photobiology Version 0.5.13
User Guide

Pedro J. Aphalo

February 17, 2015

Contents

1 Introduction

We have developed a set of packages to facilitate the calculation of many different quantities that can be derived from spectral irradiance data. The base package in this suite is called photobiology, and is the package described here. There other specialized packages for quantification of ultraviolet radiation and visible radiation (photobiologyWavebands), or based on Phytochrome (photobiologyPhy), Cryptochrome (photobiologyCry) (both photoreceptors present in plants). Other packages in the suite provide example spectral data for filters (photobiologyFilters), lamps (photobiologyLamps), LEDs (photobiologyLEDs), sunlight (photobiologySun) and broadband sensors (photobiologySensors) . In the future it will be submitted to CRAN (Comprehensive R archive network), it is meanwhile available from `https://www.r4photobiology.info/`. There is also a public Git repository at `https://bitbucket.org/aphalo/` from where the source code of the current an earlier versions can be cloned.

2 Installation and use

The functions in the package photobiology are made available by installing the packages `photobiology` (once) and loading it from the library when needed.

To load the package into the workspace we use `library(photobiology)`.

```
library(photobiology)
library(photobiologygg)
library(photobiologyWavebands)
library(photobiologySun)
library(photobiologyFilters)
library(photobiologySensors)
library(ggplot2)
library(ggtern)
```

3 Spectral data and their handling

The package provides two sets of functions for many operations: functions programmed following a functional paradigm, and functions using an object-oriented paradigm. The former functions take as arguments numeric vectors and are probably faster. The later ones take 'spectra' objects as arguments, are easier to use, and at least at the moment, to some extent slower. For every-day use 'spectra' objects are recommended, but when maximum performance or flexibility in scripts is desired, the use of the functions taking numeric vectors as arguments may allow optimizations that are not possible with the object-oriented higher level functions.

3.1 Example spectral data

A few data objects are included in the package for use in examples. Two simulated solar spectra, `sun.spct` and `sun.daily.spct`, are include as `source.spct` objects. The same data are also included as data.frame objects. Chromaticity data for humans are also included...

3.2 Using numeric vectors

When dealing with spectra, we operate on pairs of vectors, one with the wavelengths in nm, and a second one with the corresponding values for the spectral data.

It is usual to 'group' these two (or more) vectors into a data frame. For vectors within a data frame we need to 'make them visible' when operating with the functions that expect numeric vectors as arguments.

3.3 Using 'spectra' objects

This package defines a family of objects based on data tables (data frame compatible objects) which impose some restrictions on the naming of the vectors, something that allows the second set of functions to 'find' the data when passed one of these objects as argument. In addition, as the data is checked when the object is built, there is no need to test for the validity of the data each time a calculation is carried out. The other advantage of using `spct` objects, is that specialized versions of generic functions like `print` and operators like `+` can be

defined for spectra. `___.spct` objects are derived from data.table objects, which in turn are derived from data.frame objects. In this package we define a *generic* spectrum type of object, derived from data table, from which specialized types of spectra are derived. This 'parenthood' hierarchy means that spectra objects can be used almost anywhere where a `data.frame` or `data.table` is expected. Many functions defined in package `data.table` are useful when working with spectra.

Although `data.tables` are syntactically compatible with `data.frames`, in some special cases the same code may have different semantics as data tables use references in some cases were data frames would use a copy of the data. In general, no such problems exist, and the different semantics only applies to data table specific syntax. If in doubt, to avoid problems, when you really intend to make a new copy of a spectrum, preserving the original object unchanged by later operations on the new 'name', use function `copy` in addition to the assignment operator.

```
# 1) data frame syntax on a data.frame
a.df <- data.frame(x = 1:3, y = rep(1, 3))
b.df <- a.df
b.df$y <- b.df$y * 2
b.df

##   x y
## 1 1 2
## 2 2 2
## 3 3 2

a.df # not modified!

##   x y
## 1 1 1
## 2 2 1
## 3 3 1

# 2) data frame syntax on a data.table
a.dt <- data.table(x = 1:3, y = rep(1, 3))
b.dt <- a.dt
b.dt$y <- b.dt$y * 2
b.dt

##    x y
## 1: 1 2
## 2: 2 2
## 3: 3 2

a.dt # not modified!

##    x y
## 1: 1 1
## 2: 2 1
## 3: 3 1

# 3) data table syntax on a data.table
a.dt <- data.table(x = 1:3, y = rep(1, 3))
b.dt <- a.dt
b.dt[ , y := y * 2]
```

```
##    x y
## 1: 1 2
## 2: 2 2
## 3: 3 2

a.dt # modified!

##    x y
## 1: 1 2
## 2: 2 2
## 3: 3 2

# 4) forcing creation of a copy
a.dt <- data.table(x = 1:3, y = rep(1, 3))
c.dt <- copy(a.dt)
c.dt[ , y := y * 2]

##    x y
## 1: 1 2
## 2: 2 2
## 3: 3 2

a.dt # not modified!

##    x y
## 1: 1 1
## 2: 2 1
## 3: 3 1
```

From the examples above one can see that in example 3) `b.dt` is not a copy of `a.dt`, but instead a reference (a new name pointing to the original object), while in examples 1), 2) and 4) `b.dt`, is a new object, initialized to the value of `a.dt`.

Spectral objects are printed in the current version of the package by the function defined in package `data.frame`, consequently, it is possible to use options from this package to control printing. The first option set below, `datatable.print.nrows`, determines the number of rows above which only 'head' and 'tail' rows are printed. The second option, `datatable.print.topn`, determines how many rows are printed when not all rows are printed.

```
options(datatable.print.nrows = 10)
options(datatable.print.topn = 2)
```

The number of rows printed can be also controlled through an explicit argument to the second parameter of `print`, `head`, and `tail`. Setting an option by means of `options` changes the default behaviour of `print`, but explicit arguments can still be used for changing this behaviour in an individual statement. The statement `a.dt` implicitly calls `print` when using R in interactive mode.

```
a.dt

##    x y
```

```
## 1: 1 1
## 2: 2 1
## 3: 3 1

print(a.dt)

##    x y
## 1: 1 1
## 2: 2 1
## 3: 3 1

print(a.dt, 1L)

##     x y
##  1: 1 1
## ---
##  3: 3 1

head(a.dt, 2L)

##    x y
## 1: 1 1
## 2: 2 1

tail(a.dt, 2L)

##    x y
## 1: 2 1
## 2: 3 1
```

3.4 Spectral data assumptions

An assumption of the package is that wavelengths are always expressed in nanometres (1 nm $= 1 \cdot 10^{-9}$ m). If the data to be analysed uses different units for wavelengths, e.g. Ångstrom (1 Å $= 1 \cdot 10^{-10}$ m), the values need to be re-scaled before any calculations.

Energy irradiances are assumed to be expressed in W m^{-2} and photon irradiances in mol m^{-2} s^{-1}, that is to say using second as unit for time. This is the default, but it is possible to set the unit for time to day in the case of `source.spct` objects.

The default time unit used is *second*, but *day* can be used by supplying the argument `"day"` to a parameter of the constructor of `source.spct` objects.

> Not respecting these assumptions will yield completely wrong results! It is extremely important to make sure that the wavelengths are in nanometres as this is what all functions expect. If wavelength values are in the wrong units, the action-spectra weights and quantum conversions will be wrongly calculated, and the values returned by most functions completely wrong, without warning.

If spectral irradiance data is in $\mathrm{W\,m^{-2}\,nm^{-1}}$, and the wavelength in nm, as is the case for many Macam spectroradiometers, the data can be used directly and functions in the package will return irradiances in $\mathrm{W\,m^{-2}}$.

If, for example, the spectral irradiance data output by a spectroradiometer is expressed in $\mathrm{mW\,m^{-2}\,nm^{-1}}$, and the wavelengths are in Ångstrom then to obtain correct results when using any of the packages in the suite, we need to rescale the data.

```
# not run
energy_irradiance(wavelength/10, irrad/1000)
```

In the example above, we take advantage of the behaviour of the S language: an operation between a scalar and vector, is equivalent to applying this operation to each member of the vector. Consequently, in the code above, each value from the vector of wavelengths is divided by 10, and each value in the vector of spectral irradiances is divided by 1000.

3.5 Spectral objects

There are basically three different approaches to the creation of spectra. The first approach consist in setting the class attribute of an existing data frame or data table, in simple terms, converting an existing object into a spectral object. This approach avoids creating a copy of the data, and should be fastest. The second approach is to use an 'as' function to create a new spectral object from a data frame or data table (the original object remains unchanged, and independent of the spectral object). The third approach is to use a function with the same name as the spectrum object class, and supply the data as numeric vector arguments. With the first two approaches the variables should be suitably named so that they can be recognized, in the third approach the formal argument to which the actual argument vector is supplied determines how it is interpreted.

3.5.1 Querying the class of a spectrum object

Consistently with R design, the package provides 'is' functions for querying the type of spectra objects.

```
is.source.spct(sun.spct)

## [1] TRUE

is.filter.spct(sun.spct)

## [1] FALSE

is.any.spct(sun.spct)

## [1] TRUE
```

In addition function `class.spc` returns directly the spectrum-related class attributes.

```
class.spct(sun.spct)

## [1] "source.spct"  "generic.spct"

class.spct(1:10)

## character(0)
```

The built-in R function `class` returns all class attributes of an R object.

```
class(sun.spct)

## [1] "source.spct"  "generic.spct" "data.table"   "data.frame"

class(1:10)

## [1] "integer"
```

3.5.2 Special attributes

`source.spct` objects have a `time.unit` attribute which can take one of two values `"second"` or `"day"`, the default is `"second"`. However, if the spectral data is for daily exposure, then the attribute should be set when the object is constructed. It is also possible to set the attribute for an existing object with function `setTimeUnit`.

`filter.spct` objects have a `Tfr.type` attribute which can take one of two values `"total"` or `"internal"`, the default being `"total"`. However, if the spectral transmittance or absorbance data is internal, meaning excluding the contribution of reflection, then the attribute should be set when the object is constructed. It is also possible to set the attribute for an existing object with function `setTfrType`.

> Spectral objects created with earlier versions of this package are missing these attributes. For this reason 'summary' and 'plot' functions may not work as expected. The objects can be updated by adding the missing attribute using the functions `setTimeUnit` and `setTfrType`.

3.5.3 Setting the class of a spectrum object

`generic.spct` objects can be created from data tables and data frames simply by setting them as such. However, a column called `w.length` must be present and contain wavelength values expressed in nm. Functions with names of the form `is.____.spct` are defined for all classes of spectra and can take as arguments any R object. In addition function `is.any.spct` can use to query if an

R object inherits from any of the classes of spectra defined in this package. Finally function `class.spct` works similarly to R's `class` functions but returns a vector containing only the names of spectra classes. The 'set' functions keep unrecognised variables, and fill missing required variables with `NA`, except for `w.length`, which if missing triggers an error.

We create a data.table object `a.spct`, and query its class.

```
a.spct <- data.table(w.length = 300:305, y = 1)
class(a.spct)

## [1] "data.table" "data.frame"

class.spct(a.spct)

## character(0)

is.any.spct(a.spct)

## [1] FALSE
```

We convert `a.spct` into a `generic.spct` object, and query its class.

```
setGenericSpct(a.spct)
class(a.spct)

## [1] "generic.spct" "data.table"   "data.frame"

class.spct(a.spct)

## [1] "generic.spct"

is.generic.spct(a.spct)

## [1] TRUE

a.spct

##    w.length y
## 1:      300 1
## 2:      301 1
## 3:      302 1
## 4:      303 1
## 5:      304 1
## 6:      305 1
```

`source.spct` objects can be created from data tables, data frames, and `generic.spct` simply by setting them as such. However, columns called `w.length` (wavelength values expressed in nm) and `s.e.irrad` ($\mathrm{W\,m^{-2}\,nm^{-1}}$) or `s.q.irrad` ($\mathrm{mol\,m^{-2}\,s^{-1}\,nm^{-1}}$) must be present.

```
b.spct <- setSourceSpct(data.table(w.length = 300:305, s.e.irrad = 1))
attr(b.spct, "time.unit")

## [1] "second"
```

```
class(b.spct)

## [1] "source.spct"  "generic.spct" "data.table"   "data.frame"

b.spct

##    w.length s.e.irrad
## 1:      300         1
## 2:      301         1
## 3:      302         1
## 4:      303         1
## 5:      304         1
## 6:      305         1
```

If the spectral irradiance is expressed per day, then the parameter `time.unit` should be set to `"day"` instead of the default of `"second"`. This information is used when printing and plotting source spectra.

```
b.d.spct <- setSourceSpct(
  data.table(w.length = 300:305, s.e.irrad = rep(1,6)),
  time.unit = "day")
attr(b.d.spct, "time.unit")

## [1] "day"

class(b.d.spct)

## [1] "source.spct"  "generic.spct" "data.table"   "data.frame"

b.d.spct

##    w.length s.e.irrad
## 1:      300         1
## 2:      301         1
## 3:      302         1
## 4:      303         1
## 5:      304         1
## 6:      305         1
```

`filter.spct` objects can be created from data tables, data frames, and `generic.spct` simply by setting them as such. However, columns called `w.length` (wavelength values expressed in nm) and `Tpc` (T%), `Tfr` (T as fraction of 1) and\or `A` (absorbance ($\log_{10}$ based)) must be present.

```
c.spct <- setFilterSpct(data.table(w.length = 300:305, Tfr = 1))
attr(c.spct, "Tfr.type")

## [1] "total"

class(c.spct)

## [1] "filter.spct"  "generic.spct" "data.table"   "data.frame"

c.spct
```

```
##    w.length Tfr
## 1:      300   1
## 2:      301   1
## 3:      302   1
## 4:      303   1
## 5:      304   1
## 6:      305   1
```

If the spectral transmittance or absorbance is the internal component, then the parameter `Tfr.type` should be set to `"internal"` instead of the default of `"total"`. This information is used when printing and plotting source spectra.

```
c.i.spct <- setFilterSpct(data.table(w.length = 300:305, Tfr = 1), "internal")
attr(c.i.spct, "Tfr.type")

## [1] "internal"

class(c.i.spct)

## [1] "filter.spct"  "generic.spct" "data.table"   "data.frame"

c.i.spct

##    w.length Tfr
## 1:      300   1
## 2:      301   1
## 3:      302   1
## 4:      303   1
## 5:      304   1
## 6:      305   1
```

`reflector.spct` objects can be created from data tables, data frames, and `generic.spct` simply by setting them as such. However, columns called `w.length` (wavelength values expressed in nm) and `Rpc` (R%), and\or `Rfr` (R as fraction of 1) must be present.

```
d.spct <- setReflectorSpct(data.table(w.length = 300:305, Rfr = 1))
class(d.spct)

## [1] "reflector.spct" "generic.spct"   "data.table"     "data.frame"

d.spct

##    w.length Rfr
## 1:      300   1
## 2:      301   1
## 3:      302   1
## 4:      303   1
## 5:      304   1
## 6:      305   1
```

`chroma.spct` objects can be created from data tables, data frames, and `generic.spct` simply by setting them as such. However, columns called `w.length` (wavelength values expressed in nm) and `x`, `y` and `z` must be present, giving the trichromic chromaticity constants.

```
e.spct <- setChromaSpct(data.table(w.length = 300:305, x = 1, y = 1, z = 1))
class(e.spct)

## [1] "chroma.spct"  "generic.spct" "data.table"   "data.frame"

e.spct

##    w.length x y z
## 1:      300 1 1 1
## 2:      301 1 1 1
## 3:      302 1 1 1
## 4:      303 1 1 1
## 5:      304 1 1 1
## 6:      305 1 1 1
```

In all cases if the expected data is not available, then it is filled-in if possible with values.

```
f.spct <- setReflectorSpct(data.table(w.length = 300:305, Rpc = 100))
class(f.spct)

## [1] "reflector.spct" "generic.spct"   "data.table"     "data.frame"

f.spct

##    w.length Rfr
## 1:      300   1
## 2:      301   1
## 3:      302   1
## 4:      303   1
## 5:      304   1
## 6:      305   1
```

