Introduction to spectrolab

Jose Eduardo Meireles, Anna K. Schweiger and Jeannine Cavender-Bares

`spectrolab` provides methods to read, process and visualize data from portable spectrometers and establishes a common interface to spectra that other packages can build on. The package introduces a `spectra` S3 class and packs a ton of functionality:

- Read spectra from raw spectral files or matrices
- Access, aggregate, subset, split or combine spectra
- Seamlessly link and manipulate metadata (such as chemistry)
- Plot spectra or spectral quantiles, shade spectral regions (e.g. vis)
- Scroll through and zoom in spectra interactively.
- Perform tasks such as vector normalization, smoothing, resampling, and sensor overlap matching

The source code can be found on our GitHub repository. Please report any bugs and ask us your questions through the issue tracker.

0.1 Installing and loading `spectrolab`

The latest stable version of `spectrolab` is on CRAN. Install it with:

```
install.packages("spectrolab")
```

You can also install it directly from GitHub using the `devtools` package.

```
library("devtools")
install_github("meireles/spectrolab")
```

Assuming that everything went smoothly, you should be able to load `spectrolab` like any other package.

0.2 Reading spectra

There are two ways to get spectra into R: 1) converting a matrix or data.frame to `spectra` or 2) reading spectra from raw data files (formats: SVC's `sig`, Spectral Evolution's `sed` and ASD's `asd`). Here are a couple examples:

0.2.1 Create spectra from a matrix or data.frame

If you already have your spectra in a matrix or data frame (e.g. when you read your data from a .csv file), you can use the function `as_spectra()` to convert it to a `spectra` object. The matrix **must** have samples in rows and bands in columns. The header of the bands columns must be (numeric) band labels. You also should declare which column has the sample names (which are mandatory) using the `name_idx` argument. If

other columns are present (other than sample name and values), their indices must be passed to `as_spectra` as the `meta_idxs` argument.

Here is an example using a dataset matrix named `spec_matrix_meta.csv` provided by the package.

```
dir_path = system.file("extdata/spec_matrix_meta.csv", package = "spectrolab")

# Read data from the CSV file. If you don't use `check.names` = FALSE when reading
# the csv, R will usually add a letter to the column names (e.g. 'X650') which will
# cause problems when converting the matrix to spectra.
spec_csv = read.csv(dir_path, check.names = FALSE)

# The sample names are in column 3. Columns 1 and 2 are metadata
achillea_spec = as_spectra(spec_csv, name_idx = 3, meta_idxs = c(1,2) )

# And now you have a spectra object with sample names and metadata...
achillea_spec
```

```
## spectra object
## number of samples: 10
## bands: 400 to 2400 (2001 bands)
## metadata (2): ident, ssp
```

0.2.2 Reading spectra: example with SVC's `.sig` files

You should use the function `read_spectra()` to read raw spectra files. You can pass a vector of file names to `read_spectra()`, but it is usually easier to pass the path to the folder where your data are.

```
# `dir_path` is the directory where our example datasets live
dir_path = system.file("extdata", "Acer_example", package = "spectrolab")

# Read .sig files
acer_spectra = read_spectra(path = dir_path, format = "sig")
```

It may be the case that file names of undesirable spectra where flagged in the field. For example, we usually add the suffixes "_WR" to denote white reference and "_BAD" to denote bad measurements (you can pick whatever you want through). You can avoid importing such files by passing those flags to the argument `exclude_if_matches` in `read_spectra()`.

```
# use the `exclude_if_matches` argument to excluded flagged files
acer_spectra = read_spectra(path = dir_path, format = "sig", exclude_if_matches = c("BAD","WR"))
```

Flagging "unusual" measurements during data collection speeds up the dataset cleanup process. However, you can also exclude bad spectra and outliers by subsetting your spectra object, as shown in the section **Subsetting spectra** below.

0.3 Inspecting and querying spectra

You can check out your spectra object in several ways. For instance, You may want to know how many spectra and how many bands are in there, retrieve the file names, etc. Of course you will need to plot the data, but that topic gets its own section further down.

```r
# Simply print the object
acer_spectra
```

```
## spectra object
## number of samples: 7
## bands: 340.5 to 2522.8 (1024 bands, **overlap not matched**)
## metadata: none
```

```r
# Get the dataset's dimensions
dim(acer_spectra)
```

```
## n_samples   n_bands
##         7      1024
```

`spectrolab` also lets you access the individual components of the `spectra`. This is done with the functions `names()` for sample names, `bands()` for band labels, `value()` for the value matrix, and `meta()` for the associated metadata (in case you have any).

```r
# Vector of all sample names. Note: Duplicated sample names are permitted
n = names(achillea_spec)

# Vector of bands
w = bands(achillea_spec)

# value matrix
r = value(achillea_spec)

# Metadata. Use simplify = TRUE to get a vector instead of a data.frame
m = meta(achillea_spec, "ssp", simplify = TRUE)
```

