:   coord:x   coord:y             sum
    <factor> <integer> <integer>       <numeric>
1      Run 1         1         1 2662.54280681174
2      Run 1         2         1 780.957674414635
```

```
3      Run 1         3         1 3296.95709988663
4      Run 1         4         1 4113.96542364712
5      Run 1         5         1 3984.42193198865
...      ...       ...       ...              ...
77     Run 1         5         9 2953.88443202541
78     Run 1         6         9 1017.10356322582
79     Run 1         7         9 3793.71846430599
80     Run 1         8         9 3427.26750932456
81     Run 1         9         9 801.721408901517
```

```
> summarize(msdata, .stat="mean") # calculate mean spectrum
```

```
MassDataFrame with 501 rows and 1 column
         :mz:             mean
    <numeric>        <numeric>
1         500 4.61638108877944
2       500.2 4.58616391303992
3       500.4 4.30644514134728
4       500.6  4.5354744085881
5       500.8  4.5443475889983
...       ...              ...
497     599.2  4.5866876186558
498     599.4 4.36153866779736
499     599.6 4.19352262818514
500     599.8 4.47338108221439
501       600 4.54975620378789
```

Either a summary function(s) can be provided explicitly, or a small number of statistics can be chosen from the `.stat` argument.

Statistics calculated via the `.stat` argument will be calculated efficiently according to the format of the data, regardless of the desired direction of iteration.

5 Visualization

As in previous versions of *Cardinal*, the `plot()` function is used to visualize mass spectra, and the `image()` function is used to visualize molecular ion images.

The primary differences between these methods when used on older classes and the new classes are:

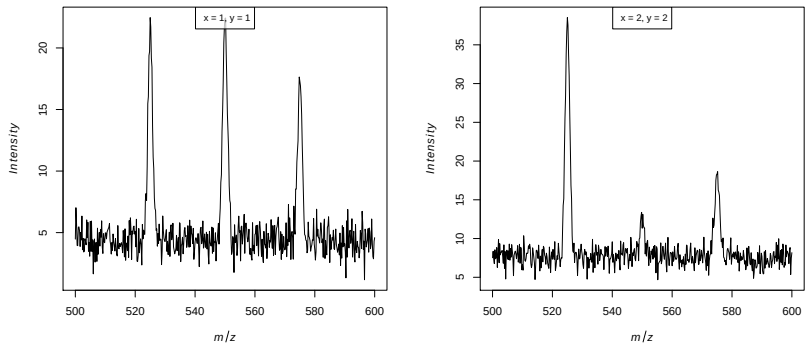
- A new default color scale for images that doesn't use the rainbow color scheme
- Non-gridded pixel coordinates are allowed to better allow for non-rastered image data
- The output visualization can be assigned to a variable for later re-plotting

5.1 Visualizing mass spectra with `plot()`

The `plot()` method is used for plotting mass spectra.

```
> plot(msdata, pixel=1)
```

```
> plot(msdata, coord=list(x=2, y=2))
```



(a) Plot of pixel $= 1$ **(b)** Plot of x $= 2$, y $= 2$

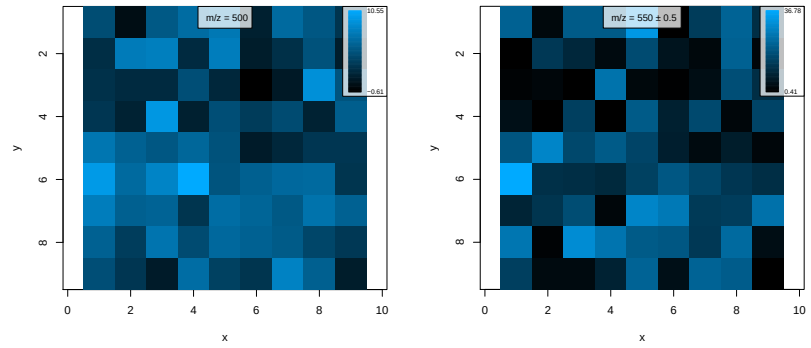
This section will be expanded in the future. See the original "Cardinal walkthrough" vignette for additional information on plotting options.

5.2 Visualizing molecular ion images with `image()`

The `image()` method is used for plotting molecular ion images.

```
> image(msdata, feature=1)
```

```
> image(msdata, mz=550, plusminus=0.5)
```



(a) Image of feature $= 1$ **(b)** Image of $m/z = 550$

This section will be expanded in the future. See the original "Cardinal walkthrough" vignette for additional information on plotting options.

5.3 Region-of-interest selection

The `selectROI()` function is used to interactively select regions-of-interest. See `?selectROI` for details.

6 Processing

The pre-processing workflow has been overhauled in *Cardinal* 2.0 to support more efficient processing of larger-than-memory datasets.

6.1 Queueing delayed processing with `process()`

The `process()` method allows queueing of delayed pre-processing steps to an imaging dataset. It expects a function that takes a vector (e.g., a mass spectra), processes it, and returns a vector of the same length as the original vector.

At its simplest, this can be used to immediately apply a transformation function to each spectrum in an MS imaging dataset.