When required data is not available, and it cannot be calculated from other columns, the required column is added and filled with `NA`s.

```
g.spct <- setReflectorSpct(data.table(w.length = 300:305, z = 1))

## Warning in check.reflector.spct(x, strict.range = strict.range):  No
reflectance data found in reflector.spct

class(g.spct)

## [1] "reflector.spct" "generic.spct"   "data.table"     "data.frame"

g.spct

##    w.length z Rfr
## 1:      300 1  NA
## 2:      301 1  NA
## 3:      302 1  NA
## 4:      303 1  NA
## 5:      304 1  NA
## 6:      305 1  NA
```

If no variable named `w.length` is present, and a variable named `wl` is found, it is renamed to `w.length`.

```
h.spct <- setReflectorSpct(data.table(wl = 300:305, Rfr = 1))
class(h.spct)

## [1] "reflector.spct" "generic.spct"   "data.table"     "data.frame"

h.spct

##    w.length Rfr
## 1:      300   1
## 2:      301   1
## 3:      302   1
## 4:      303   1
## 5:      304   1
## 6:      305   1
```

The range of the input is checked, and warnings or errors issued. The wavelength range test cannot be overriden as the most likely reason for it to be triggered is the expression of wavelengths in units other than the expected nanometres (nm).

```
(try(h1.spct <- setReflectorSpct(data.table(wl = 5999:6001, Rfr = 1))))

## [1] "Error in check.generic.spct(x) : \n  Off-range w.length values [5999...6001] instead of within 100 nm and
## attr(,"class")
## [1] "try-error"
## attr(,"condition")
## <simpleError in check.generic.spct(x): Off-range w.length values [5999...6001] instead of within 100 nm and 60
```

Transmittance and reflectance values are checked to be within their valid range, and spectral irradiance values to be zero or positive. By default failure of this test generates a fatal error.

```
(try(h2.spct <- setReflectorSpct(data.table(wl = 300:305, Rfr = -1))))

## [1] "Error in range_check(x, strict.range = strict.range) : \n  Off-range reflectance values [-1...-1] instead
## attr(,"class")
## [1] "try-error"
## attr(,"condition")
## <simpleError in range_check(x, strict.range = strict.range): Off-range reflectance values [-1...-1] instead of
```

The constructor converts percents to fractions before applying the tests.

```
(try(h3.spct <- setReflectorSpct(data.table(wl = 300:305, Rpc = -100))))

## [1] "Error in range_check(x, strict.range = strict.range) : \n  Off-range reflectance values [-1...-1] instead
## attr(,"class")
## [1] "try-error"
## attr(,"condition")
## <simpleError in range_check(x, strict.range = strict.range): Off-range reflectance values [-1...-1] instead of
```

Setting `strict.range = FALSE` issues a warning intead of an error, and returns an `.spct` object possibly containing the *bad data*.

```
h4.spct <- setReflectorSpct(data.table(wl = 300:305, Rfr = -1),
                            strict.range = FALSE)

## Warning in range_check(x, strict.range = strict.range): Off-range reflectance
values [-1...-1] instead of [0..1]

class(h4.spct)

## [1] "reflector.spct" "generic.spct"   "data.table"     "data.frame"

h4.spct

##    w.length Rfr
## 1:      300  -1
## 2:      301  -1
## 3:      302  -1
## 4:      303  -1
## 5:      304  -1
## 6:      305  -1
```

Setting `strict.range = NULL` skips tests, and silently returns an `.spct` object possibly containing the *bad data*.

```
h5.spct <- setReflectorSpct(data.table(wl = 300:305, Rfr = -1),
                            strict.range = NULL)
class(h5.spct)

## [1] "reflector.spct" "generic.spct"   "data.table"     "data.frame"

h5.spct

##    w.length Rfr
## 1:      300  -1
## 2:      301  -1
## 3:      302  -1
## 4:      303  -1
## 5:      304  -1
## 6:      305  -1
```

The constructor converts percents to fractions before applying the tests.

```
h6.spct <- setReflectorSpct(data.table(wl = 300:305, Rpc = -100),
                            strict.range = FALSE)

## Warning in range_check(x, strict.range = strict.range): Off-range reflectance
values [-1...-1] instead of [0..1]

class(h6.spct)

## [1] "reflector.spct" "generic.spct"   "data.table"     "data.frame"

h6.spct

##    w.length Rfr
## 1:      300  -1
## 2:      301  -1
```

```
## 3:      302  -1
## 4:      303  -1
## 5:      304  -1
## 6:      305  -1
```

A very frequent source of off-range values is measurement noise. This noise should be preferably explicitly dealt with before any further calculations. Normal R and/or `data.table` syntax can be used resolve these problems by smoothing or replacement of the problem data. If the 'bad' data is outside the range of interest it can be trimmed by means of function `trim_spct` after creating a spectrum with setting `strict.range = FALSE` or setting `strict.range = NULL`

We next use the example solar spectral irradiance data included in the package.

```
class(sun.spct)

## [1] "source.spct"  "generic.spct" "data.table"   "data.frame"

sun.spct

##      w.length    s.e.irrad    s.q.irrad
##   1:      293 2.609665e-06 6.391730e-12
##   2:      294 6.142401e-06 1.509564e-11
##  ---
## 507:      799 4.185850e-01 2.795738e-06
## 508:      800 4.069055e-01 2.721132e-06
```

We can set a spectrum object to a different type of spectrum, but as above this can result in NAs in case of missing data. We need to make a copy of `sun.spct`, because being part of the package, it is protected and should not be modified by user code.

```
i.spct <- copy(sun.spct)
setGenericSpct(i.spct)
class(i.spct)

## [1] "generic.spct" "data.table"   "data.frame"
```

3.5.4 Using 'as' functions

Here we briefly describe the 'as' functions, as what has been discussed above for 'set' functions also applies to 'as' functions, as they simply make a copy of their argument before calling the set functions, and then return this new object.

We can make a 'generic.spct' copy of any spectrum object.

```
j.spct <- as.generic.spct(sun.spct)
class(j.spct)

## [1] "generic.spct" "data.table"   "data.frame"

class(sun.spct)

## [1] "source.spct"  "generic.spct" "data.table"   "data.frame"
```

Or of a data frame.

```
k.df <- data.frame(wl = 400:410, anything = 1)
k.spct <- as.generic.spct(k.df)
class(k.spct)

## [1] "generic.spct" "data.table"   "data.frame"

class(k.df)

## [1] "data.frame"
```

Both `as.source.spct` and `as.filter.spct` accept an argument for setting the `time.unit` and `Tfr.type` attributes, respectively, with the same defaults as described above.

3.5.5 Using constructors

This approach is similar to using function `data.frame` to create a data frame, but in this case the names of the arguments are meaningful and convey information on the nature of the spectral data and basis of expression. In the examples below we supply a single value for the spectral data. This value gets recycled as is normal in R, but of course in real use it is more usual to supply a vector of the same length as the `w.length` vector.

```
s.spct <- source.spct(w.length = 300:305, s.e.irrad = 100)
class.spct(s.spct)

## [1] "source.spct"  "generic.spct"

s.spct

##    w.length s.e.irrad
## 1:      300       100
## 2:      301       100
## 3:      302       100
## 4:      303       100
## 5:      304       100
## 6:      305       100
```

```
s.spct <- source.spct(w.length = 300:305, s.q.irrad = 40, time.unit = "day")
class.spct(s.spct)
```

```
## [1] "source.spct"  "generic.spct"

s.spct

##    w.length s.q.irrad
## 1:      300        40
## 2:      301        40
## 3:      302        40
## 4:      303        40
## 5:      304        40
## 6:      305        40
```

```
l.spct <- filter.spct(w.length = 300:305, Tpc = 100)
class.spct(l.spct)

## [1] "filter.spct"  "generic.spct"

l.spct

##    w.length Tfr
## 1:      300   1
## 2:      301   1
## 3:      302   1
## 4:      303   1
## 5:      304   1
## 6:      305   1
```

```
l.spct <- filter.spct(w.length = 300:305, A = 2)
class.spct(l.spct)

## [1] "filter.spct"  "generic.spct"

l.spct

##    w.length A
## 1:      300 2
## 2:      301 2
## 3:      302 2
## 4:      303 2
## 5:      304 2
## 6:      305 2
```

```
wl1 <- 300:305
m.spct <- reflector.spct(w.length = wl1, Rfr = 0.5)
class.spct(m.spct)

## [1] "reflector.spct" "generic.spct"

m.spct

##    w.length Rfr
## 1:      300 0.5
## 2:      301 0.5
```

```
## 3:      302 0.5
## 4:      303 0.5
## 5:      304 0.5
## 6:      305 0.5
```

```
l.spct <- response.spct(w.length = 300:305, s.e.response = 0.5)
class.spct(l.spct)

## [1] "response.spct" "generic.spct"

l.spct

##    w.length s.e.response
## 1:      300          0.5
## 2:      301          0.5
## 3:      302          0.5
## 4:      303          0.5
## 5:      304          0.5
## 6:      305          0.5
```

Both `source.spct` and `filter.spct` functions accept an argument for setting the `time.unit` and `Tfr.type` attributes, respectively, with the same defaults as described above. Functions `source.spct`, `filter.spct` and `reflector.spct` have a `strict.range` formal argument, that alters checks as described in section 3.5.3 above.

3.5.6 Row binding spectra

Sometimes, especially for plotting, we may want to row-bind spectra. When the aim is that the returned object retains its class attributes, and other spectrum related attributes like the time unit, functions `rbind` from base R, and its reimplementation from package `data.table`, and function `rbindlist` also defined in package `data.table` should NOT be used. Package photobiology provides function `rbinspct` for row-binding spectra, with the necessary checks for consistency of the bound spectra.

```
# STOPGAP
shade.spct <- copy(sun.spct)
```

```
rbindspct(list(sun.spct, shade.spct))

##       w.length    s.e.irrad    s.q.irrad
##    1:      293 2.609665e-06 6.391730e-12
##    2:      293 2.609665e-06 6.391730e-12
##   ---
## 1015:      800 4.069055e-01 2.721132e-06
## 1016:      800 4.069055e-01 2.721132e-06
```

It is also possible to add an ID factor, to be able to still recognize the origin of the observations after the binding. If the supplied list is anonymous, then capital letters will be used for levels.

```
rbindspct(list(sun.spct, shade.spct), idfactor = TRUE)

##       w.length    s.e.irrad spct.idx    s.q.irrad
##    1:      293 2.609665e-06        A 6.391730e-12
##    2:      294 6.142401e-06        A 1.509564e-11
##   ---
## 1015:      799 4.185850e-01        B 2.795738e-06
## 1016:      800 4.069055e-01        B 2.721132e-06
```

In contrast, if a named list with no missing names, is supplied as argument, these names are used for the levels of the ID factor.

```
rbindspct(list(sun = sun.spct, shade = shade.spct), idfactor = TRUE)

##       w.length    s.e.irrad spct.idx    s.q.irrad
##    1:      293 2.609665e-06    shade 6.391730e-12
##    2:      294 6.142401e-06    shade 1.509564e-11
##   ---
## 1015:      799 4.185850e-01      sun 2.795738e-06
## 1016:      800 4.069055e-01      sun 2.721132e-06
```

If a character string is supplied as argument, then this will be used as the name of the factor.

```
rbindspct(list(sun = sun.spct, shade = shade.spct), idfactor = "ID")

##       w.length    s.e.irrad    ID    s.q.irrad
##    1:      293 2.609665e-06 shade 6.391730e-12
##    2:      294 6.142401e-06 shade 1.509564e-11
##   ---
## 1015:      799 4.185850e-01   sun 2.795738e-06
## 1016:      800 4.069055e-01   sun 2.721132e-06
```

3.6 Conversions between radiation units

The functions `e2q` and `q2e` can be used on source spectra to convert spectral energy irradiance into spectral photon irradiance and vice versa. The first argument should be a spectrum, and the second optional argument sets the action with `"add"` and `"replace"` as possible values. In the second case the whole spectrum object is copied, while in the first case a column is added but the unchanged columns are references to the original ones, rather than copies.

```
b.spct

##    w.length s.e.irrad
## 1:      300         1
## 2:      301         1
## 3:      302         1
## 4:      303         1
## 5:      304         1
## 6:      305         1
```

```
b1.spct <- e2q(b.spct, "replace")
b.spct
```

```
##    w.length s.e.irrad
## 1:      300         1
## 2:      301         1
## 3:      302         1
## 4:      303         1
## 5:      304         1
## 6:      305         1
```

```
b1.spct
```

```
##    w.length    s.q.irrad
## 1:      300 2.507767e-06
## 2:      301 2.516127e-06
## 3:      302 2.524486e-06
## 4:      303 2.532845e-06
## 5:      304 2.541204e-06
## 6:      305 2.549564e-06
```

```
b2.spct <- e2q(b.spct, "add")
b.spct
```

```
##    w.length s.e.irrad
## 1:      300         1
## 2:      301         1
## 3:      302         1
## 4:      303         1
## 5:      304         1
## 6:      305         1
```

```
b2.spct
```

```
##    w.length s.e.irrad    s.q.irrad
## 1:      300         1 2.507767e-06
## 2:      301         1 2.516127e-06
## 3:      302         1 2.524486e-06
## 4:      303         1 2.532845e-06
## 5:      304         1 2.541204e-06
## 6:      305         1 2.549564e-06
```

For `filter.spct` objects functions `T2A` and `A2T` allow conversion between spectral transmittance and spectral absorbance and vice versa.

3.7 Remapping a spectrum to different wavelengths

Converting spectra available at a given set of wavelengths values to a different one, is frequently needed when operating with several spectra of different origin. One can increase the *apparent* resolution by interpolation, and reduce it by local averaging or smoothing and resampling. The same function works on all `spct` objects, interpolating every column except `w.length` and replacing in this last column the old wavelength values with the new ones supplied as argument. The optional argument `fill.value` control what value is assigned to wavelengths in

the new data that are outside the range of the old wavelengths.

```
interpolate_spct(sun.spct, seq(400, 500, by = 0.1))
```

```
##       w.length s.e.irrad    s.q.irrad
##    1:    400.0 0.6081049 2.033314e-06
##    2:    400.1 0.6099118 2.039879e-06
##   ---
## 1000:    499.9 0.7241901 3.026228e-06
## 1001:    500.0 0.7240982 3.026450e-06
```

3.8 Trimming spectral objects

> Because of how `[ ]` operators work in R, and especially on objects of classes derived from `data.table` some object attributes are lost when this operator is used to subset spct objects, consequently it is safer to use the function described in this section.

Sometimes it is desirable to change the range of wavelengths included in a spectrum. If we are interested in a given part of the spectrum, there is no need to do calculations or plotting the whole spectrum. Sometimes we may want to expand the range of wavelengths, filling the expansion of all other variables with a certain value (i.e. a number, or NA.)

