photobiology Version 0.7.1
User Guide

Pedro J. Aphalo

July 19, 2015

Contents

1 Introduction **1**

2 Installation and use **1**

3 Spectra **2**
3.1 Printing . . . 4
3.2 Classes . . . 5
3.3 Data assumptions . . . 5
3.4 Querying the class . . . 7
3.5 Construction . . . 8
3.5.1 Setting the class of an object . . . 8
3.5.2 Using 'as' functions . . . 16
3.5.3 Using constructors . . . 16
3.6 Special attributes . . . 18
3.6.1 Retrieving attributes . . . 19
3.6.2 Setting attributes . . . 20

4 Collections of spectra **21**
4.1 Classes . . . 21
4.2 Construction . . . 22
4.2.1 Constructors . . . 22
4.2.2 Using 'as' functions . . . 22
4.3 Querying the class . . . 22

5 Wavebands **23**
5.1 Construction . . . 23
5.2 Querying the class . . . 25
5.3 Retrieving properties . . . 25

6 Collections of wavebands **26**
6.1 Construction . . . 26
6.1.1 List constructor . . . 26
6.1.2 Special constructor . . . 26

7 Transformations: using operators **30**
7.1 Binary operators . . . 30
7.2 Unary operators and maths functions . . . 34
7.3 Options . . . 35

8 Transformations: methods and functions **37**
8.1 Row binding spectra . . . 37
8.2 Conversions between radiation units . . . 41
8.3 Normalizing a spectrum . . . 42
8.4 Rescaling a spectrum . . . 43
8.5 Wavelength interpolation . . . 44
8.6 Trimming . . . 44
8.7 Convoluting weights . . . 46
8.7.1 Individual spectra . . . 46
8.7.2 Vectors . . . 49
8.8 Tagging with bands and colours . . . 49
8.8.1 Individual spectra . . . 50

9 Summaries **52**
9.1 Summary . . . 54
9.2 Wavelength . . . 54
9.2.1 Individual spectra . . . 54
9.2.2 Collections of spectra . . . 55
9.3 Peaks and valleys . . . 56
9.3.1 Individual spectra . . . 56
9.4 Irradiance . . . 58
9.4.1 Individual spectra . . . 58
9.4.2 Collections of spectra . . . 62
9.4.3 Numeric vectors . . . 63
9.5 Fluence . . . 64
9.6 Photon and energy ratios . . . 66
9.6.1 Individual spectra . . . 66
9.6.2 Collections of spectra . . . 68
9.6.3 Vectors . . . 68
9.7 Normalized difference indexes . . . 68
9.8 Individual spectra . . . 68
9.9 Transmittance, reflectance, absorptance and absorbance . . . 69
9.9.1 Individual spectra . . . 69
9.9.2 Collections of spectra . . . 73
9.10 Integrated response . . . 73
9.10.1 Individual spectra . . . 73

9.10.2 Collections of spectra 76
9.11 Integration over wavelengths 77
9.11.1 Calculation from individual spectra 77

10 Handling 'noisy' spectral data 78
10.0.1 Trimming of noisy regions 78
10.0.2 Smoothing 79
10.0.3 Parallel averaging and summaries 83

11 Astronomy 86
11.1 Position of the sun 86
11.2 Times of sunrise, solar noon and sunset 87

12 RGB colours 90

13 Optimizing performance 95

14 Example data 96

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 plant photoreceptors (photobiologyPlants). 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.

Package photobiology 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 everyday 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.

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(photobiologyReflectors)
library(photobiologySensors)
library(photobiologyPlants)
library(ggplot2)
library(ggtern)
```

3 Spectra

This package defines a family of classes based on data tables (data frame compatible objects) which impose some restrictions on the naming of the vectors, something that allows methods and some 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 also `data.table` objects, and `data.frame` objects, as a result of how classes have been derived. In this package we define a *generic* or *base* spectrum class, derived from `data.table`, from which specialized types of spectra are in turn 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
```

```r
##   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`.

3.1 Printing

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
```