```
> tmp <- process(msdata, function(x) x + 1, label="add1")
> tmp

An object of class 'MSImagingExperiment'
  <501 feature, 81 pixel> imaging dataset
    imageData(2): data0 data1
    featureData(1): fid
    pixelData(1): pid
    processing complete(1): add1
    processing pending(0):
    raster dimensions(2): x := 9, y := 9
    mass range: 500 to 600
    centroided: FALSE
```

By supplying the argument `delay=TRUE`, the transformation function is instead queued, but not applied. This can be used to add multiple transformation functions to the queue. These are unaffected by additional transformations, such as subsetting the dataset.

A call to `process()` without `delay=TRUE` will apply all of the queued processing functions to the dataset.

```
> tmp <- msdata %>%
+   process(function(x) ifelse(x > 0, x, 0), label="pos", delay=TRUE) %>%
+   process(function(x) x + 1, label="add1", delay=TRUE) %>%
+   process(log2, label="log2", delay=TRUE) %>%
+   select(x <= 4, y <= 4) %>%
+   filter(mz < 550)
> process(tmp, BPPARAM=SerialParam())

An object of class 'MSImagingExperiment'
  <250 feature, 16 pixel> imaging dataset
    imageData(2): data0 data1
```

```
    featureData(1): fid
    pixelData(1): pid
    processing complete(3): pos add1 log2
    processing pending(0):
    raster dimensions(2): x := 4, y := 4
    mass range: 500.0 to 549.8
    centroided: FALSE
```

Internally, `process()` applies the processing functions using `pixelApply()` or `featureApply()`, so it will be executed in parallel if a parallel backend is registered. Use the `BPPARAM` argument to specify another parallel backend.

6.2 Example processing workflow

When applied to new *Cardinal* classes such as `MSImagingExperiment`, processing methods such as `smoothSignal()`, `reduceBaseline()`, and `peakPick()` are queued to the dataset. They are applied the next time `process()` is called.

```
> tmp <- msdata %>%
+   smoothSignal() %>%
+   reduceBaseline() %>%
+   peakPick() %>%
+   peakFilter() %>%
+   select(x == 1, y == 1) %>%
+   process(plot=TRUE,
+     par=list(layout=c(1,3)),
+     BPPARAM=SerialParam())
```

Cardinal 2.0 implements the following pre-processing methods for `MSImagingExperiment`:

- `normalize()` performs normalization, including TIC normalization
- `smoothSignal()` performs smoothing, to reduce noise in the spectra
- `reduceBaseline()` performs baseline reduction
- `peakPick()` detects peaks
- `peakAlign()` aligns peaks to a set of reference peaks
- `peakFilter()` removes peaks according to criteria such as peak frequency
- `peakBin()` bins spectra to a set of reference peaks

This section will be expanded in the future. See the help pages of these functions for more details on their use.

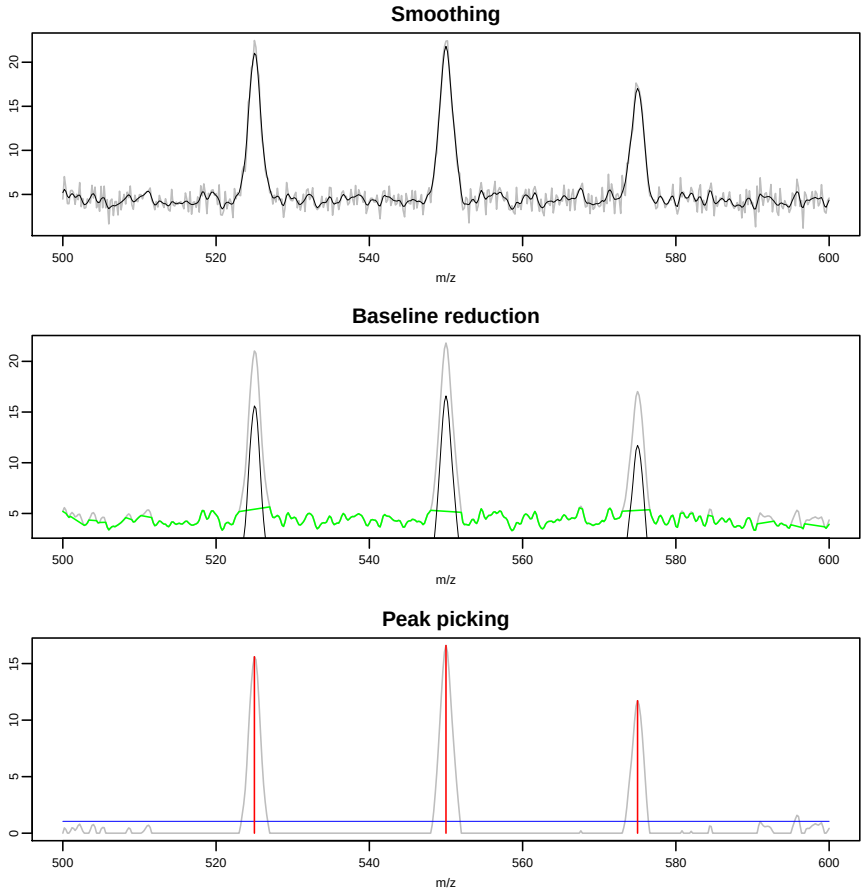


Figure 3: Example processing workflow

7 Analysis

Many of the analysis methods in *Cardinal* still need to be updated to work with the new classes. To continue using the analysis methods designed for older classes, an `MSImageExperiment` object can be coerced to a `MSImageSet` object using the `as()` method.