We can supply the arguments `band`, `low.limit`, `high.limit`, and `fill`. Either `band` or `xxx.limit` arguments should supplied, but not both at once. We use `head` to print the first six lines.

```
head(trim_spct(sun.spct, PAR()))
```

```
##    w.length s.e.irrad    s.q.irrad
## 1:      400 0.6081049 2.033314e-06
## 2:      401 0.6261742 2.098967e-06
## 3:      402 0.6497388 2.183388e-06
## 4:      403 0.6207287 2.091091e-06
## 5:      404 0.6370489 2.151395e-06
## 6:      405 0.6263786 2.120596e-06
```

```
head(trim_spct(sun.spct, low.limit = 297))
```

```
##    w.length    s.e.irrad    s.q.irrad
## 1:      297 0.0001533491 3.807181e-10
## 2:      298 0.0003669677 9.141345e-10
## 3:      299 0.0007845430 1.960893e-09
## 4:      300 0.0012645540 3.171207e-09
## 5:      301 0.0026237179 6.601607e-09
## 6:      302 0.0039225827 9.902505e-09
```

By default trim_spct trims its argument by copy, this can be changed by setting byref = TRUE but as sun.spct is protected as part of the package, we cannot use it here.

```
my_sun.spct <- copy(sun.spct)
head(trim_spct(my_sun.spct, low.limit = 297, byref = TRUE))

##    w.length    s.e.irrad    s.q.irrad
## 1:      297 0.0001533491 3.807181e-10
## 2:      298 0.0003669677 9.141345e-10
## 3:      299 0.0007845430 1.960893e-09
## 4:      300 0.0012645540 3.171207e-09
## 5:      301 0.0026237179 6.601607e-09
## 6:      302 0.0039225827 9.902505e-09
```

The default `fill` value is `NULL` which means deleting the values outside the trimmed region. It is possible to supply a different argument.

```
head(trim_spct(sun.spct, low.limit = 297, fill = 0))

##    w.length    s.e.irrad    s.q.irrad
## 1: 293.0000 0.0000000000 0.000000e+00
## 2: 294.0000 0.0000000000 0.000000e+00
## 3: 295.0000 0.0000000000 0.000000e+00
## 4: 296.0000 0.0000000000 0.000000e+00
## 5: 296.9999 0.0000000000 0.000000e+00
## 6: 297.0000 0.0001533491 3.807181e-10
```

```
head(trim_spct(sun.spct, low.limit = 297, fill = NA))

##    w.length    s.e.irrad    s.q.irrad
## 1: 293.0000           NA           NA
## 2: 294.0000           NA           NA
## 3: 295.0000           NA           NA
## 4: 296.0000           NA           NA
## 5: 296.9999           NA           NA
## 6: 297.0000 0.0001533491 3.807181e-10
```

In addition, when fill is not `NULL`, expansion is possible.

```
head(trim_spct(sun.spct, low.limit = 290, fill = 0))

##    w.length    s.e.irrad    s.q.irrad
## 1:      290 0.000000e+00 0.000000e+00
## 2:      291 0.000000e+00 0.000000e+00
## 3:      292 0.000000e+00 0.000000e+00
## 4:      293 2.609665e-06 6.391730e-12
## 5:      294 6.142401e-06 1.509564e-11
## 6:      295 2.176175e-05 5.366385e-11
```

3.9 Summaries

Functions `integrate_spct` and `average_spct` take into account each individual wavelength step, so they return valid results even for spectra measured at arbitrary and varying wavelength steps.

```
integrate_spct(sun.spct)

##      e.irrad      q.irrad
## 2.691249e+02 1.255336e-03

average_spct(sun.spct)

##      e.irrad      q.irrad
## 5.308183e-01 2.476007e-06
```

The 'usual' and a couple of new functions are available for spectra, but redefined to return wavelengths.

```
range(sun.spct)

## [1] 293 800

min(sun.spct)

## [1] 293

max(sun.spct)

## [1] 800

midpoint(sun.spct)

## [1] 546.5

spread(sun.spct)

## [1] 507

stepsize(sun.spct)

## [1] 1 1
```

Function `stepsize` computes the size of every single step in the spectrum, and returns the range of these values. In the example above for a simulated spectrum the step size is uniform, but in data from array spectrometers this is not the norm.

```
stepsize(sun_May_morning.spct)

## [1] 0.43 0.48
```

Specialized definitions of `summary` and the corresponding `print` methods are available for spectral objects. In the case of `source.spct` objects the `time.unit` attribute makes it possible to print the summary using the correct units.

```
summary(sun.spct)

## wavelength ranges from 293 to 800 nm
```

```
## largest wavelength step size is 1 nm
## spectral irradiance ranges from 2.61e-06 to 0.8205 W m-2 nm-1
## energy irradiance is 268.9 W m-2
## photon irradiance is 1254 umol s-1 m-2
```

```
summary(sun.daily.spct)

## wavelength ranges from 290 to 800 nm
## largest wavelength step size is 1 nm
## spectral irradiance ranges from 0 to 32.61 kJ d-1 m-2 nm-1
## energy irradiance is 10.92 MJ m-2
## photon irradiance is 51.33 mol d-1 m-2
```

3.10 Defining wavebands

All functions use `wavebands` as definitions of the range of wave lengths and the spectral weighting function (SWF) to use in the calculations. A few other bits of information may be included to fine-tune calculations. The waveband definitions do NOT describe whether input spectral irradiances are photon or energy based, nor whether the output irradiance will be based on photon or energy units. Waveband objects belong to the S3 class "waveband".

When defining a waveband which uses a SWF, a function can be supplied either based on energy effectiveness, on photon effectiveness, or one function for each one. If only one function is supplied the other one is built automatically, but if performance is a concern it is better to provide two separate functions. Another case when you might want to enter the same function twice, is if you are using an absorptance spectrum as SWF, as the percent of radiation absorbed will be independent of whether photon or energy units are used for the spectral irradiance.

Two different functions can be used to create a waveband: `waveband` and `new_waveband`.

The difference is that `waveband` accepts the limits through a single argument, which can be any R object for which there is a suitable `range` function, which returns the range of wavelengths as a numeric vector of length 2.

```
my_PAR <- new_waveband(400, 700)
my_PARx <- new_waveband(400, 700, wb.name = "my_PARx")

my_CIE_1 <-
  new_waveband(250, 400, weight = "SWF", SWF.e.fun = CIE.e.fun, SWF.norm = 298)
my_CIE_2 <-
  new_waveband(250, 400, weight = "SWF", SWF.q.fun = CIE.q.fun, SWF.norm = 298)
my_CIE_3 <-
  new_waveband(250, 400, weight = "SWF", SWF.e.fun = CIE.e.fun,
                         SWF.q.fun = CIE.q.fun, SWF.norm = 298)
```

The first example above, can be also written as, and in the same way all other statements above can be rewritten, replacing the two separate limits with an array of length 2:

```
my_PAR <- waveband(c(400, 700))
```

The function `waveband` is useful when wanting to create a waveband covering the whole range of an spectrum, on when creating an unweighted waveband which covers exactly the same range of wavelengths as an existing weighted waveband.

```
waveband(sun.spct)

## Total
## low (nm) 293
## high (nm) 800
## weighted none

waveband(my_CIE_1)

## range.250.400
## low (nm) 250
## high (nm) 400
## weighted none
```

The function `split_bands` can be used to generate lists of unweighted wavebands in two different ways: a) it can be used to split a range of wavelengths given by an R object into a series of adjacent wavebands, or b) with a list of objects returning ranges, it can be used to create non-adjacent and even overlapping wavebands.

The code chunk bellow shows an example of two variations of case a). With the default value for `length.out` of `NULL` each numerical value in the input is taken as a wavelength (nm) at the boundary between adjacent wavebands. If a numerical value is supplied to `length.out`, then the whole wavelength range of the input is split into this number of equally spaced adjacent wavebands.

```
split_bands(c(200, 225, 300))

## $wb1
## range.200.225
## low (nm) 200
## high (nm) 225
## weighted none
##
## $wb2
## range.225.300
## low (nm) 225
## high (nm) 300
## weighted none

split_bands(c(200, 225, 300), length.out = 2)

## $wb1
## range.200.250
## low (nm) 200
## high (nm) 250
```

```
## weighted none
##
## $wb2
## range.250.300
## low (nm) 250
## high (nm) 300
## weighted none
```

In both examples above, the output is a list of two wavebands, but the boundary is at a different wavelength. The chunk bellow gives a few more examples of the use of case a).

```
split_bands(sun.spct, length.out = 2)

## $wb1
## range.293.546.5
## low (nm) 293
## high (nm) 546
## weighted none
##
## $wb2
## range.546.5.800
## low (nm) 546
## high (nm) 800
## weighted none

split_bands(PAR(), length.out = 2)

## $wb1
## range.400.550
## low (nm) 400
## high (nm) 550
## weighted none
##
## $wb2
## range.550.700
## low (nm) 550
## high (nm) 700
## weighted none

split_bands(c(200, 800), length.out = 3)

## $wb1
## range.200.400
## low (nm) 200
## high (nm) 400
## weighted none
##
## $wb2
## range.400.600
## low (nm) 400
## high (nm) 600
## weighted none
##
## $wb3
## range.600.800
```

```
## low (nm) 600
## high (nm) 800
## weighted none

# we use head show the first two out of 100 wavebands
head(split_bands(c(200, 800), length.out = 100), 2)

## $wb1
## range.200.206
## low (nm) 200
## high (nm) 206
## weighted none
##
## $wb2
## range.206.212
## low (nm) 206
## high (nm) 212
## weighted none
```

Now we demonstrate case b). This is handles by recursion, so each list element can be anything that is a valid input to the function, including a nested list. However, the returned value is always a flat list of wavebands.

```
split_bands(list(A = c(200, 300), B = c(400, 500), C = c(250, 350)))

## $A
## range.200.300
## low (nm) 200
## high (nm) 300
## weighted none
##
## $B
## range.400.500
## low (nm) 400
## high (nm) 500
## weighted none
##
## $C
## range.250.350
## low (nm) 250
## high (nm) 350
## weighted none

split_bands(list(c(100, 150, 200), c(800, 825)))

## $wb.a
## range.100.150
## low (nm) 100
## high (nm) 150
## weighted none
##
## $<NA>
## range.150.200
## low (nm) 150
## high (nm) 200
## weighted none
```

```
##
## $wb.b
## range.800.825
## low (nm) 800
## high (nm) 825
## weighted none
```

In case b) if we supply a numeric value to `length.out`, this value is used recursively for each element of the list.

```
split_bands(list(R = Red(), B = Blue()), length.out  =  2)

## $R
## range.610.685
## low (nm) 610
## high (nm) 685
## weighted none
##
## $<NA>
## range.685.760
## low (nm) 685
## high (nm) 760
## weighted none
##
## $B
## range.450.475
## low (nm) 450
## high (nm) 475
## weighted none
##
## $<NA>
## range.475.500
## low (nm) 475
## high (nm) 500
## weighted none

split_bands(list(c(100, 150, 200), c(800, 825)), length.out = 1)

## $wb.a
## range.100.200
## low (nm) 100
## high (nm) 200
## weighted none
##
## $wb.b
## range.800.825
## low (nm) 800
## high (nm) 825
## weighted none
```

The function `is.waveband` can the used to query any R object.

```
is.waveband(my_CIE_1)

## [1] TRUE
```

```
is.waveband(PAR())

## [1] TRUE

is.waveband(sun.spct)

## [1] FALSE
```

The function `is.effective` can the used to query any R object.

```
is.effective(my_CIE_1)

## [1] TRUE

is.effective(GEN.G())

## [1] TRUE

is.effective(PAR())

## [1] FALSE

is.effective(sun.spct)

## [1] NA
```

3.11 Using operators with spectra

3.11.1 Binary operators

The basic maths operators have definitions for spectra. It is possible to sum, subtract, multiply and divide spectra. These operators can be used even if the spectral data is on different arbitrary sets of wavelengths. Operators by default use values expressed in energy units. Only certain operations are meaningful for a given combination of objects belonging to different classes, and meaningless combinations return `NA` also issuing a warning (see Table 1). By default operations are carried out on spectral energy irradiance for `source.spct` objects and transmittance for `filter.spct` objects.

```
sun.spct * sun.spct

##      w.length    s.e.irrad
##   1:      293 6.810352e-12
##   2:      294 3.772909e-11
##  ---
## 507:      799 1.752134e-01
## 508:      800 1.655721e-01

sun.spct / sun.spct

##      w.length s.e.irrad
##   1:      293         1
```

Table 1: Binary operators and operands. Validity and class of result. All operations marked 'Y' are allowed, those marked 'N' are forbiden and return `NA` and issue a warning.

e1	+	-	*	/	^	e2	result
source.spct	Y	Y	Y	Y	Y	source.spct	source.spct
filter.spct (T)	N	N	Y	Y	N	filter.spct	filter.spct
filter.spct (A)	Y	Y	N	N	N	filter.spct	filter.spct
reflector.spct	N	N	Y	Y	N	reflector.spct	reflector.spct
response.spct	Y	Y	Y	Y	N	response.spct	response.spct
chroma.spct	Y	Y	Y	Y	Y	chroma.spct	chroma.spct
source.spct	Y	Y	Y	Y	Y	numeric	source.spct
filter.spct	Y	Y	Y	Y	Y	numeric	filter.spct
reflector.spct	Y	Y	Y	Y	Y	numeric	reflector.spct
response.spct	Y	Y	Y	Y	Y	numeric	response.spct
chroma.spct	Y	Y	Y	Y	Y	numeric	chroma.spct
source.spct	N	N	Y	Y	N	response.spct	response.spct
source.spct	N	N	Y	Y	N	filter.spct (T)	source.spct
source.spct	N	N	Y	Y	N	filter.spct (A)	source.spct
source.spct	N	N	Y	Y	N	reflector.spct	source.spct
source.spct	N	N	Y	N	N	waveband (no BSWF)	source.spct
source.spct	N	N	Y	N	N	waveband (BSWF)	response.spct

```
##   2:      294         1
##  ---
## 507:      799         1
## 508:      800         1

sun.spct + sun.spct

##      w.length    s.e.irrad
##   1:      293 5.219330e-06
##   2:      294 1.228480e-05
##  ---
## 507:      799 8.371699e-01
## 508:      800 8.138111e-01

sun.spct - sun.spct

##      w.length s.e.irrad
##   1:      293         0
##   2:      294         0
##  ---
## 507:      799         0
## 508:      800         0
```

When meaningful operations between different spectra are also allowed. For example, it is possible to simulate the effect of a filter on a light source by multiplying (or convoluting) the two spectra.

```
sun.spct * polyester.new.spct

##      w.length    s.e.irrad
##   1:      293 7.828995e-09
##   2:      294 1.842720e-08
##  ---
## 507:      799 3.809123e-01
## 508:      800 3.706909e-01
```

If we have two layers of the filter, this can be approximated using either of these two statements.

```
sun.spct * polyester.new.spct * polyester.new.spct

##      w.length    s.e.irrad
##   1:      293 2.348699e-11
##   2:      294 5.528161e-11
##  ---
## 507:      799 3.466302e-01
## 508:      800 3.376994e-01

sun.spct * polyester.new.spct^2

##      w.length    s.e.irrad
##   1:      293 2.348699e-11
##   2:      294 5.528161e-11
##  ---
## 507:      799 3.466302e-01
## 508:      800 3.376994e-01
```

Operators are also defined for operations between a spectrum and a numeric vector (with normal recycling).

```
sun.spct * 2
```

```
##      w.length    s.e.irrad
##   1:      293 5.219330e-06
##   2:      294 1.228480e-05
##  ---
## 507:      799 8.371699e-01
## 508:      800 8.138111e-01
```

```
2 * sun.spct
```

```
##      w.length    s.e.irrad
##   1:      293 5.219330e-06
##   2:      294 1.228480e-05
##  ---
## 507:      799 8.371699e-01
## 508:      800 8.138111e-01
```

```
sun.spct * c(0,1)
```

```
##      w.length    s.e.irrad
##   1:      293 0.000000e+00
##   2:      294 6.142401e-06
##  ---
## 507:      799 0.000000e+00
## 508:      800 4.069055e-01
```

There is one special case, for `chroma.spct`: if the numeric operand has length three, containing three *named* values 'x', 'y' and 'z', the corresponding value is used for each of the chromaticity 'columns' in the `chroma.spct`. Un-named values or differently named values are not treated specially.