Table 1: Classes for spectral data

Name	Variables	Attributes
generic_spct	w.length	
cps_spct	w.length, cps	
source_spct	w.length, s.e.irrad, s.q.irrad	time.unit, bswf
filter_spct	w.length, Tfr, A	Tfr.type
reflector_spct	w.length, Rfr	Rfr.type
object_spct	w.length, Tfr, Rfr	Tfr.type, Rfr.type
response_spct	w.length, s.e.response, s.q.response	time.unit
chroma_spct	w.length, x, y, z	

3.2 Classes

The package defines several classes intended to be used to store different types of spectral data. They are all derived from `generic_spct`, which in turn is derived from `data.table`. Table 3 lists them.

The *design* imposes that data from different observations are never present as different *data columns*, if present, additional data columns represent different properties from the same observation event. In most cases, one spectral object corresponds to one spectral observation, but some functions are compatible or can be used to create spectral objects where the spectral data from different observations are stored "longitudinally" and "tagged" with a factor with a level for each observation event. These observations must use consistent units of expression and attribute values.

3.3 Data assumptions

An assumption of the package is that wavelengths are always expressed in nanometres (1 nm $= 1 \cdot 10^{-9}\,\mathrm{m}$). If the data to be analysed uses different units for wavelengths, e.g. Ångstrom ($1\,\text{Å} = 1 \cdot 10^{-10}\,\mathrm{m}$), the values need to be re-scaled before any calculations. Table 2 lists the units of expression for the different variables listed in Table 3.

Energy irradiances are assumed to be expressed in $\mathrm{W\,m^{-2}}$ and photon irradiances in $\mathrm{mol\,m^{-2}\,s^{-1}}$, 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* and *exposure* can be used by supplying the arguments `"day"` or `"exposure"`[1] to a parameter of the constructor of `source_spct` objects.

The attributes are normally set when an object spectral object is created, either using default values of values supplied as arguments to the constructor.

[1] The meaning of `"exposure"` is the total exposure time, in other words, fluence instead of irradiance.

Table 2: Variables used for spectral data and their units of expression: A: as stored in objects of the spectral classes, B: also recognized by the `set` family of functions for spectra and automatically converted. `time.unit` accepts in addition to the character strings listed in the table, objects of classes `lubridate::duration` and `period`, in addition `numeric` values are interpreted as seconds. `exposure.time` accepts these same values, but not the character strings.

Variables	Unit of expression	Attribute value
A: stored		
w.length	nm	
cps	$n\,s^{-1}$	
s.e.irrad	$W\,m^{-2}\,nm^{-1}$	time.unit = "second"
s.e.irrad	$J\,m^{-2}\,d^{-1}\,nm^{-1}$	time.unit = "day"
s.e.irrad	varies	time.unit = *duration*
s.q.irrad	$mol\,m^{-2}\,s^{-1}\,nm^{-1}$	time.unit = "second"
s.q.irrad	$mol\,m^{-2}\,d^{-1}\,nm^{-1}$	time.unit = "day"
s.q.irrad	$mol\,m^{-2}\,nm^{-1}$	time.unit = "exposure"
s.q.irrad	varies	time.unit = *duration*
Tfr	[0,1]	Tfr.type = "total"
Tfr	[0,1]	Tfr.type = "internal"
A	a.u.	Tfr.type = "internal"
Rfr	[0,1]	Rfr.type = "total"
Rfr	[0,1]	Rfr.type = "specular"
s.e.response	$x\,J^{-1}\,s^{-1}\,nm^{-1}$	time.unit = "second"
s.e.response	$x\,J^{-1}\,d^{-1}\,nm^{-1}$	time.unit = "day"
s.e.response	$x\,J^{-1}\,nm^{-1}$	time.unit = "exposure"
s.e.response	varies	time.unit = *duration*
s.q.response	$x\,mol^{-1}\,s^{-1}\,nm^{-1}$	time.unit = "second"
s.q.response	$x\,mol^{-1}\,d^{-1}\,nm^{-1}$	time.unit = "day"
s.q.response	$x\,mol^{-1}\,nm^{-1}$	time.unit = "exposure"
s.q.response	varies	time.unit = *duration*
x, y, z	[0,1]	
B: converted		
wl → w.length	nm	
wavelength → w.length	nm	
Tpc → Tfr	[0,100]	Tfr.type = "total"
Tpc → Tfr	[0,100]	Tfr.type = "internal"
Rpc → Rfr	[0,100]	Rfr.type = "total"
Rpc → Rfr	[0,100]	Rfr.type = "specular"
counts.per.second → cps	$n\,s^{-1}$	