0.4 Subsetting spectra

You can subset the `spectra` using a notation *similar* to the `[i, j]` function used in matrices and data.frames. The first argument in `[i, ]` matches *sample names*, whereas the second argument `[ , j]` matches the *band names*. Here are some examples of how `[` works in `spectra`:

- `x[1:3, ]` will keep the first three samples of `x`, i.e. `1:3` are indexes.
- `x["sp_1", ]` keeps **all** entries in `x` where sample names match `"sp_1"`
- `x[ , 800:900]` will keep bands labeled `800`, `801`, `802`, ..., `900`.
- `x[ , bands(x, 800, 900)]` will keep bands between `800` and `900`, including those with non-integer labels, e.g. `876.32`.
- `x[ , 1:5]` will **fail**!. *bands **cannot** be subset by indexes!*

Subsetting lets you, for instance, exclude noisy regions at the beginning and end of the spectrum or limit the data to specific entries.

```r
# Subset band regions. Here we know that bands are integers (e.g. 400, 401, ...)
spec_sub_vis = achillea_spec[ , 400:700 ]

# Subset spectra to all entries where sample_name matches "ACHMI_7" or
```

```r
# get the first three samples
spec_sub_byname = achillea_spec["ACHMI_7", ]
spec_sub_byidx  = achillea_spec[ 1:3, ]
```

The resolution of some spectra may be different from 1nm. In those cases, the best way to subset spectra is using the `min` and `max` arguments for `bands`:

```r
acer_spectra_trim = acer_spectra[ , bands(acer_spectra, 400, 2400) ]
```

Note that you can (1) subset samples using indexes and (2) use characters or numerics to subset bands. As said before, you cannot use indexes to subset bands though.

```r
# Subsetting samples by indexes works and so does subsetting bands by numerics or characters.
spec_sub_byidx[1, "405"] == spec_sub_byidx[1, 405]
```

```
## ACHMI_1
##    TRUE
```

But remember that you CANNOT use indexes to subset bands!

```r
# Something that is obvioulsy an index, like using 2 instead of 401 (the 2nd band
# in our case), will fail.
spec_sub_byidx[ , 2]
```

```
## Error: 1 labels not found: 2
```

```
`Error in i_match_ij_spectra(this = this, i = i, j = j) : band subscript out of bounds. Use band labels
```

```
## Error in eval(expr, envir, enclos): object 'Error in i_match_ij_spectra(this = this, i = i, j = j) :
```

0.5 Plotting

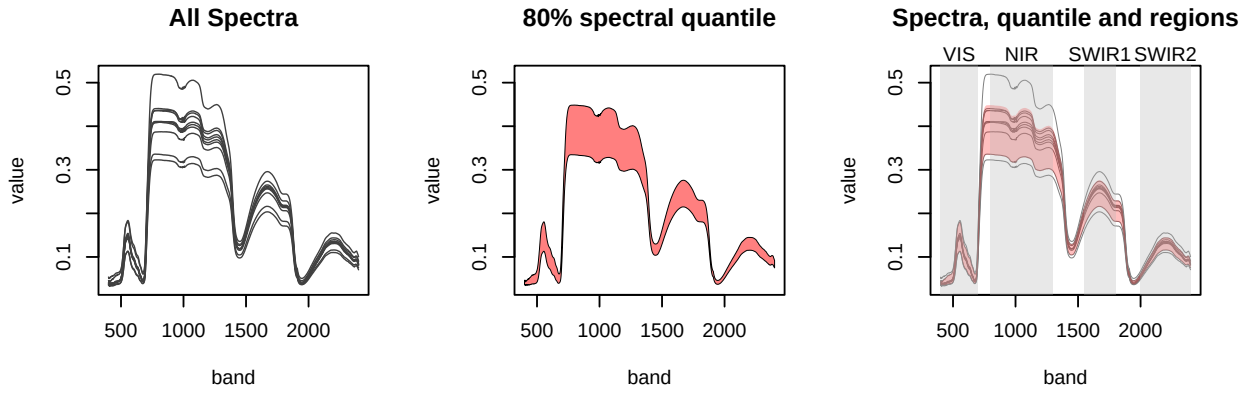
The workhorse function for statically plotting `spectra` is `plot()`. It will jointly plot each spectrum in the `spectra` object. You should be able to pass the usual plot arguments to it, such as `col`, `ylab`, `lwd`, etc.

You can also plot the quantiles of a `spectra` object with `plot_quantile()`. It's second argument, `total_prob`, is the total "mass" that the quantile encompasses. For instance, a `total_prob = 0.95` covers 95% of the variation in the `spectra` object, i.e. it is the `0.025 to 0.975` quantile. The quantile plot can stand alone or be added to a current plot if `add = TRUE`.