Operators are also defined for operations between an spectrum and a `waveband` object. The next to code chunks demonstrate how the class of the result depends on whether the `waveband` object describes a range of wavelengths or a range of wavelengths plus a BSWF.

```
is.effective(UVB())
```

```
## [1] FALSE
```

```
clipped.spct <- sun.spct * UVB()
class.spct(clipped.spct)
```

```
## [1] "source.spct"  "generic.spct"
```

```
clipped.spct
```

```
##     w.length    s.e.irrad
##  1:  293.000 2.609665e-06
##  2:  294.000 6.142401e-06
## ---
## 22:  314.000 1.054126e-01
## 23:  314.999 1.127828e-01
```

```
is.effective(CIE())

## [1] TRUE

weighted.spct <- sun.spct * CIE()
class.spct(weighted.spct)

## [1] "response.spct" "generic.spct"

weighted.spct

##      w.length s.e.response
##   1:  293.000 2.609665e-06
##   2:  294.000 6.142401e-06
##  ---
## 107:  399.000 7.378801e-05
## 108:  399.999 7.395674e-05
```

And of course these operations can be combined into more complex statements, including parentheses, when needed. The example below estimates the difference in effective spectral irradiance according to the CIE98 definition, between sunlight and sunlight filtered with a polyester film. Of course, the result is valid only for the solar spectral data used, which corresponds to Southern Finland.

```
sun.spct * CIE() - sun.spct * polyester.new.spct * CIE()

##      w.length s.e.response
##   1:  293.000 2.601836e-06
##   2:  294.000 6.123973e-06
##  ---
## 107:  399.000 6.493345e-06
## 108:  399.999 6.434308e-06
```

3.11.2 Unary operators and math functions

The most common math functions, as well as unary minus and plus, are also implemented for spectral objects (see Table 2).

```
+sun.spct

##      w.length    s.e.irrad
##   1:      293 2.609665e-06
##   2:      294 6.142401e-06
##  ---
## 507:      799 4.185850e-01
## 508:      800 4.069055e-01

-sun.spct

##      w.length     s.e.irrad
##   1:      293 -2.609665e-06
##   2:      294 -6.142401e-06
```

Table 2: Unary operators and math functions. Validity and class of result. All operations marked 'Y' are allowed, those marked 'N' are not implemented and return `NA` and issue a warning.

e1	`+`	`-`	`log()`	`log10()`	`exp()`	`sqrt()`	result
source.spct	Y	Y	Y	Y	Y	Y	source.spct
filter.spct	Y	Y	Y	Y	Y	Y	filter.spct
reflector.spct	Y	Y	Y	Y	Y	Y	reflector.spct
response.spct	Y	Y	Y	Y	Y	Y	response.spct
chroma.spct	Y	Y	Y	Y	Y	Y	chroma.spct

```
##  ---
## 507:      799 -4.185850e-01
## 508:      800 -4.069055e-01

log(sun.spct)

##      w.length   s.e.irrad
##   1:      293 -12.8562887
##   2:      294 -12.0002949
##  ---
## 507:      799  -0.8708754
## 508:      800  -0.8991742

log10(sun.spct)

##      w.length  s.e.irrad
##   1:      293 -5.5834152
##   2:      294 -5.2116619
##  ---
## 507:      799 -0.3782164
## 508:      800 -0.3905064

exp(sun.spct)

##      w.length s.e.irrad
##   1:      293  1.000003
##   2:      294  1.000006
##  ---
## 507:      799  1.519809
## 508:      800  1.502162

sqrt(sun.spct)

##      w.length   s.e.irrad
##   1:      293 0.001615446
##   2:      294 0.002478387
##  ---
## 507:      799 0.646981421
## 508:      800 0.637891472
```

3.11.3 Options affecting math operators and functions

The behaviour of the operators depends on the value of two global options. If we would like the operators to operate on spectral photon irradiance and return spectral photon irradiance instead of spectral energy irradiance, this behaviour can be set, and will remain active until unset or reset.

```
options(photobiology.radiation.unit = "photon")
sun.spct * UVB()

##     w.length    s.q.irrad
##  1:  293.000 6.391730e-12
##  2:  294.000 1.509564e-11
## ---
## 22:  314.000 2.766867e-07
## 23:  314.999 2.969737e-07

options(photobiology.radiation.unit = "energy")
sun.spct * UVB()

##     w.length    s.e.irrad
##  1:  293.000 2.609665e-06
##  2:  294.000 6.142401e-06
## ---
## 22:  314.000 1.054126e-01
## 23:  314.999 1.127828e-01
```

For filters, an option controls whether transmittance, the default, or absorbance is use in the operations, and returned.

```
options(photobiology.filter.qty = "absorbance")
polyester.new.spct * 2

##      w.length          A
##   1:      190 3.91721463
##   2:      191 4.00000000
##  ---
## 610:      799 0.08191722
## 611:      800 0.08096325

options(photobiology.filter.qty = "transmittance")
polyester.new.spct ^ 2

##      w.length      Tfr
##   1:      190 0.000121
##   2:      191 0.000100
##  ---
## 610:      799 0.828100
## 611:      800 0.829921
```

Either option can be unset, by means of the `NULL` value.

```
options(photobiology.radiation.unit = NULL)
options(photobiology.filter.qty = NULL)
```

3.12 Calculating irradiance or exposure

The package includes two 'families' of functions, one taking as argument spectrum objects and another taking numeric vectors as arguments. We prefer in general the first 'family'.

In each 'family' there is one basic function for these calculations `irradiance()` or `irrad`, and specialized functions for 'photon' and 'energy' based calculations.

3.12.1 Irradiances from spectra

The code using `spct` objects is simple, to integrate the whole spectrum we can use

```
irrad(sun.spct)

##    Total
## 268.9214
## attr(,"time.unit")
## [1] "second"
## attr(,"radiation.unit")
## [1] "energy irradiance total"
```

and, to integrate a range of wavelength, in the example, photosynthetically active radiation, we use `PAR()` that is a predefined waveband constructor.

```
irrad(sun.spct, PAR(), unit.out = "energy") # W m-2

##      PAR
## 196.7004
## attr(,"time.unit")
## [1] "second"
## attr(,"radiation.unit")
## [1] "energy irradiance total"

irrad(sun.spct, PAR(), unit.out = "photon") # mol s-1 m-2

##          PAR
## 0.0008937598
## attr(,"time.unit")
## [1] "second"
## attr(,"radiation.unit")
## [1] "photon irradiance total"

irrad(sun.spct, PAR(), unit.out = "photon") * 1e6 # umol s-1 m-2

##      PAR
## 893.7598
## attr(,"time.unit")
## [1] "second"
## attr(,"radiation.unit")
## [1] "photon irradiance total"
```

The default for `irrad`, when no argument `unit.out` is supplied, is to return the irradiance value in energy irradiance units, unless the R `photobiology.radiation.unit` option is set.

```
irrad(sun.spct, PAR()) # W m-2

##      PAR
## 196.7004
## attr(,"time.unit")
## [1] "second"
## attr(,"radiation.unit")
## [1] "energy irradiance total"

options(photobiology.radiation.unit = "photon")
irrad(sun.spct, PAR()) # mol s-1 m-2

##          PAR
## 0.0008937598
## attr(,"time.unit")
## [1] "second"
## attr(,"radiation.unit")
## [1] "photon irradiance total"

options(photobiology.radiation.unit = NULL)
```

Functions `e_irrad` and `q_irrad` save some typing, and always return the same type of spectral irradiance quantity, independently of global option `photobiology.radiation.unit`.

```
e_irrad(sun.spct, PAR()) # W m-2

##      PAR
## 196.7004
## attr(,"time.unit")
## [1] "second"
## attr(,"radiation.unit")
## [1] "energy irradiance total"

q_irrad(sun.spct, PAR()) * 1e6 # umol s-1 m-2

##      PAR
## 893.7598
## attr(,"time.unit")
## [1] "second"
## attr(,"radiation.unit")
## [1] "photon irradiance total"

q_irrad(sun.daily.spct, PAR()) # mol d-1 m-2

##      PAR
## 36.29631
## attr(,"time.unit")
## [1] "day"
## attr(,"radiation.unit")
## [1] "photon irradiance total"
```

We can use predefined waveband constructors, waveband objects, or define wavebands on the fly.

```
my_par <- PAR()
e_irrad(sun.spct, my_par) # W m-2

##      PAR
## 196.7004
## attr(,"time.unit")
## [1] "second"
## attr(,"radiation.unit")
## [1] "energy irradiance total"

e_irrad(sun.spct, waveband(c(400,700))) # W m-2

##  range.400.700
##       196.7004
## attr(,"time.unit")
## [1] "second"
## attr(,"radiation.unit")
## [1] "energy irradiance total"
```

Lists of wavebands are also accepted as argument.

```
e_irrad(sun.spct, list(CIE(), CIE(298), CIE(300)))

##  CIE98.298.tr.lo  CIE98.298.tr.lo  CIE98.300.tr.lo
##       0.08177754       0.08177754       0.12607648
## attr(,"time.unit")
## [1] "second"
## attr(,"radiation.unit")
## [1] "energy irradiance total"

my_wavebands <- list(Red(), Blue(), Green())
e_irrad(sun.spct, my_wavebands)

##    Red.ISO   Blue.ISO  Green.ISO
##   79.61176   37.57760   49.30478
## attr(,"time.unit")
## [1] "second"
## attr(,"radiation.unit")
## [1] "energy irradiance total"
```

These functions have an additional argument `quantity`, with default `"total"`, which can take values controlling the output.

```
irrad(sun.spct, UV_bands())

##  UVB.ISO.tr.lo        UVA.ISO
##      0.5881141     27.7365487
## attr(,"time.unit")
## [1] "second"
## attr(,"radiation.unit")
## [1] "energy irradiance total"

irrad(sun.spct, UV_bands(), quantity = "total")
```

```
##  UVB.ISO.tr.lo        UVA.ISO
##      0.5881141     27.7365487
## attr(,"time.unit")
## [1] "second"
## attr(,"radiation.unit")
## [1] "energy irradiance total"

irrad(sun.spct, UV_bands(), quantity = "contribution")

##  UVB.ISO.tr.lo        UVA.ISO
##    0.002186936    0.103139966
## attr(,"time.unit")
## [1] "second"
## attr(,"radiation.unit")
## [1] "energy irradiance contribution"

irrad(sun.spct, UV_bands(), quantity = "contribution.pc")

##  UVB.ISO.tr.lo        UVA.ISO
##      0.2186936     10.3139966
## attr(,"time.unit")
## [1] "second"
## attr(,"radiation.unit")
## [1] "energy irradiance contribution.pc"

irrad(sun.spct, UV_bands(), quantity = "relative")

##  UVB.ISO.tr.lo        UVA.ISO
##     0.02076332     0.97923668
## attr(,"time.unit")
## [1] "second"
## attr(,"radiation.unit")
## [1] "energy irradiance relative"

irrad(sun.spct, UV_bands(), quantity = "relative.pc")

##  UVB.ISO.tr.lo        UVA.ISO
##       2.076332      97.923668
## attr(,"time.unit")
## [1] "second"
## attr(,"radiation.unit")
## [1] "energy irradiance relative.pc"

irrad(sun.spct, UV_bands(), quantity = "average")

##  UVB.ISO.tr.lo        UVA.ISO
##     0.02673246     0.32631234
## attr(,"time.unit")
## [1] "second"
## attr(,"radiation.unit")
## [1] "energy irradiance average"
```

3.12.2 Irradiances from numeric vectors

The code using numeric vectors is more complicated, but adds some additional flexibility. Under normal circumstances it is easier to use the functions described above.

Function `irradiance` takes an array of wavelengths (sorted in strictly increasing order), and the corresponding values of spectral irradiance. By default the input is assumed to be in energy units, but parameter `unit.in` cab be used to adjust the calculations to expect photon units. The type of unit used for the calculated irradiance (or exposure) is set by the parameter `unit.out` with no default. If no `w.band` parameter is supplied, the whole spectrum spectrum input is used, unweighted, to calculate the total irradiance. If a `w.band` is supplied, then the range of wavelengths specified and SWF if present are used for calculating the irradiance. If the waveband definition does not include a SWF, then the unweighted irradiance is returned, if the definition includes a SWF, then a weighted irradiance is returned.

The functions `photon_irradiance()` and `energy_irradiance()`, just call `irradiance()` with the `unit.out` set to `"photon"` or `"energy"` respectively.

The functions taking numerical vectors as arguments can be used with data stored as vectors, or using `with` with data frames, data tables, lists, and spectra objects.

```
with(sun.data, photon_irradiance(w.length, s.e.irrad, PAR()))

##          PAR
## 0.0008937598

with(sun.spct, photon_irradiance(w.length, s.e.irrad, PAR()))

##          PAR
## 0.0008937598
```

Lists of wavebands are also accepted as argument.

```
with(sun.data, energy_irradiance(w.length, s.e.irrad, list(CIE(), CIE(298), CIE(300))))

##  CIE98.298  CIE98.298  CIE98.300
## 0.08177754 0.08177754 0.12607648

my_wavebands <- list(Red(), Blue(), Green())
with(sun.data, energy_irradiance(w.length, s.e.irrad, my_wavebands))

##   Red.ISO  Blue.ISO Green.ISO
##  79.61176  37.57760  49.30478
```

The recommended practice is to use `with`, as above.

The are also available convenience functions for calculating how 'total' irradiance is split among different contiguous bands of the spectrum. The functions `split_photon_irradiance()` and `split_energy_irradiance()`, just call `split_irradiance()` with the `unit.out` set to `"photon"` or `"energy"` respectively.

```
with(sun.data,
     split_energy_irradiance(w.length, s.e.irrad,
                             c(300, 400, 500, 600, 700, 800))
)

## range.300.400 range.400.500 range.500.600 range.600.700 range.700.800
##      28.32326      69.63243      68.53291      58.53508      43.89636

with(sun.data,
     split_energy_irradiance(w.length, s.e.irrad,
                             c(400, 500, 600, 700),
                             scale = "percent")
)

## range.400.500 range.500.600 range.600.700
##      35.40024      34.84126      29.75849

with(sun.data,
     split_photon_irradiance(w.length, s.e.irrad,
                             c(400, 500, 600, 700),
                             scale = "percent")
)

## range.400.500 range.500.600 range.600.700
##      29.40969      35.13940      35.45092
```

3.13 Calculating ratios

The package includes two 'families' of functions, one taking as argument spectrum objects and another taking numeric vectors as arguments. We prefer in general the first 'family'.

In each 'family' there is one basic function for these calculations `waveband_ratio()` or `ratio`, and specialized functions for 'photon' and 'energy' based calculations.

3.13.1 Ratios from spectra

The functions described here, in there simplest use, calculate a ratio between two wavebands. The function `q_ratio` returning photon ratios. However both waveband parameters can take lists of wavebands as arguments, with normal recycling rules in effect.

```
q_ratio(sun.spct, UVB(), PAR())

##  UVB.ISO.tr.lo: PAR(q:q)
##              0.001708359
## attr(,"radiation.unit")
## [1] "q:q ratio"

q_ratio(sun.spct,
        list(UVC(), UVB(), UVA()),
        UV())
```

```
##  UVB.ISO.tr.lo: UV.ISO.tr.lo(q:q)        UVA.ISO: UV.ISO.tr.lo(q:q)
##                         0.01786255                         0.98213745
## attr(,"radiation.unit")
## [1] "q:q ratio"

q_ratio(sun.spct,
        UVB(),
        list(UV(), PAR()))

##  UVB.ISO.tr.lo: UV.ISO.tr.lo(q:q)           UVB.ISO.tr.lo: PAR(q:q)
##                        0.017862550                        0.001708359
## attr(,"radiation.unit")
## [1] "q:q ratio"
```

Function `e_ratio` returns energy ratios.

```
e_ratio(sun.spct, UVB(), PAR())

##  UVB.ISO.tr.lo: PAR(e:e)
##               0.002989897
## attr(,"radiation.unit")
## [1] "e:e ratio"

e_ratio(sun.spct,
        list(UVC(), UVB(), UVA()),
        UV())

##  UVB.ISO.tr.lo: UV.ISO.tr.lo(e:e)        UVA.ISO: UV.ISO.tr.lo(e:e)
##                         0.02076332                         0.97923668
## attr(,"radiation.unit")
## [1] "e:e ratio"
```

Function `qe_ratio`, has only one waveband parameter, and returns the 'photon' to 'energy' ratio,

```
qe_ratio(sun.spct, PAR())

##    q:e( PAR)
## 4.543762e-06
## attr(,"radiation.unit")
## [1] "q:e ratio"

qe_ratio(sun.spct, list(Blue(), Green(), Red()))

##  q:e( Blue.ISO) q:e( Green.ISO)   q:e( Red.ISO)
##    3.964423e-06    4.465210e-06    5.679688e-06
## attr(,"radiation.unit")
## [1] "q:e ratio"
```

Function `eq_ratio`, has only one waveband parameter, and returns the'energy' to 'photon' ratio,

```
eq_ratio(sun.spct, PAR())
```

```
## e:q( PAR)
##    220082
## attr(,"radiation.unit")
## [1] "e:q ratio"

eq_ratio(sun.spct, list(Blue(), Green(), Red()))

##  e:q( Blue.ISO) e:q( Green.ISO)   e:q( Red.ISO)
##        252243.5        223953.6        176066.0
## attr(,"radiation.unit")
## [1] "e:q ratio"
```

If we would like to calculate a conversion factor between PPFD (PAR photon irradiance in mol s-1 m-2) and PAR (energy) irradiance (W m-2) for a light source for which we have spectral data we could use the following code.

```
conv.factor <- qe_ratio(sun.spct, PAR())

PPFD.mol.photon <- 1000e-6
PAR.energy <- PPFD.mol.photon / conv.factor
conv.factor

##    q:e( PAR)
## 4.543762e-06
## attr(,"radiation.unit")
## [1] "q:e ratio"

PPFD.mol.photon * 1e6

## [1] 1000

PAR.energy

## q:e( PAR)
##   220.082
## attr(,"radiation.unit")
## [1] "q:e ratio"
```

3.13.2 Ratios from vectors

The function `waveband_ratio()` takes basically the same parameters as `irradiance`, but two waveband definitions instead of one, and two `unit.out` definitions instead of one. This is the base function used in all the vector based 'ratio' functions in the photobiology package.