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 when creating a new object.

```
# not run
my.spct <- source_spct(w.length = wavelength/10, s.e.irrad = 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.

3.4 Querying the class

Before giving examples of how to construct objects to store spectral data we show how to query the class of an object, and how to query the class of a spectrum. 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.

```r
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.

```r
class(sun.spct)

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

class(1:10)

## [1] "integer"
```

3.5 Construction

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 Setting the class of an 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.

```r
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.

```r
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.

```r
b.spct <- setSourceSpct(data.table(w.length = 300:305, s.e.irrad = 1))
getTimeUnit(b.spct)

## [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")
getTimeUnit(b.d.spct)

## [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))
getTfrType(c.spct)

## [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")
getTfrType(c.i.spct)

## [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
```

`object_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), `Tfr` (Transmittance, T, as fraction of one), and\or `Rfr` (Reflectance, R, as fraction of one) must be present.

```
e.spct <- setObjectSpct(data.table(w.length = 300:305, Tfr = 0.5, Rfr = 0.5))
class(e.spct)

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

e.spct
```

```
##    w.length Tfr Rfr
## 1:      300 0.5 0.5
## 2:      301 0.5 0.5
## 3:      302 0.5 0.5
## 4:      303 0.5 0.5
## 5:      304 0.5 0.5
## 6:      305 0.5 0.5
```

`object_spct` objects can be also created by merging a `filter_spct` object and `reflector_spct` object if they share `w.length` values.

```
merged.spct <- merge(c.spct, d.spct)
class(merged.spct)

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

merged.spct

##    w.length Tfr Rfr
## 1:      300   1   1
## 2:      301   1   1
## 3:      302   1   1
## 4:      303   1   1
## 5:      304   1   1
## 6:      305   1   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 overridden as the most likely reason for it to be triggered is the expression of wavelengths in units other than the expected nanometres (nm).

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

## Error in check.generic_spct(x, multiple.wl = multiple.wl):  Off-range minimum
w.length value 5999 instead of within 100 nm and 5000 nm
```

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.

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

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

The constructor converts percents to fractions before applying the tests.

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

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

Setting `strict.range = FALSE` issues a warning instead of an error, and returns a `reflector_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 a `reflector_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"
```

3.5.2 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.

3.5.3 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
```

Function `source_spct` accepts an argument for setting the `time.unit`, and functions `filter_spct` and `object_spct` accept an argument for setting the `Tfr.type` attributes, with the same defaults as described above. Functions `source_spct`, `filter_spct`, `reflector_spct` and `object_spct` have a `strict.range` formal argument, that alters checks as described in section 3.5.1 above.

3.6 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. `source_spct` objects have a `bswf.used` attribute which can take one of several values "none", "unknown" or the name of a BSWF. It is also possible to set the attributes for an existing object with functions `setTimeUnit` and `setBSWFUsed`, and to query the value of the attributes for an existing object with functions `getTimeUnit` and `getBSWFUsed`.

3.6.1 Retrieving attributes

The function `is_effective` returns `TRUE` if the value of the `bswf.used` is neither `NULL` nor equal to "none".

```
is_effective(sun.spct)

## [1] FALSE

is_effective(sun.spct * VIS())

## [1] FALSE

getBSWFUsed(sun.spct * VIS())

## [1] "none"

is_effective(sun.spct * CIE())

## [1] TRUE

getBSWFUsed(sun.spct * CIE())

## [1] "CIE98.298"
```

`filter_spct` and `object_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`, and to query the value of the attribute for an existing object with function `setTfrType`.