The function `plot_regions()` helps shading different spectral regions. `spectrolab` provides a `default_spec_regions()` matrix as an example, but you obviously can customize it for your needs (see the help page for `plot_regions` for details).

```r
# Simple spectra plot
par(mfrow = c(1, 3))
plot(achillea_spec, lwd = 0.75, lty = 1, col = "grey25", main = "All Spectra")

# Stand along quantile plot
plot_quantile(achillea_spec, total_prob = 0.8, col = rgb(1, 0, 0, 0.5), lwd = 0.5, border = TRUE)
```

```
title("80% spectral quantile")

# Combined individual spectra, quantiles and shade spectral regions
plot(achillea_spec, lwd = 0.25, lty = 1, col = "grey50", main="Spectra, quantile and regions")
plot_quantile(achillea_spec, total_prob = 0.8, col = rgb(1, 0, 0, 0.25), border = FALSE, add = TRUE)
plot_regions(achillea_spec, regions = default_spec_regions(), add = TRUE)
```



Last but not least, spectrolab also allows you to interactively explore spectra through a `shiny` app with the `plot_interactive()` function.

0.6 Manipulating samples names, band labels, metadata and value

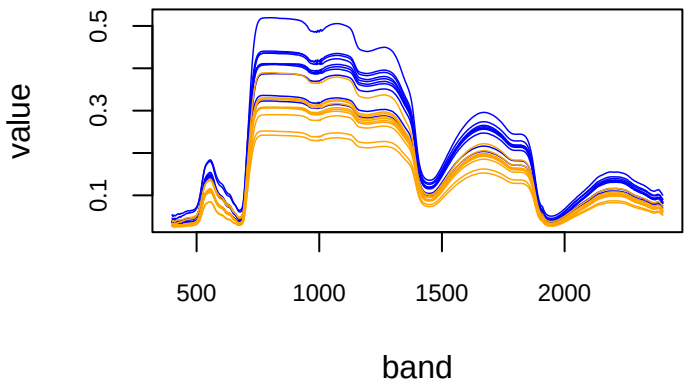
You may want to edit certain simple attributes of `spectra`, such as making all sample names lowercase This is easily attainable in `spectrolab`.

```
spec_new = achillea_spec

# Replace names with a lowercase version
names(spec_new) = tolower(names(achillea_spec))

# Check the results
names(spec_new)[1:5]
```

```
## [1] "achmi_1" "achmi_2" "achmi_3" "achmi_4" "achmi_5"
```

If you want to fiddle with the value itself, this is easy, too.

```
# Scale value by 0.75
spec_new = spec_new * 0.75

# Plot the results
plot(achillea_spec, col = "blue", lwd = 0.75, cex.axis = 0.75)
plot(spec_new, col = "orange", lwd = 0.75, add = TRUE)
```



Or you can also edit or add new metadata to the `spectra` object.

```
## Adding metadata to a spectra object: a dummy N content
n_content = rnorm(n = nrow(achillea_spec), mean = 2, sd = 0.5)
meta(achillea_spec, label = "N_percent") = n_content
```

0.6.1 Converting a spectra object into a matrix or data.frame

It is also possible to convert a `spectra` object to a matrix or data.frame using the `as.matrix()` or `as.data.frame()` functions. This is useful if you want to export your data in a particular format, such as csv.

If you're converting spectra to a matrix, `spectrolab` will (1) place bands in columns, assigning band labels to `colnames`, and (2) samples in rows, assigning sample names to `rownames`. Since `R` imposes strict rules on column name formats and sometimes on row names, `as.matrix()` will try to fix potential dimname issues if `fix_names != "none"`. Note that `as.matrix()` will not keep metadata.

Conversion to data.frame is similar, but keeps the metadata by default (unless you set the `metadata` argument to `FALSE`).

```
# Make a matrix from a `spectra` object
spec_as_mat = as.matrix(achillea_spec, fix_names = "none")
spec_as_mat[1:4, 1:3]
```

```
##                400        401        402
## ACHMI_1 0.03734791 0.03698631 0.03804012
## ACHMI_2 0.04608409 0.04536371 0.04436544
## ACHMI_3 0.04058113 0.04025678 0.03958125
## ACHMI_4 0.04063730 0.03993420 0.03793455
```

```
# Make a matrix from a `spectra` object
spec_as_df = as.data.frame(achillea_spec, fix_names = "none", metadata = TRUE)
spec_as_df[1:4, 1:5]
```

```
##   sample_name ident                   ssp N_percent        400
## 1     ACHMI_1 10526 Achillea millefolium   1.689960 0.03734791
## 2     ACHMI_2 10527 Achillea millefolium   1.634288 0.04608409
## 3     ACHMI_3 10528 Achillea millefolium   2.256354 0.04058113
## 4     ACHMI_4 10529 Achillea millefolium   2.472812 0.04063730
```