The derived functions are: `photon_ratio()`, `energy_ratio()`, and `photons_energy_ratio`. The packages and use these to define some convenience functions, and here we give and example for a function not yet implemented, but which you may find as a useful example.

In contrast to the functions described in the previous section, these functions only accept individual waveband definitions (not lists of them).

If for example we would like to calculate the ratio between UVB and PAR radiation, we would use either of the following function calls, depending on which type of units we desire.

```
with(sun.data,
     photon_ratio(w.length, s.e.irrad, UVB(), PAR())
)

## [1] 0.001708359

with(sun.data,
     energy_ratio(w.length, s.e.irrad, UVB(), PAR())
)

## [1] 0.002989897
```

3.14 Calculating average transmittance, absorbance and reflectance

The functions `transmittance`, `absorbance` and `reflectance` take filter.spct and reflector.spct objects as arguments, and return an average value for these quantities **assuming** a light source with a flat spectral energy output. Values expressed as percentages.

```
transmittance(polyester.new.spct, list(UVB(), UVA(), PAR()))

##     UVB.ISO     UVA.ISO         PAR
## 0.007671429 0.782682353 0.920245000
## attr(,"time.unit")
## [1] "none"
## attr(,"Tfr.type")
## [1] "total"
## attr(,"radiation.unit")
## [1] "transmittance average"

transmittance(polyester.new.spct, list(UVB(), UVA(), PAR()), pc.out = TRUE)

##    UVB.ISO    UVA.ISO        PAR
##  0.7671429 78.2682353 92.0245000
## attr(,"time.unit")
## [1] "none"
## attr(,"Tfr.type")
## [1] "total"
## attr(,"radiation.unit")
## [1] "transmittance average (%)"
```

This function has an additional argument `quantity`, with default `"average"`, which can take values controlling the output.

```
transmittance(polyester.new.spct, UV_bands())

##     UVB.ISO     UVA.ISO
```

```
## 0.007671429 0.782682353
## attr(,"time.unit")
## [1] "none"
## attr(,"Tfr.type")
## [1] "total"
## attr(,"radiation.unit")
## [1] "transmittance average"

transmittance(polyester.new.spct, UV_bands(), quantity = "total")

##  UVB.ISO  UVA.ISO
##   0.2685  66.5280
## attr(,"time.unit")
## [1] "none"
## attr(,"Tfr.type")
## [1] "total"
## attr(,"radiation.unit")
## [1] "transmittance total"

transmittance(polyester.new.spct, UV_bands(), quantity = "contribution")

##      UVB.ISO      UVA.ISO
## 0.0006178763 0.1530952496
## attr(,"time.unit")
## [1] "none"
## attr(,"Tfr.type")
## [1] "total"
## attr(,"radiation.unit")
## [1] "transmittance contribution"

transmittance(polyester.new.spct, UV_bands(), quantity = "contribution.pc")

##     UVB.ISO     UVA.ISO
##  0.06178763 15.30952496
## attr(,"time.unit")
## [1] "none"
## attr(,"Tfr.type")
## [1] "total"
## attr(,"radiation.unit")
## [1] "transmittance contribution.pc"

transmittance(polyester.new.spct, UV_bands(), quantity = "relative")

##     UVB.ISO     UVA.ISO
## 0.004019672 0.995980328
## attr(,"time.unit")
## [1] "none"
## attr(,"Tfr.type")
## [1] "total"
## attr(,"radiation.unit")
## [1] "transmittance relative"

transmittance(polyester.new.spct, UV_bands(), quantity = "relative.pc")

##    UVB.ISO    UVA.ISO
##  0.4019672 99.5980328
## attr(,"time.unit")
```

```
## [1] "none"
## attr(,"Tfr.type")
## [1] "total"
## attr(,"radiation.unit")
## [1] "transmittance relative.pc"

transmittance(polyester.new.spct, UV_bands(), quantity = "average")

##     UVB.ISO     UVA.ISO
## 0.007671429 0.782682353
## attr(,"time.unit")
## [1] "none"
## attr(,"Tfr.type")
## [1] "total"
## attr(,"radiation.unit")
## [1] "transmittance average"
```

An equivalent function returning absorbance instead of transmittance takes the same arguments as `transmittance`, except for `pc.out` which is not useful for absorbance.

```
absorbance(polyester.new.spct, list(UVB(), UVA(), PAR()))

##    UVB.ISO    UVA.ISO        PAR
## 2.30895546 0.12895143 0.03610019
## attr(,"time.unit")
## [1] "none"
## attr(,"Tfr.type")
## [1] "total"
## attr(,"radiation.unit")
## [1] "absorbance average"
```

This function has an additional argument `quantity`, with default `"average"`, which can take values controlling the output.

```
transmittance(polyester.new.spct, UV_bands())

##     UVB.ISO     UVA.ISO
## 0.007671429 0.782682353
## attr(,"time.unit")
## [1] "none"
## attr(,"Tfr.type")
## [1] "total"
## attr(,"radiation.unit")
## [1] "transmittance average"

transmittance(polyester.new.spct, UV_bands(), quantity = "total")

##  UVB.ISO  UVA.ISO
##   0.2685  66.5280
## attr(,"time.unit")
## [1] "none"
## attr(,"Tfr.type")
## [1] "total"
## attr(,"radiation.unit")
## [1] "transmittance total"
```

```
transmittance(polyester.new.spct, UV_bands(), quantity = "contribution")

##      UVB.ISO      UVA.ISO
## 0.0006178763 0.1530952496
## attr(,"time.unit")
## [1] "none"
## attr(,"Tfr.type")
## [1] "total"
## attr(,"radiation.unit")
## [1] "transmittance contribution"

transmittance(polyester.new.spct, UV_bands(), quantity = "contribution.pc")

##     UVB.ISO     UVA.ISO
##  0.06178763 15.30952496
## attr(,"time.unit")
## [1] "none"
## attr(,"Tfr.type")
## [1] "total"
## attr(,"radiation.unit")
## [1] "transmittance contribution.pc"

transmittance(polyester.new.spct, UV_bands(), quantity = "relative")

##     UVB.ISO     UVA.ISO
## 0.004019672 0.995980328
## attr(,"time.unit")
## [1] "none"
## attr(,"Tfr.type")
## [1] "total"
## attr(,"radiation.unit")
## [1] "transmittance relative"

transmittance(polyester.new.spct, UV_bands(), quantity = "relative.pc")

##    UVB.ISO    UVA.ISO
##  0.4019672 99.5980328
## attr(,"time.unit")
## [1] "none"
## attr(,"Tfr.type")
## [1] "total"
## attr(,"radiation.unit")
## [1] "transmittance relative.pc"

transmittance(polyester.new.spct, UV_bands(), quantity = "average")

##     UVB.ISO     UVA.ISO
## 0.007671429 0.782682353
## attr(,"time.unit")
## [1] "none"
## attr(,"Tfr.type")
## [1] "total"
## attr(,"radiation.unit")
## [1] "transmittance average"
```

It is more likely that we would like to calculate these values with reference to

light of a certain spectral quality. This needs to be calculated by hand, which is not difficult. For example, for UV-B, which we can calculate, either by trimming the waveband as shown here, or by extending the sun spectrum with zeros.

```
tr.UVB <- trim_waveband(UVB(), sun.spct, trim = TRUE)
irrad(sun.spct * polyester.new.spct, tr.UVB) /
                            irrad(sun.spct, tr.UVB) * 100

##  UVB.ISO.tr.lo
##       1.970182
## attr(,"time.unit")
## [1] "second"
## attr(,"radiation.unit")
## [1] "energy irradiance total"
```

And for a list of wavebands, as percentages.

```
irrad(sun.spct * polyester.new.spct, list(UVB(), UVA(), PAR()), wb.trim = TRUE) /
  irrad(sun.spct, list(UVB(), UVA(), PAR()), wb.trim = TRUE) * 100

##  UVB.ISO.tr.lo        UVA.ISO            PAR
##       1.970182      81.876793      92.059521
## attr(,"time.unit")
## [1] "second"
## attr(,"radiation.unit")
## [1] "energy irradiance total"
```

3.15 Calculating integrated response

The functions `response`, `e_response` and `q_response` take response.spct objects as arguments, and return the integrated value for each waveband (integrated over wavelength) **assuming** a light source with a flat spectral energy or photon output respectively.

If no waveband is supplied as argument, the whole spectrum is integrated.

```
response(Vital_BW_20.spct)

##    Total
## 20.00984
## attr(,"radiation.unit")
## [1] "energy response total"

e_response(Vital_BW_20.spct)

##    Total
## 20.00984
## attr(,"radiation.unit")
## [1] "energy response total"
```

```
q_response(Vital_BW_20.spct) * 1e-6
```

```
##    Total
## 8.174317
## attr(,"radiation.unit")
## [1] "photon response total"
```

When a waveband, or list of wavebands, is supplied the response is calculated for the wavebands.

```
e_response(Vital_BW_20.spct, UVB())

##  UVB.ISO
## 18.84361
## attr(,"radiation.unit")
## [1] "energy response total"

q_response(Vital_BW_20.spct, UVB()) * 1e-6

##  UVB.ISO
## 7.681362
## attr(,"radiation.unit")
## [1] "photon response total"
```

```
e_response(Vital_BW_20.spct, list(UVB(), UVA()))

##        UVB.ISO  UVA.ISO.tr.hi
##     18.8436051      0.1532823
## attr(,"radiation.unit")
## [1] "energy response total"
```

This function has an additional argument `quantity`, with default `"total"`, which can take values controlling the output.

```
response(Vital_BW_20.spct, UV_bands())

##  UVC.ISO.tr.lo        UVB.ISO  UVA.ISO.tr.hi
##      1.0129557     18.8436051      0.1532823
## attr(,"radiation.unit")
## [1] "energy response total"

response(Vital_BW_20.spct, UV_bands(), quantity = "total")

##  UVC.ISO.tr.lo        UVB.ISO  UVA.ISO.tr.hi
##      1.0129557     18.8436051      0.1532823
## attr(,"radiation.unit")
## [1] "energy response total"

response(Vital_BW_20.spct, UV_bands(), quantity = "contribution")

##  UVC.ISO.tr.lo        UVB.ISO  UVA.ISO.tr.hi
##    0.050622869    0.941716787    0.007660344
## attr(,"radiation.unit")
## [1] "energy response contribution"

response(Vital_BW_20.spct, UV_bands(), quantity = "contribution.pc")
```

```
##  UVC.ISO.tr.lo        UVB.ISO  UVA.ISO.tr.hi
##      5.0622869     94.1716787      0.7660344
## attr(,"radiation.unit")
## [1] "energy response contribution.pc"

response(Vital_BW_20.spct, UV_bands(), quantity = "relative")

##  UVC.ISO.tr.lo        UVB.ISO  UVA.ISO.tr.hi
##    0.050622869    0.941716787    0.007660344
## attr(,"radiation.unit")
## [1] "energy response relative"

response(Vital_BW_20.spct, UV_bands(), quantity = "relative.pc")

##  UVC.ISO.tr.lo        UVB.ISO  UVA.ISO.tr.hi
##      5.0622869     94.1716787      0.7660344
## attr(,"radiation.unit")
## [1] "energy response relative.pc"

response(Vital_BW_20.spct, UV_bands(), quantity = "average")

##  UVC.ISO.tr.lo        UVB.ISO  UVA.ISO.tr.hi
##    0.101295567    0.538388717    0.002128921
## attr(,"radiation.unit")
## [1] "energy response average"
```

If we would like to calculate these values with reference to light of a certain spectral irradiance. This can be achieved by multiplying the sensor's spectral responsivity by the light source spectral irradiance.

```
e_response(sun.spct * Vital_BW_20.spct, UVB())

##  UVB.ISO.tr.lo
##     0.05566856
## attr(,"radiation.unit")
## [1] "energy response total"

q_response(sun.spct * Vital_BW_20.spct, UVB()) * 1e-6

##  UVB.ISO.tr.lo
##     0.02168264
## attr(,"radiation.unit")
## [1] "photon response total"
```

And for a list of wavebands

```
q_response(sun.spct * KIPP_PQS1_PAR_quantum.spct, list(UVA(), PAR())) * 1e-6

##  UVA.ISO.tr.lo            PAR
##       0.177506      42.570282
## attr(,"radiation.unit")
## [1] "photon response total"
```

3.16 Integrating a generic spectrum

In some cases we may want to integrate the values of arbitrary columns other than `w.length` in an spectral object. All spectral classes are derived from `generic.spct`, so the examples in this section apply to objects of any of the 'child' spectral classes as well.

```
integrate_spct(sun.spct)

##      e.irrad      q.irrad
## 2.691249e+02 1.255336e-03

integrate_spct(sun.spct * UVA())

## e.irrad
## 27.9836

e_irrad(sun.spct, UVA())

##  UVA.ISO
## 27.73655
## attr(,"time.unit")
## [1] "second"
## attr(,"radiation.unit")
## [1] "energy irradiance total"
```

The function `integrate_spct` integrates every column holding numeric values from a spectrum object, except for `w.length`.

```
my.sun.spct <- copy(sun.spct)
my.sun.spct[ , one := 1L]

##      w.length    s.e.irrad    s.q.irrad one
##   1:      293 2.609665e-06 6.391730e-12   1
##   2:      294 6.142401e-06 1.509564e-11   1
##  ---
## 507:      799 4.185850e-01 2.795738e-06   1
## 508:      800 4.069055e-01 2.721132e-06   1

integrate_spct(my.sun.spct)

##      e.irrad      q.irrad          one
## 2.691249e+02 1.255336e-03 5.070000e+02

spread(sun.spct)

## [1] 507
```

In the simple example above, the integral of `one` gives us the span in nanometres of the spectrum.