`reflector_spct` and `object_spct` objects have a `Rfr.type` attribute which can take one of two values "total" or "specular", the default being "total". However, if the spectral reflectance data is specular, meaning excluding the contribution of scattered 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 `setRfrType`, and to query the value of the attribute for an existing object with function `setRfrType`.

`source_spct` and `response_spct` objects have a `time.unit` attribute. This can contain either character strings or `duration` objects as defined in package `lubridate`. When querying the `time.unit` of an object, by default the stored value is returned, but it is possibly to force the returned object to be a `lubridate::duration`, in which case *character strings* are converted into `duration`, possibly `NA`.

```r
getTimeUnit(sun.spct)

## [1] "second"

getTimeUnit(sun.daily.spct)

## [1] "day"

getTimeUnit(sun.spct, force.duration = TRUE)

## [1] "1s"

getTimeUnit(sun.daily.spct, force.duration = TRUE)

## [1] "86400s (~1 days)"
```

3.6.2 Setting attributes

In most cases attributes are set during the constructions of spectral objects and should be changed with *set* functions only to correct earlier mistakes, or exceptionally update objects created earlier versions of the package. Although functions are available for directly setting attributes, these functions are very rarely used in user code, as changing attributes alone normally invalidates the data object. In some cases one may need to copy or restore attributes between objects that have become corrupted by their loss. In general, conversion methods should be used instead.

For example, sometimes it may be useful to change the time unit used for expresing spectral irradiance or spectral response, and this can be achieved with function `convertTimeUnit`. This function both converts spectral data to the new unit of expression and set the `time.unit` attribute, preserving the validity of the data object.

```r
ten.minutes.spct <-
  convertTimeUnit(sun.spct, time.unit = duration(10, "minutes"), byref = FALSE)
head(ten.minutes.spct)

##    w.length   s.e.irrad    s.q.irrad
## 1:      293 0.001565799 3.835038e-09
## 2:      294 0.003685440 9.057383e-09
## 3:      295 0.013057050 3.219831e-08
## 4:      296 0.040680713 1.006575e-07
## 5:      297 0.092009436 2.284309e-07
## 6:      298 0.220180621 5.484807e-07

getTimeUnit(ten.minutes.spct)

## [1] "600s (~10 minutes)"
```

Table 3: Classes for colletion of spectral objects

Name	Member objects	Attributes
generic_mspct	generic_spct	names, dim
cps_mspct	cps_spct	names, dim
source_mspct	source_spct	names, dim
filter_mspct	filter_spct	names, dim
reflector_mspct	reflector_spct	names, dim
object_mspct	object_spct	names, dim
response_mspct	response_spct	names, dim
chroma_mspct	chroma_spct	names, dim

Spectral objects created with earlier (pre-release) versions of this package are missing some attributes. For this reason 'summary' and 'plot' functions may not work as expected with them. These *old* objects can be updated by adding the missing attribute using functions `setTimeUnit`, `setBSWFUsed`, `setTfrType` and `setRfrType`. However, in many cases function `update_spct` can be used to set the missing attributes to default values, or the scripts re-run to rebuild the data objects from raw data.

4 Collections of spectra

4.1 Classes

The package defines several classes intended to be used to store *collections* of different types of spectral data. They are all derived from `generic_mspct`, which in turn is derived from `list`. Table **??** lists them.

Objects of these classes, except for those of class `generic_mspct`, can contain members belonging to one of the classes. Being all other spectral object classes derived from `generic_spct`, `generic_mspct` objects can contain heterogeneous collections of spectra. In all cases, there are no restrictions on the lengths, wavelength range and/or wavelength step size, or attributes other than `class` of the contained spectra. Mimicking R's arrays and matrixes, a `dim` attribute is always present and `dim` methods are provided. These allows the storage of time series of spectral data, or (hyper)spectral image data, or even higher dimensional spectral data. The handling of 1D and 2D spectral collections is already implemented in the summary methods. Handling of 3D and higher dimensional data can be implemented in the future without changing the class definition. By having implemented `dim`, also methods `ncol` and `nrow` are available as they use `dim` internally. Array-like subscripting is **not** implemented.