```
> msdata0b <- as(msdata0, "MSImageSet")
> msdata0b

An object of class "MSImageSet"
Slot "processingData":
Processing data
  Cardinal version: 2.0.4
  Files:
  Normalization:
  Smoothing:
  Baseline reduction:
  Spectrum representation:
  Peak picking:
```

```
Slot "experimentData":
Experiment data
  Experimenter name:
  Laboratory:
  Contact:
  Title:
  URL:
  PMIDs:
  No abstract available.

Slot "imageData":
An object of class 'MSImageData'
  iData: 501 x 81 matrix (0.3 Mb)

Slot "pixelData":
An object of class 'IAnnotatedDataFrame'
  pixelNames: x = 1, y = 1 x = 2, y = 1 ... x = 9, y = 9 (81 total)
  varLabels: x y sample pid
  varMetadata: labelType labelDescription

Slot "featureData":
An object of class 'AnnotatedDataFrame'
  featureNames: m/z = 500 m/z = 500.2 ... m/z = 600 (501 total)
  varLabels: mz fid
  varMetadata: labelDescription

Slot "protocolData":
An object of class 'AnnotatedDataFrame': none

Slot ".__classVersion__":
         R    Biobase       iSet  SImageSet MSImageSet
   "3.5.2"   "2.42.0"    "0.1.0"    "0.1.0"    "0.7.0"
```

`MSImageSet` objects can be coerced to the newer `MSImageExperiment` objects as well.

```
> msdata0c <- as(msdata0b, "MSImagingExperiment")
> msdata0c

An object of class 'MSContinuousImagingExperiment'
  <501 feature, 81 pixel> imaging dataset
    imageData(1): intensity
    featureData(1): fid
    pixelData(1): pid
    processing complete(0):
    processing pending(0):
    raster dimensions(2): x := 9, y := 9
    mass range: 500 to 600
    centroided: FALSE
```

See the vignettes from the *CardinalWorkflows* package for a more in-depth walkthrough of Cardinal's analytic methods on real experimental data.

8 Session info

- R version 3.5.2 (2018-12-20), `x86_64-w64-mingw32`
- Locale: `LC_COLLATE=C`, `LC_CTYPE=English_United States.1252`, `LC_MONETARY=English_United States.1252`, `LC_NUMERIC=C`, `LC_TIME=English_United States.1252`
- Running under: `Windows Server 2012 R2 x64 (build 9600)`
- Matrix products: default
- Base packages: base, datasets, grDevices, graphics, methods, parallel, stats, stats4, utils
- Other packages: BiocGenerics 0.28.0, BiocParallel 1.16.6, Cardinal 2.0.4, EBImage 4.24.0, ProtGenerics 1.14.0, S4Vectors 0.20.1
- Loaded via a namespace (and not attached): Biobase 2.42.0, BiocManager 1.30.4, BiocStyle 2.10.0, DBI 1.0.0, MASS 7.3-51.1, Matrix 1.2-15, R6 2.4.0, RCurl 1.95-4.11, Rcpp 1.0.0, abind 1.4-5, assertthat 0.2.0, biglm 0.9-1, bitops 1.0-6, compiler 3.5.2, crayon 1.3.4, digest 0.6.18, dplyr 0.8.0.1, evaluate 0.13, fftwtools 0.9-8, glue 1.3.0, grid 3.5.2, htmltools 0.3.6, htmlwidgets 1.3, irlba 2.3.3, jpeg 0.1-8, knitr 1.21, lattice 0.20-38, locfit 1.5-9.1, magrittr 1.5, matter 1.8.3, pillar 1.3.1, pkgconfig 2.0.2, png 0.1-7, purrr 0.3.0, rlang 0.3.1, rmarkdown 1.11, signal 0.7-6, sp 1.3-1, tibble 2.0.1, tidyselect 0.2.5, tiff 0.1-5, tools 3.5.2, xfun 0.5, yaml 2.2.0

References

[1] Thorsten Schramm, Alfons Hester, Ivo Klinkert, Jean-Pierre Both, Ron M. A. Heeren, Alain Brunelle, Olivier Laprévote, Nicolas Desbenoit, Marie-France Robbe, Markus Stoeckli, Bernhard Spengler, and Andreas Römpp. imzml – a common data format for the flexible exchange and processing of mass spectrometry imaging data. *Journal of Proteomics*, 75(16):5106 – 5110, 2012. Special Issue: Imaging Mass Spectrometry: A User's Guide to a New Technique for Biological and Biomedical Research. URL: http://www.sciencedirect.com/science/article/pii/S1874391912005568, doi:http://dx.doi.org/10.1016/j.jprot.2012.07.026.