3.17 Tagging observations in a spectrum

The function `tag` can be used to tag different parts of a spectrum according to wavebands.

```
tag(sun.spct, PAR(), byref = FALSE)

##      w.length    s.e.irrad    s.q.irrad wl.color wb.f
##   1:      293 2.609665e-06 6.391730e-12  #000000   NA
##   2:      294 6.142401e-06 1.509564e-11  #000000   NA
##  ---
## 509:      799 4.185850e-01 2.795738e-06  #000000   NA
## 510:      800 4.069055e-01 2.721132e-06  #000000   NA

tag(sun.spct, UV_bands(), byref = FALSE)

##      w.length    s.e.irrad    s.q.irrad wl.color      wb.f
##   1:      293 2.609665e-06 6.391730e-12  #000000 UVB.tr.lo
##   2:      294 6.142401e-06 1.509564e-11  #000000 UVB.tr.lo
##  ---
## 509:      799 4.185850e-01 2.795738e-06  #000000        NA
## 510:      800 4.069055e-01 2.721132e-06  #000000        NA
```

The added factor and colour data can be used for further processing or for plotting. Information about the tagging and wavebands is stored in an attribute tag.attr in every tagged spectrum, this yields a more compact output and keeps a 'trace' of the tagging.

```
tg.sun.spct <- tag(sun.spct, PAR(), byref = FALSE)
attr(tg.sun.spct, "spct.tags")

## $time.unit
## [1] "second"
##
## $wb.key.name
## [1] "Bands"
##
## $wl.color
## [1] TRUE
##
## $wb.color
## [1] TRUE
##
## $wb.num
## [1] 1
##
## $wb.colors
## $wb.colors[[1]]
##   PAR.CMF
## "#735B57"
##
##
## $wb.names
## [1] "PAR"
##
## $wb.list
## $wb.list[[1]]
## PAR
## low (nm) 400
## high (nm) 700
## weighted none
```

Additional functions are available which return a tagged spectrum and take as input a list of wavebands, but no spectral data. They 'build' an spectrum from the data in the wavebands, and are useful for plotting the boundaries of wavebands.

```
wb2tagged_spct(UV_bands())

##    w.length s.e.irrad s.q.irrad Tfr Rfl s.e.response wl.color wb.f y
## 1:  99.9999         0         0   0   0            0  #000000   NA 0
## 2: 100.0000         0         0   0   0            0  #000000  UVC 0
## 3: 279.9999         0         0   0   0            0  #000000  UVC 0
## 4: 280.0000         0         0   0   0            0  #000000  UVB 0
## 5: 314.9999         0         0   0   0            0  #000000  UVB 0
## 6: 315.0000         0         0   0   0            0  #000000  UVA 0
## 7: 399.9999         0         0   0   0            0  #03001E  UVA 0
## 8: 400.0000         0         0   0   0            0  #03001E   NA 0

wb2rect_spct(UV_bands())

##    w.length s.e.irrad s.q.irrad Tfr Rfl s.e.response wl.color wb.f wl.high
## 1:    190.0         0         0   0   0            0  #000000  UVC     280
## 2:    297.5         0         0   0   0            0  #000000  UVB     315
## 3:    357.5         0         0   0   0            0  #000000  UVA     400
##    wl.low y
## 1:    100 0
## 2:    280 0
## 3:    315 0
```

Function wb2tagged_spct returns a tagged spectrum, with two rows for each waveband, corresponding to the low and high wavelength boundaries, while function wb2rect_spct returns a spectrum with only one row per waveband, with w.length set to its midpoint but with additional columns xmin and xmax corresponding to the low and high wavelength boundaries of the wavebands.

Function is.tagged can be used to query if an spectrum is tagged or not, and function untag removes the tags.

```
tg.sun.spct

##      w.length    s.e.irrad    s.q.irrad wl.color wb.f
##   1:      293 2.609665e-06 6.391730e-12  #000000   NA
##   2:      294 6.142401e-06 1.509564e-11  #000000   NA
##  ---
## 509:      799 4.185850e-01 2.795738e-06  #000000   NA
## 510:      800 4.069055e-01 2.721132e-06  #000000   NA

is.tagged(tg.sun.spct)

## [1] TRUE

untag(tg.sun.spct)

##      w.length    s.e.irrad    s.q.irrad
##   1:      293 2.609665e-06 6.391730e-12
##   2:      294 6.142401e-06 1.509564e-11
##  ---
## 509:      799 4.185850e-01 2.795738e-06
## 510:      800 4.069055e-01 2.721132e-06
```

```
is.tagged(tg.sun.spct)

## [1] FALSE
```

In the chuck above, we can see how this works, using in this case the default byref = TRUE which adds the tags in place, or "by reference", to the spct object supplied as argument.

In the chunk bellow, we demonstrate that if an already tagged spectrum is re-tagged, the old tags are replaced with new ones, with a warning.

```
tag(tg.sun.spct, PAR())

##      w.length    s.e.irrad    s.q.irrad wl.color wb.f
##   1:      293 2.609665e-06 6.391730e-12  #000000   NA
##   2:      294 6.142401e-06 1.509564e-11  #000000   NA
##  ---
## 509:      799 4.185850e-01 2.795738e-06  #000000   NA
## 510:      800 4.069055e-01 2.721132e-06  #000000   NA

tag(tg.sun.spct, VIS())

## Warning in tag.generic.spct(tg.sun.spct, VIS()):  Overwriting old tags in
spectrum

##      w.length    s.e.irrad    s.q.irrad wl.color wb.f
##   1:      293 2.609665e-06 6.391730e-12  #000000   NA
##   2:      294 6.142401e-06 1.509564e-11  #000000   NA
##  ---
## 511:      799 4.185850e-01 2.795738e-06  #000000   NA
## 512:      800 4.069055e-01 2.721132e-06  #000000   NA
```

3.18 Calculating weighted spectral irradiances

This calculation is not very frequently used, but it is very instructive to look at spectral data in this way, as it can make apparent the large effect that small measuring errors can have on the estimated effective irradiances or exposures.

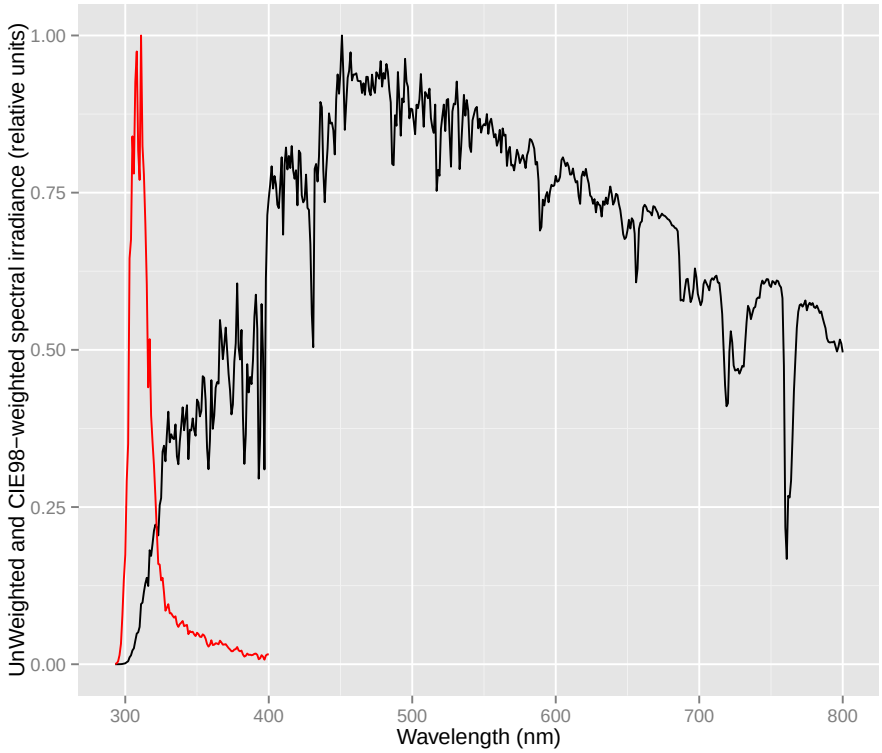
3.18.1 Weighted spectral irradiance from spectrum objects

The multiplication operator is defined for operations between a `source.spct` and a `waveband`, so this is the easiest way of doing the calculations.

```
sun.CIE.spct <- sun.spct * CIE()
sun.CIE.spct

##      w.length s.e.response
##   1:  293.000 2.609665e-06
##   2:  294.000 6.142401e-06
##  ---
## 107:  399.000 7.378801e-05
## 108:  399.999 7.395674e-05
```

We here plot, using `ggplot2`, weighted (in red) and unweighted irradiances using simulated solar spectral irradiance data stored as a source.spct object, and applying the BSWF weights on the fly.

```
ggplot(data = sun.spct, aes(x = w.length, y = s.e.irrad/max(s.e.irrad))) +
  geom_line() +
  geom_line(data = sun.spct * CIE(), colour = "red",
            aes(y = s.e.response/max(s.e.response))) +
  labs(x = "Wavelength (nm)",
       y = "UnWeighted and CIE98-weighted spectral irradiance (relative units)")
```

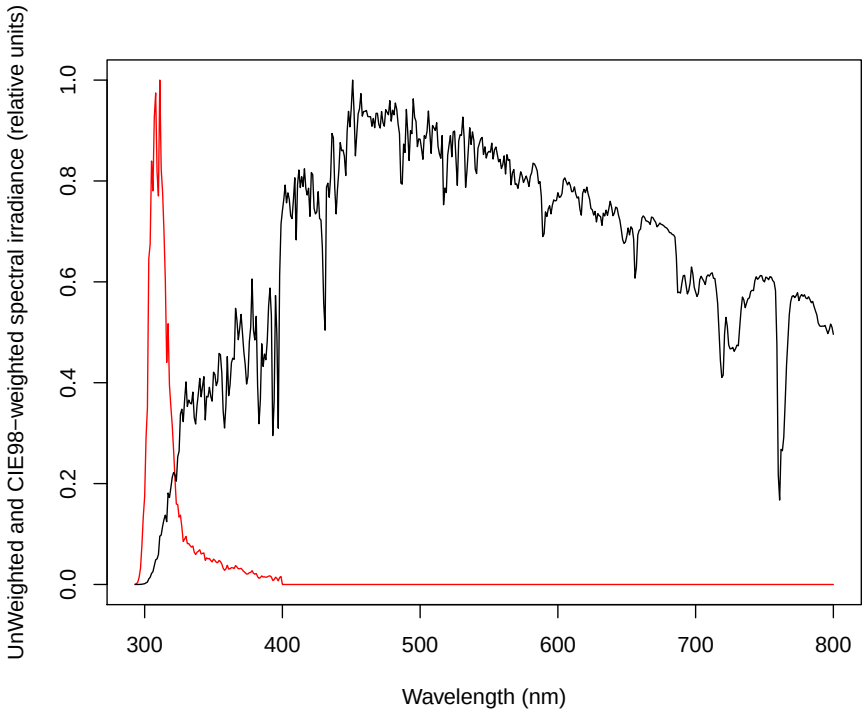


3.18.2 Weighted spectral irradiance from vectors

It is also possible to use vectors, and base R plot functions.

```
weighted.s.e.irrad <-
  with(sun.spct,
       s.e.irrad * calc_multipliers(w.length, CIE())
  )
plot(weighted.s.e.irrad/max(weighted.s.e.irrad) ~ w.length, type = "l",
     data = sun.spct,
     col = "red",
     xlab = "Wavelength (nm)",
     ylab = "UnWeighted and CIE98-weighted spectral irradiance (relative units)")
```

```r
lines(s.e.irrad/max(s.e.irrad) ~ w.length, col = "black", data = sun.spct)
```



3.19 Auxiliary functions for manipulation of spectra

To stack spectral objects the function `rbindspct` should be used instead of `rbind` or `rbindlist` from package `data.table`. The functions from package `data.table` strip the spectral class attributes from the output, returning always a `data.table` object.

`subset` has the same problem, and in addition also removes comments, so it should be also avoided, and `trim_spct` used instead.

In earlier versions of the package we had included our own version of `rbindlist` to override the one defined in package `data.table`, but this was triggering warnings and causing other problems. Because of this it has been removed. (This is a, hopefully, temporary limitation.)

Sometimes it is needed to add (parallel sum) between two spectra, even if the two spectra have been measured at different wavelengths or wavelength steps. This can happen for example when we would like to calculate the spectrum of a combination of two light sources from the individual spectra.

A function `interpolate_spectrum` is also included to facilitate interpolation of spectral values. It is used internally, but can also be used by itself when interpolation is needed. Under the hood it uses R's `spline` function if there are fewer than 25 data points, and uses `approx` otherwise. It allows easier control of values to be used for extrapolation.

3.20 Dealing with real 'noisy' spectral data

The first thing to do is to think whether any part of the spectral measurements can be *a priori* known to be equal to zero. For example for the solar spectrum at ground level it is safe to assume that the spectral irradiance is zero for all wavelengths shorter than 290 nm. If the data are noisy, it is best to discard these data before calculating any effective UV doses.

Another possibility is do smoothing of the spectral data using one a series of possible algorithms. Smoothing can distort the spectrum becasue distinguishing between real peaks and valleys from noise is difficult.

A third possibility is, when replicate measurements are available, to calculate "parallel" means, medians or other summary quantities, at each value of wavelength.

We will discuss these three approaches in each of the sections below.

3.20.1 Trimming of regions known a priori to contain only noise

In the following example we use a longer wavelength (297 nm) just to show how the function works, because the example spectral data set starts at 293 nm.

```
head(sun.spct, 2L)

##    w.length    s.e.irrad    s.q.irrad
## 1:      293 2.609665e-06 6.391730e-12
## 2:      294 6.142401e-06 1.509564e-11
```

Sub-setting can be easily done as follows if the data are in a data.frame (of course, replacing `w.length` with the name used in your data frame for the wavelengths array):

```
trimmed.sun.spct <- trim_spct(sun.spct, low.limit = 297)
head(trimmed.sun.spct, 2L)

##    w.length    s.e.irrad    s.q.irrad
## 1:      297 0.0001533491 3.807181e-10
## 2:      298 0.0003669677 9.141345e-10
```

The code above deletes the data outside the limits. However, if we supply a different value than the default NULL for the parameter `fill`, the w.length values are kept, and the trimmed spectral irradiance values replaced by the value supplied.

```
trimmed.sun.spct <- trim_spct(sun.spct, low.limit = 297, fill = NA)
head(trimmed.sun.spct, 2L)

##    w.length s.e.irrad s.q.irrad
## 1:      293        NA        NA
## 2:      294        NA        NA
```

The code above sets the spectral irradiance values for waveelngths outside the limits to `NA`, but, for example when plotting, it is useful to replace the noise in the spectrum with zeros.

```
trimmed.sun.spct <- trim_spct(sun.spct, low.limit = 297, fill = 0)
head(trimmed.sun.spct, 2L)

##    w.length s.e.irrad s.q.irrad
## 1:      293         0         0
## 2:      294         0         0
```

After 'cleaning' the data we just use the trimmed ($\approx$ sub-setted) spectral data object in further calculations or plotting.

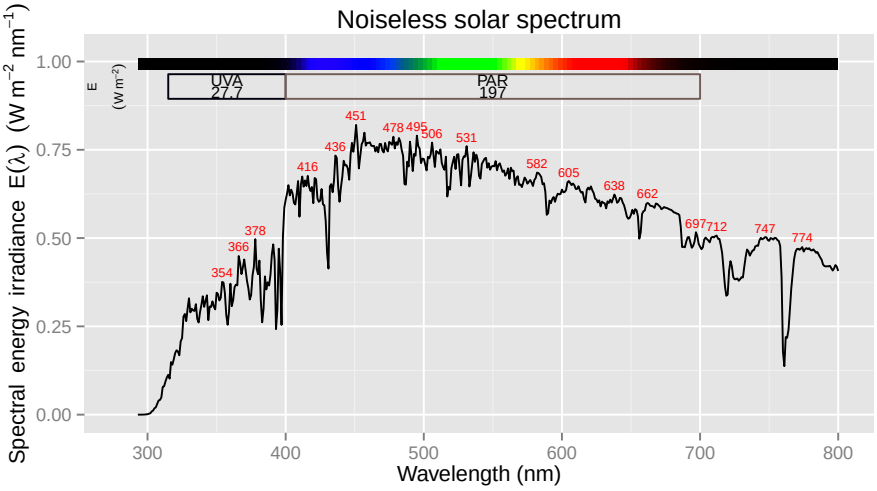
If the data are in a data frame, instead of a spct object then `subset` or indexing can be used. If the data are available as vectors, different options: 1) create a data frame from your data, 2) use the function `trim_tails()` from this package, or 3) just use R commands. Here we give examples of the use of `trim_tails()`, using the same data as in earlier examples. First we 'trim' (delete) all data for wavelengths shorter than 293 nm.