4.2 Construction

4.2.1 Constructors

We can construct a collection using a list of spectral objects as a starting point, in this case the spectral transmittance for two glass filters.

```
two_filters.mspct <- filter_mspct(list(gg400 = gg400.spct,
                                       og550 = og550.spct))
```

We can also create heterogeneous collections, but this reduces the number of methods that can be used on the resulting collection.

```
mixed.mspct <- generic_mspct(list(filter = clear.spct, source = sun.spct))
```

4.2.2 Using 'as' functions

The `as` functions for collections of spectra, not only change the class of the collection object, but also apply the corresponding as functions to the member objects. They copy the original objects and then convert the copy, which is returned.

```
two_gen.mscpt <- as.generic_mspct(two_filters.mspct)
class(two_gen.mscpt)

## [1] "generic_mspct" "matrix"

lapply(two_gen.mscpt, class_spct)

## $gg400
## [1] "generic_spct"
##
## $og550
## [1] "generic_spct"
```

4.3 Querying the class

`is.` functions are defined for these classes. R's `class` method can also be used.

```
is.filter_mspct(two_filters.mspct)

## [1] TRUE

class(two_filters.mspct)

## [1] "filter_mspct"  "generic_mspct" "list"
```

In addition to using `class` to query the class of the collection, we can use base R's `lapply` together with `class` or `class_spct` to query the class of each of the members of the collection.

```
is.filter_mspct(mixed.mspct)

## [1] FALSE

is.any_mspct(mixed.mspct)

## [1] TRUE

class(mixed.mspct)

## [1] "generic_mspct" "list"

lapply(mixed.mspct, class_spct)

## $filter
## [1] "filter_spct"  "generic_spct"
##
## $source
## [1] "source_spct"  "generic_spct"

lapply(mixed.mspct, class)

## $filter
## [1] "filter_spct"  "generic_spct" "data.table"   "data.frame"
##
## $source
## [1] "source_spct"  "generic_spct" "data.table"   "data.frame"
```

5 Wavebands

When a range of wavelengths or a range of wavelengths plus a spectral weighting function (SWF) is needed for radiation summaries or transformations, methods, operators and functions defined in package photobiology use `waveband` objects to store and exchange these data. A few other bits of information can 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. All waveband objects belong to the S3 class `waveband`.

5.1 Construction

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 returning the range of wavelengths (nm) as a numeric vector of length 2.

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

## range.400.700
## low (nm) 400
## high (nm) 700
## weighted none

my_PARb

## range.400.700
## low (nm) 400
## high (nm) 700
## weighted none
```

The examples above, show that both constructors return equivalent waveband objects.

Below we give a name to the waveband.

```
my_PARx <- waveband(c(400, 700), wb.name = "my_PARx")
```

Three examples of how to define equivalent wavebands based on a SWF follow.

```
my_CIE_1 <-
  waveband(c(250, 400), weight = "SWF", SWF.e.fun = CIE_e_fun, SWF.norm = 298)
my_CIE_2 <-
  waveband(c(250, 400), weight = "SWF", SWF.q.fun = CIE_q_fun, SWF.norm = 298)
my_CIE_3 <-
  waveband(c(250, 400), weight = "SWF", SWF.e.fun = CIE_e_fun,
                         SWF.q.fun = CIE_q_fun, SWF.norm = 298)
```

The function `waveband` is also useful when wanting to create a waveband covering the whole range of an spectrum, or 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
```

5.2 Querying the class

The function `is.waveband` can the used to query any R object. This function returns a logical value.

```
is.waveband(my_CIE_1)

## [1] TRUE

is.waveband(PAR())

## [1] TRUE

is.waveband(sun.spct)

## [1] FALSE
```

Base R function `class` can the used to retrieve the class attribute of any R object. This function returns a character vector.

```
class(PAR())

## [1] "waveband" "list"
```

5.3 Retrieving properties

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] FALSE
```

6 Collections of wavebands

In the current implementation there is no special class used for storing collections of `waveband` objects. We simply use base R's `list` class.

6.1 Construction

6.1.1 List constructor

Just base R's functions used to create a list object.