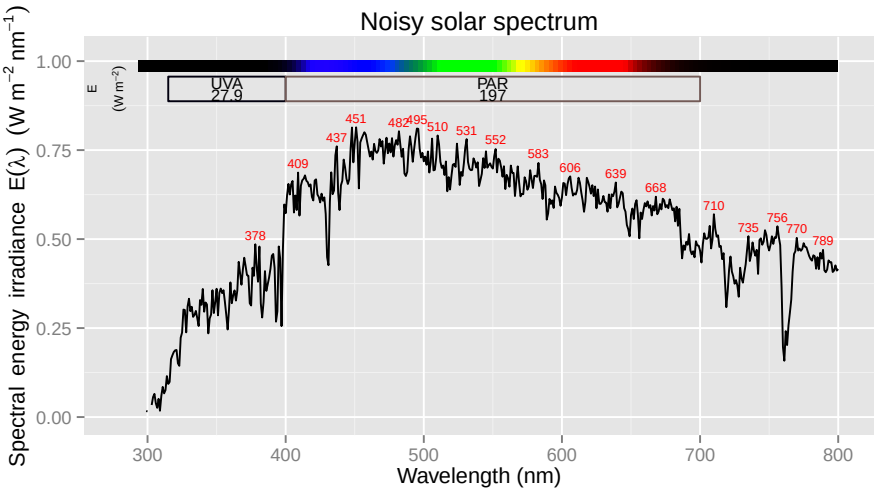
3.20.2 Smoothing of spectral data

Function `smooth_spct` can be used to smoothen noise in spectra. Smoothing is effective in removing noise, but in case of spectra with a fine structure like the one for sunlight, the details of real peaks and valleys are also smoothed out. Smoothing should be used with great care as it can cause bias and distort the shape of spectra.

We first generate a noisy solar spectrum by adding random noise to a noiseless solar spectrum. We will use this data to demonstrate smoothing.

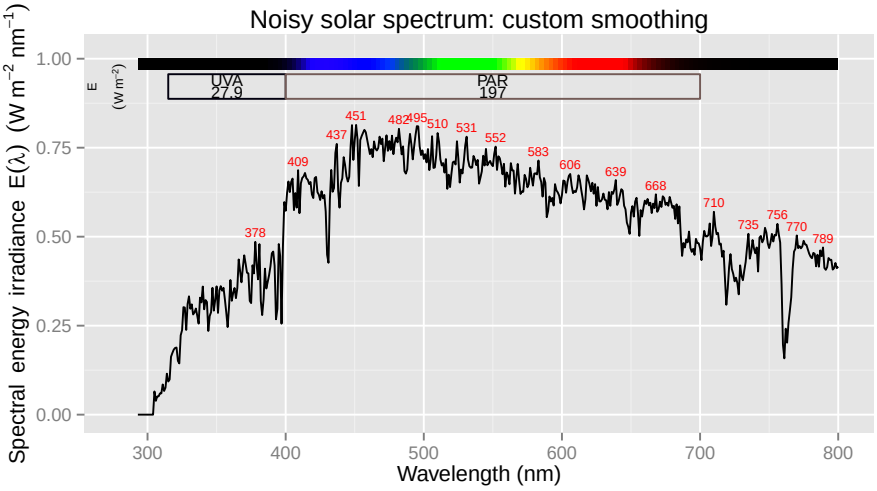
```
noisy.sun.spct <- sun.spct +
  rnorm(length(sun.spct$w.length), sd = 0.04) *
  irrad(sun.spct, quantity = "average")
plot(sun.spct) + labs(title = "Noiseless solar spectrum")
plot(noisy.sun.spct) + labs(title = "Noisy solar spectrum")

## Warning:  Removed 1 rows containing missing values (geom_path).
```





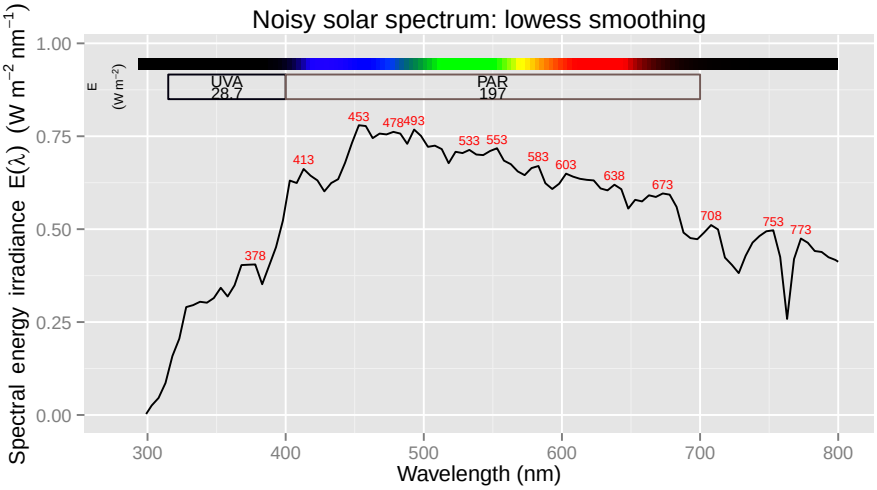
The default `"custom''` method is our own, and is suitable for small amounts of noise, as it only applies smoothing to low signal regions of the spectrum, and also forces to zero those regions which are 'detected' to contain mostly noise. The strength parameter should be used to adjust the sensitivity to noise according to the signal-to-noise ratio in the spectral data. This algorithm is quite safe, and tends to preserve most of the fine structure of spectra.

```
plot(smooth_spct(noisy.sun.spct))  +
  labs(title = "Noisy solar spectrum: custom smoothing")
```

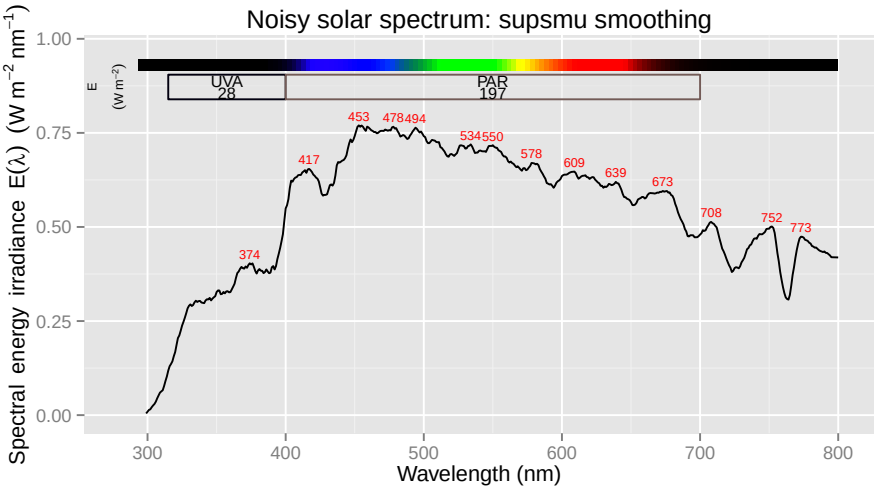


Methods `"lowess"` and `"supsmu"` are general purpose methods, which with their default values for the parameters tend to smooth spectra very agresively. They remove a significant portion of the spectral detail but could be useful when the data is very noisy or when the overall shape of a spectrum is of interest rather than the finer structure.

```
plot(smooth_spct(noisy.sun.spct, method = "lowess"))  +
  labs(title = "Noisy solar spectrum: lowess smoothing")

## Warning in range_check(x, strict.range = strict.range):  Negative spectral
energy irradiance values; minimun s.e.irrad = -0.0032
```

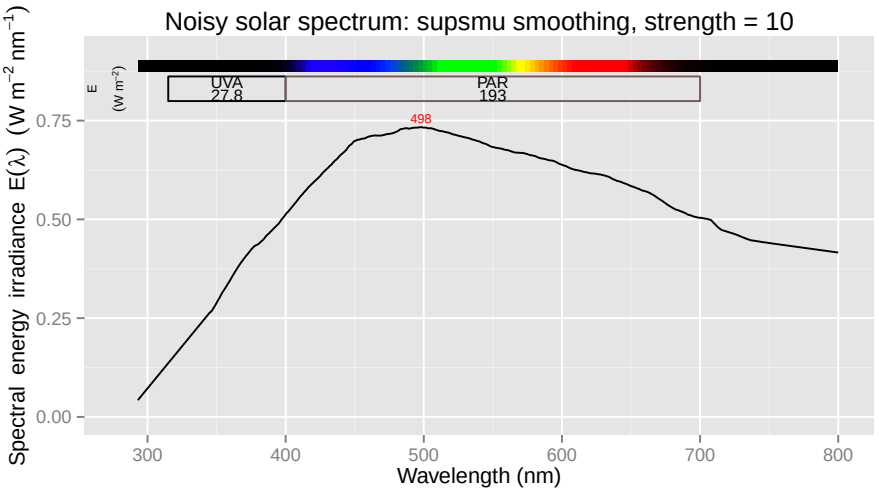


```
plot(smooth_spct(noisy.sun.spct, method = "supsmu"))  +
  labs(title = "Noisy solar spectrum: supsmu smoothing")

## Warning in range_check(x, strict.range = strict.range):  Negative spectral
energy irradiance values; minimun s.e.irrad = -0.0055
## Warning:  Removed 6 rows containing missing values (geom_path).
```



Stronger or weaker smoothing is also possible.

```
plot(smooth_spct(noisy.sun.spct, method = "supsmu", strength = 10))  +
  labs(title = "Noisy solar spectrum: supsmu smoothing, strength = 10")
```



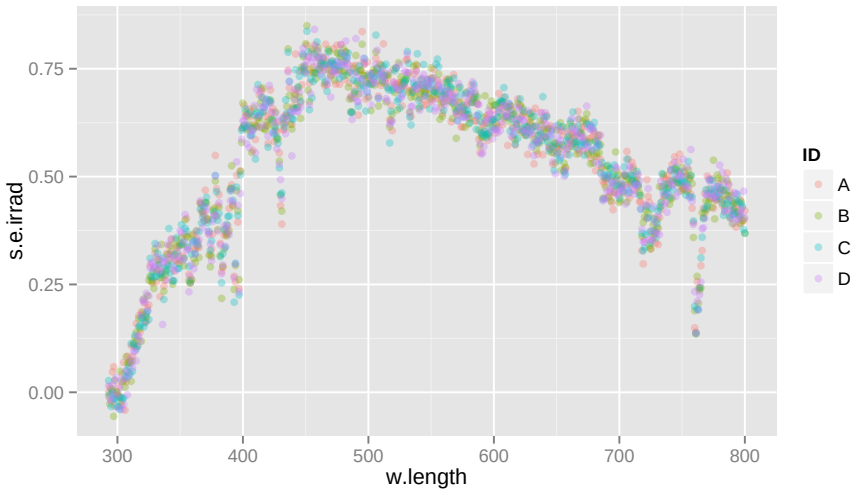
Function `smooth_spct` is generic with specializtions for `source.spct`, `filter.spct`, `reflector.spct`, and `response.spct`.

3.20.3 Parallel averaging and other parallel summaries

It is quite easy to calculate 'parallel' summary quantities using `data.table` syntax and function `rbindspct`. However, one should be careful with the handling of `NA` values, and specially make sure that all spectra have values for spectral irradiance at the same wavelengths. Similar code to that given in this section, using `source.spct` objects, can be used for `filter.spct`, `reflector.spct`, and `response.spct` objects.

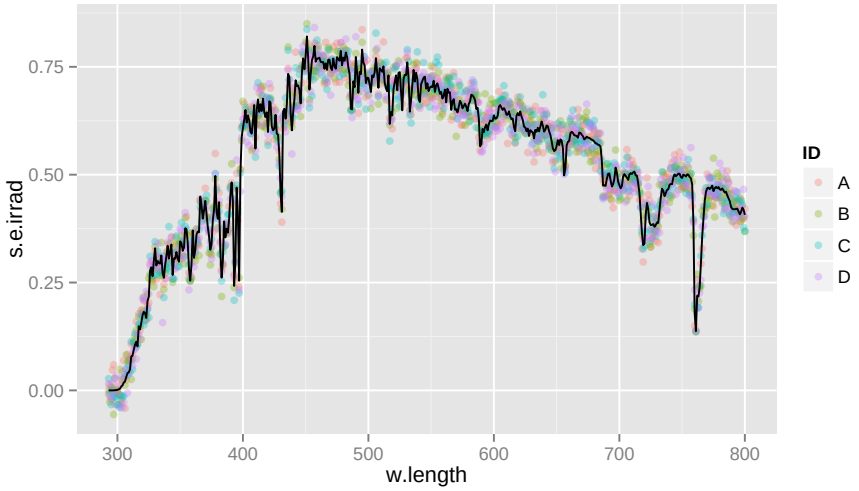
For the `source.spct` example we generate four independent noisy replicates of the same solar spectrum, to demonstrate parallel summaries. Under normal use we would use `rbindspct` to bind the different true replicate measured spectra.

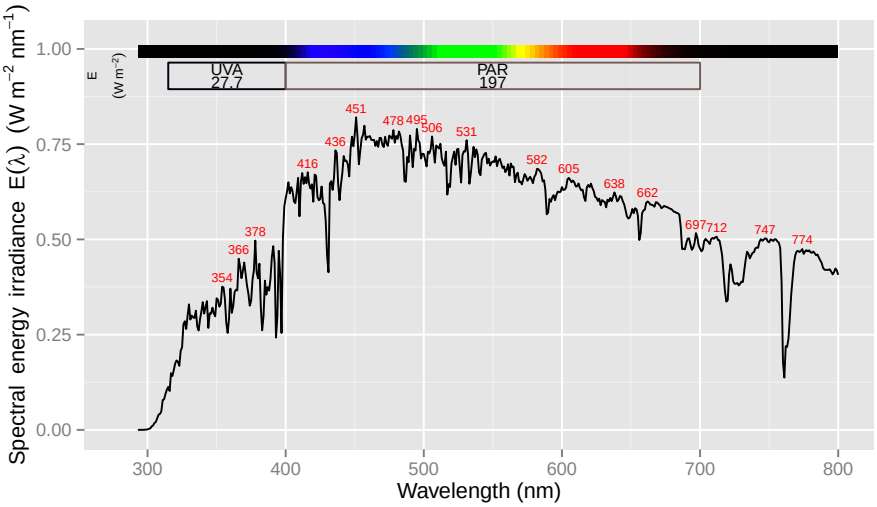
```
bound.spct <- rbindspct(list(sun.spct, sun.spct, sun.spct, sun.spct),
                        idfactor = "ID")

noisy.sun.spct <- bound.spct +
  rnorm(length(bound.spct$w.length), sd = 0.05) *
  irrad(sun.spct, quantity = "average")

raw.data.fig <- ggplot(noisy.sun.spct, aes(w.length, s.e.irrad, colour = ID)) +
  geom_point(alpha = 0.33)
raw.data.fig
```




```
mean.spct <- bound.spct[ , .(s.e.irrad = mean(s.e.irrad, na.rm = TRUE)),
                         by = w.length]
setSourceSpct(mean.spct)
mean.spct

##      w.length    s.e.irrad
##   1:      293 2.609665e-06
##   2:      294 6.142401e-06
##  ---
```

```
## 507:       799 4.185850e-01
## 508:       800 4.069055e-01

mean.fig <- raw.data.fig + geom_line(data = mean.spct, colour = "black")
mean.fig

plot(mean.spct)
```






```
median.spct <- bound.spct[ , .(s.e.irrad = median(s.e.irrad, na.rm = TRUE)),
                           by = w.length]
setSourceSpct(median.spct)
median.spct

##      w.length    s.e.irrad
```

```
##   1:      293 2.609665e-06
##   2:      294 6.142401e-06
##  ---
## 507:      799 4.185850e-01
## 508:      800 4.069055e-01
```

If the wavelength values in the different spectra are not the same, or if NAs are present, this slightly more complex code will make diagnosis of any problems much easier, and the resulting spectrum will still behave as a `source.spct` object when applying any other functions.

```
meanx.spct <- bound.spct[ , .(s.e.irrad = mean(s.e.irrad, na.rm = TRUE),
                              sd = sd(s.e.irrad, na.rm = TRUE),
                              n = length(na.omit(s.e.irrad)),
                              n.na = sum(is.na(s.e.irrad))
                              ), by = w.length]
setSourceSpct(meanx.spct)
meanx.spct
```

```
##      w.length    s.e.irrad sd n n.na
##   1:      293 2.609665e-06  0 4    0
##   2:      294 6.142401e-06  0 4    0
##  ---
## 507:      799 4.185850e-01  0 4    0
## 508:      800 4.069055e-01  0 4    0
```

```
medianx.spct <- bound.spct[ , .(s.e.irrad = median(s.e.irrad, na.rm = TRUE),
                                mad = mad(s.e.irrad, na.rm = TRUE),
                                n = length(na.omit(s.e.irrad)),
                                n.na = sum(is.na(s.e.irrad))
                                ), by = w.length]
setSourceSpct(medianx.spct)
medianx.spct
```

```
##      w.length    s.e.irrad mad n n.na
##   1:      293 2.609665e-06   0 4    0
##   2:      294 6.142401e-06   0 4    0
##  ---
## 507:      799 4.185850e-01   0 4    0
## 508:      800 4.069055e-01   0 4    0
```

The two code chunks above can be easilly modified as needed, but they do not preserve all the attributes of the original spectra.