```
wavebands <- list(waveband(c(300,400)), waveband(c(400,500)))
wavebands

## [[1]]
## range.300.400
## low (nm) 300
## high (nm) 400
## weighted none
##
## [[2]]
## range.400.500
## low (nm) 400
## high (nm) 500
## weighted none
```

6.1.2 Special constructor

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 'split' boundaries are 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 case is handled 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
```

Table 4: Binary operators and operands. Validity and class of result. All operations marked 'Y' are allowed, those marked 'N' are forbidden and return `NA` and issue a warning.

e1	+	-	*	/	^	e2	result
cps_spct	Y	Y	Y	Y	Y	cps_spct	cps_spct
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
object_spct	N	N	N	N	N	object_spct	–
response_spct	Y	Y	Y	Y	N	response_spct	response_spct
chroma_spct	Y	Y	Y	Y	Y	chroma_spct	chroma_spct
cps_spct	Y	Y	Y	Y	Y	numeric	cps_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
object_spct	N	N	N	N	N	numeric	–
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	N	N	N	object_spct	–
source_spct	N	N	Y	N	N	waveband (no BSWF)	source_spct
source_spct	N	N	Y	N	N	waveband (BSWF)	source_spct

7 Transformations: using operators

7.1 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 4). 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
##   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 convolving) 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] "source_spct"  "generic_spct"

is_effective(weighted.spct)

## [1] TRUE

weighted.spct

##      w.length    s.e.irrad
##   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()

## Warning in oper.e.generic_spct(e1, e2, '-'):  One or both operands are
effective spectral irradiances

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

Table 5: Unary operators and maths 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
cps_spct	Y	Y	Y	Y	Y	Y	cps_spct
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
object_spct	N	N	N	N	N	N	–
response_spct	Y	Y	Y	Y	Y	Y	response_spct
chroma_spct	Y	Y	Y	Y	Y	Y	chroma_spct

7.2 Unary operators and maths functions

The most common mat functions, as well as unary minus and plus, are also implemented for spectral objects (see Table 5).

```
+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
##  ---
## 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
```

Table 6: Options recognized by functions in the photobiology package and the values they can take.

Option	default	function
Base R		
digits	7	$d-3$ used by `summary`
Package `data.table`		
datatable.print.nrows	$n = 100$	nrow(*spct*) $> n$ short" printing
datatable.print.topn	$n = 5$	nrows to print at top and bottom
datatable.verbose	`FALSE`	give verbose output or not
R4photobioloy suite		
photobiology.radiation.unit	`"energy"`	output ($\mathrm{W\,m^{-2}\,nm^{-1}}$)
	`"photon"`	output ($\mathrm{mol\,m^{-2}\,s^{-1}\,nm^{-1}}$)
photobiology.filter.qty	`"transmittance"`	output (/1)
	`"absorptance"`	output (/1)
	`"absorbance"`	output (a.u. $\log_1 0$ base)
photobiology.use.hinges	`NULL`	guess automatically
	`TRUE`	do not insert hinges
	`FALSE`	do insert hinges
photobiology.auto.hinges.limit	0.5	wavelength step (nm)
photobiology.waveband.trim	`TRUE`	trim or exclude
photobiology.use.cached.mult	`FALSE`	cache intermediate results or not
photobiology.verbose	`FALSE`	give verbose output or not

```
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
```

7.3 Options

Table 6 lists all the recognized options affecting maths operators and functions, and their default values. Within the suite all functions have a default value which is used when the options are undefined. Options are set using base R's function `options`, and queried with functions `options` and `getOption`.