4 Astronomical calculations

4.1 Position of the sun in the sky

In photobiology research we sometimes need to calculate the position on the sun at arbitrary locations and positions. The function `sun_angles` returns the azimuth in degrees eastwards, altitude in degrees above the horizon, solar disk diameter in degrees and sun to earth distance in astronomical units. The time

should be a `POSIXct` vector, possibly of length one, and it is easiest to use package `lubridate` for working with time and dates.

```
sun_angles(now(), lat = 34, lon = 0)

## $time
## [1] "2015-02-17 22:49:28 EET"
##
## $azimuth
## [1] 282.7896
##
## $elevation
## [1] -38.62033
##
## $diameter
## [1] 0.5395772
##
## $distance
## [1] 0.988181

sun_angles(ymd_hms("2014-01-01 0:0:0", tz = "UTC"))

## $time
## [1] "2014-01-01 UTC"
##
## $azimuth
## [1] 181.9507
##
## $elevation
## [1] -66.96255
##
## $diameter
## [1] 0.5422513
##
## $distance
## [1] 0.9833078
```

4.2 Calculating times of sunrise and sunset

```
day_night()

## $day
## [1] "2015-02-17 EET"
##
## $sunrise
## [1] "2015-02-17 08:14:01 EET"
##
## $noon
## [1] "2015-02-17 14:13:57 EET"
##
## $sunset
## [1] "2015-02-17 20:13:59 EET"
##
## $daylength
```

```
## Time difference of 11.99941 hours
##
## $nightlength
## Time difference of 12.00059 hours

day_night(ymd("2014-05-30", tz = "UTC"), lat = 30, lon = 0)

## $day
## [1] "2014-05-30 UTC"
##
## $sunrise
## [1] "2014-05-30 08:04:12 EEST"
##
## $noon
## [1] "2014-05-30 14:57:32 EEST"
##
## $sunset
## [1] "2014-05-30 21:51:05 EEST"
##
## $daylength
## Time difference of 13.78135 hours
##
## $nightlength
## Time difference of 10.21865 hours

day_night(ymd("2014-05-30", tz = "UTC"), lat = 30, lon = 0, twilight = "civil")

## $day
## [1] "2014-05-30 UTC"
##
## $sunrise
## [1] "2014-05-30 08:34:25 EEST"
##
## $noon
## [1] "2014-05-30 14:57:32 EEST"
##
## $sunset
## [1] "2014-05-30 21:20:49 EEST"
##
## $daylength
## Time difference of 12.77342 hours
##
## $nightlength
## Time difference of 11.22658 hours
```

5 Calculating equivalent RGB colours for display

Two functions allow calculation of simulated colour of light sources as R colour definitions. Three different functions are available, one for monochromatic light taking as argument wavelength values, and one for polychromatic light taking as argument spectral energy irradiances and the corresponding wave length values. The third function can be used to calculate a representative RGB colour

for a band of the spectrum represented as a range of wavelength, based on the assumption of a flat energy irradiance across the range. By default CIE coordinates for *typical* human vision are used, but the functions have a parameter that can be used for supplying a different chromaticity definition.

Examples for monochromatic light:

```
w_length2rgb(550) # green

## wl.550.nm
## "#00FF00"

w_length2rgb(630) # red

## wl.630.nm
## "#FF0000"

w_length2rgb(380) # UVA

## wl.380.nm
## "#000000"

w_length2rgb(750) # far red

## wl.750.nm
## "#000000"

w_length2rgb(c(550, 630, 380, 750)) # vectorized

## wl.550.nm wl.630.nm wl.380.nm wl.750.nm
## "#00FF00" "#FF0000" "#000000" "#000000"
```

Examples for wavelength ranges:

```
w_length_range2rgb(c(400,700))

## 400-700 nm
##  "#735B57"

w_length_range2rgb(400:700)

## Using only extreme wavelength values.

## 400-700 nm
##  "#735B57"

w_length_range2rgb(sun.spct$w.length)

## Using only extreme wavelength values.

## 293-800 nm
##  "#554340"

w_length_range2rgb(550)

## Calculating RGB values for monochromatic light.

## wl.550.nm
## "#00FF00"
```

Examples for spectra as vectors, in this case for the solar spectrum:

```
with(sun.spct, s_e_irrad2rgb(w.length, s.e.irrad))

## [1] "#544F4B"

with(sun.spct, s_e_irrad2rgb(w.length, s.e.irrad, sens = ciexyzCMF2.spct))

## [1] "#544F4B"

with(sun.spct, s_e_irrad2rgb(w.length, s.e.irrad, sens = ciexyzCMF10.spct))

## [1] "#59534F"

with(sun.spct, s_e_irrad2rgb(w.length, s.e.irrad, sens = ciexyzCC2.spct))

## [1] "#B63C37"

with(sun.spct, s_e_irrad2rgb(w.length, s.e.irrad, sens = ciexyzCC10.spct))

## [1] "#BD3C33"
```

Examples with `source.spct` objects.

```
rgb_spct(sun.spct)

## [1] "#544F4B"

rgb_spct(sun.spct, sens = ciexyzCMF2.spct)

## [1] "#544F4B"
```

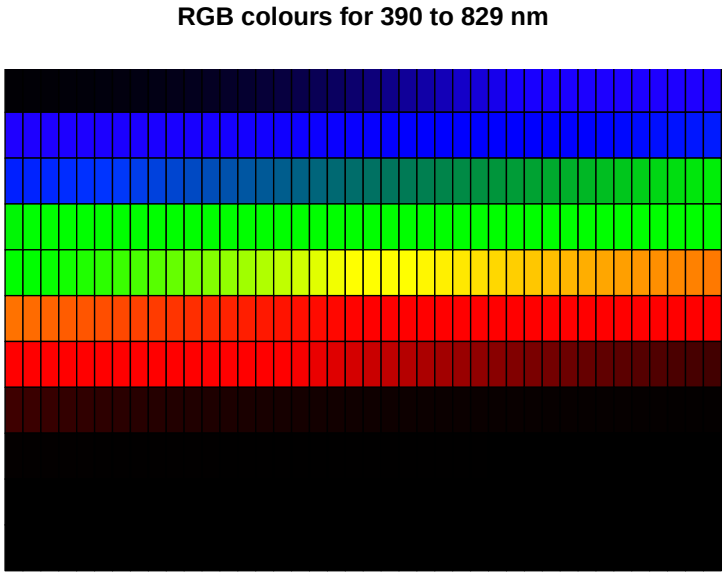
And also a `color` method for `source.spct`.

```
color(sun.spct)

## source CMF  source CC
##  "#544F4B"  "#B63C37"

color(sun.spct * rg630.spct)

## source CMF  source CC
##  "#4A0000"  "#FF0000"
```

Here we plot the RGB colours for the range covered by the CIE 2006 proposed standard calculated at each 1 nm step:

```
wl <- c(390, 829)

my.colors <- w_length2rgb(wl[1]:wl[2])

colCount <- 40 # number per row
rowCount <- trunc(length(my.colors) / colCount)

plot( c(1,colCount), c(0,rowCount), type = "n", ylab = "", xlab = "",
```

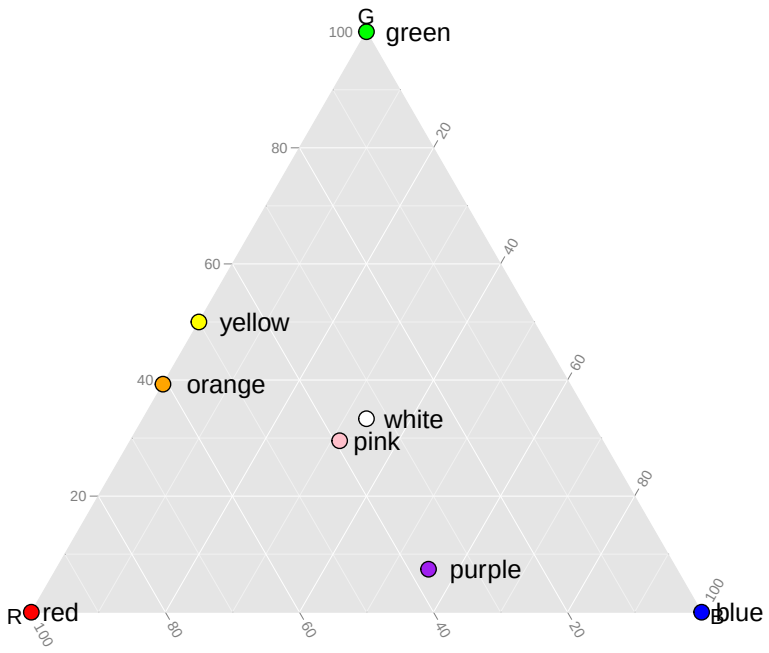
```
      axes = FALSE, ylim = c(rowCount,0))
title(paste("RGB colours for", as.character(wl[1]),
            "to", as.character(wl[2]), "nm"))

for (j in 0:(rowCount-1))
{
  base <- j*colCount
  remaining <- length(my.colors) - base
  RowSize <- ifelse(remaining < colCount, remaining, colCount)
  rect((1:RowSize)-0.5,j-0.5, (1:RowSize)+0.5,j+0.5,
       border = "black",
       col = my.colors[base + (1:RowSize)])
}
```



Given a color in any of the above ways, yields RGB values that can be used to locate the position of any colour on Maxwell's triangle. Here using R's predefined colours.

```
colours <- c("red", "green", "yellow", "white", "orange",
             "blue", "pink", "purple")
rgb.values <- col2rgb(colours)
test.data <-
  data.frame(colour = colours,
             R = rgb.values[1, ], G = rgb.values[2, ], B = rgb.values[3, ])
maxwell.tern <- ggtern(data = test.data,
```

```
                        aes(x = R, y = G, z = B, label = colour, fill = colour)) +
  geom_point(shape = 21, size = 4) + geom_text(hjust = -0.3) +
  labs(x = "R", y = "G", z = "B") + scale_fill_identity()
maxwell.tern
```



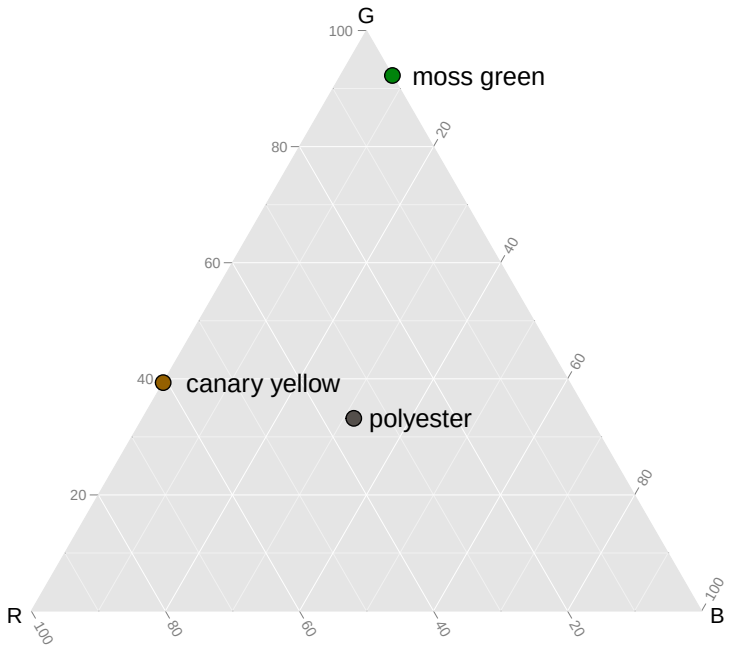
We simulate the spectra of filtered sunlight by multiplying the solar spectrum by filter transmittance spectra.

```
yellow.light.spct <- canary.yellow.new.spct * sun.spct
green.light.spct <- moss.green.new.spct * sun.spct
polyester.light.spct <- polyester.new.spct * sun.spct
```

Now using the filtered sunlight spectra we calculate colours based on human vision photoreceptors.

```
coord <- 1 # CMF
yellow.filter <- color(yellow.light.spct)[coord]
green.filter <- color(green.light.spct)[coord]
polyester.filter <- color(polyester.light.spct)[coord]
colours <- c(yellow.filter, green.filter, polyester.filter)
rgb.values <- col2rgb(colours)
test.data <- data.frame(colour = colours,
                        R = rgb.values[1, ], G = rgb.values[2, ], B = rgb.values[3, ],
                        labels = c("canary yellow", "moss green", "polyester"))
```

```
maxwell.tern <- ggtern(data = test.data,
                       aes(x = R, y = G, z = B, fill = colour, label = labels)) +
  geom_point(shape = 21, size = 4) +
  geom_text(hjust = -0.15) +
  labs(x = "R", y = "G", z = "B")  +
  scale_fill_identity()
maxwell.tern
```



6 Optimizing performance

When developing the current version of photobiology quite a lot of effort was spent in optimizing performance, especially of the functions accepting vectors as arguments, as in one of our experiments, we need to process several hundred thousands of measured spectra. The defaults should provide good performance in most cases, however, some further improvements are achievable, when a series of different calculations are done on the same spectrum, or when a series of spectra measured at exactly the same wavelengths are used for calculating weighted irradiances or exposures.

In the case of doing calculations repeatedly on the same spectrum, a small improvement in performance can be achieved by setting the parameter `check.spectrum = FALSE` for all but the first call to `irradiance()`, or

`photon_irradiance()`, or `energy_irradiance()`, or the equivalent function for ratios. It is also possible to set this parameter to FALSE in all calls, and do the check beforehand by explicitly calling `check_spectrum()`.

In the case of calculating weighted irradiances on many spectra having exactly the same wavelength values, then a significant improvement in the performance can be achieved by setting `use.cached.mult = TRUE`, as this reuses the multipliers calculated during successive calls based on the same waveband. However, to achieve this increase in performance, the tests to ensure that the wavelength values have not changed, have to be kept to the minimum. Currently only the length of the wavelength array is checked, and the cached values discarded and recalculated if the length changes. For this reason, this is not the default, and when using caching the user is responsible for making sure that the array of wavelengths has not changed between calls.

You can use the package `microbenchmark` to time the code and find the parts that slow it down. I have used it, and also I have used profiling to optimize the code for speed. The choice of defaults is based on what is best when processing a moderate number of spectra, say less than a few hundreds, as opposed to many thousands.