The behaviour of the operators defined in this package 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)
```

Table 7: Transformation methods for spectra. Key: + available, – not available, f available in the future.

methods	source	response	filter	reflector	object	chroma
`merge`	+	+	+	+	+	+
`rbindspct`	+	+	+	+	+	+
`e2q, q2e`	+	+	–	–	–	–
`A2T, T2A`	–	–	+	–	–	–
`subset`	+	+	+	+	+	+
`trim_spct`	+	+	+	+	+	+
`interpolate_spct`	+	+	+	+	+	+
`f_scale`	+	+	+	+	+	+
`normalize`	+	+	+	+	+	+
math operators	+	+	+	+	+	+
math functions	+	+	+	+	+	+
`tag`	+	+	+	+	+	+

Table 8: Transformation methods for collections of spectra. Key: + available, – not available, f available in the future.

methods	source	response	filter	reflector	object	chroma
`mutate_mspct`	+	+	+	+	+	+
`rbindspct`	+	+	+	+	+	+
`e2q, q2e`	+	+	–	–	–	–
`A2T, T2A`	–	–	+	–	–	–
`trim_spct`	f	f	f	f	f	f
`interpolate_spct`	f	f	f	f	f	f
`f_scale`	f	f	f	f	f	f
`normalize`	f	f	f	f	f	f
math operators	f	f	f	f	f	f
math functions	f	f	f	f	f	f
`tag`	f	f	f	f	f	f

8 Transformations: methods and functions

In this section we describe methods and functions that take one or more spectral objects, and in some cases also waveband objects, as arguments and return another spectral object (Table 7) or that take a collection of spectral objects, and in some cases also waveband objects, as argumnents and return a collection of spectral objects (Table 8).

8.1 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.

In addition to lists of spectral objects, `rbindspct` accepts objects of `generic_mspct` and derived classes as input.

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

By default an ID factor named `spct.idx` is added so that it is possible to identify the origin of the observations after the binding. If the supplied list has named members, then these names are used as factor levels. Otherwise level names are generated automatically.

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

##       w.length    s.e.irrad spct.idx    s.q.irrad
##    1:      293 2.609665e-06   spct_1 6.391730e-12
##    2:      294 6.142401e-06   spct_1 1.509564e-11
##   ---
## 1015:      799 4.185850e-01   spct_2 2.795738e-06
## 1016:      800 4.069055e-01   spct_2 2.721132e-06
```

It is possible to suppress the creation of the ID factor, but this is rarely of any use.

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

##       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
```

If a named list of spectra 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      sun 6.391730e-12
##    2:      294 6.142401e-06      sun 1.509564e-11
##   ---
## 1015:      799 4.185850e-01    shade 2.795738e-06
## 1016:      800 4.069055e-01    shade 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   sun 6.391730e-12
##    2:      294 6.142401e-06   sun 1.509564e-11
##   ---
## 1015:      799 4.185850e-01 shade 2.795738e-06
## 1016:      800 4.069055e-01 shade 2.721132e-06
```

In the special case when the members of the list are `source_spct` objects containing effective spectral irradiance data, and they are not based on the same BSWF, an additional factor `BSWF` will be automatically added, and the `BSWF` attribute of the resulting spectrum set to `"multiple"`.

```
rb2.spct <- rbindspct(list(sun.spct * CIE(), shade.spct * CIE()))
rb2.spct

##      w.length    s.e.irrad spct.idx
##   1:  293.000 2.609665e-06   spct_1
##   2:  294.000 6.142401e-06   spct_1
##  ---
## 215:  399.000 7.378801e-05   spct_2
## 216:  399.999 7.395674e-05   spct_2

getBSWFUsed(rb2.spct)

## [1] "CIE98.298"
```

```
rb1.spct <- rbindspct(list(sun.spct * CIE(), sun.spct * GEN.G()))

## Warning in check.generic_spct(x, multiple.wl = multiple.wl):  'w.length'
values are not all unique with 25 duplicate values.

rb1.spct

##      w.length    s.e.irrad spct.idx      BSWF
##   1:  293.000 2.609665e-06   spct_1 CIE98.298
##   2:  294.000 6.142401e-06   spct_1 CIE98.298
##  ---
## 129:  313.000 1.491456e-03   spct_2 GEN.G.300
## 130:  313.299 5.104908e-06   spct_2 GEN.G.300

getBSWFUsed(rb1.spct)

## [1] "multiple"
```

The warning above is caused by the different length of the members of the list, and in this case it is safe to ignore it.

```
range(sun.spct * CIE())

## [1] 293.000 399.999

range(sun.spct * GEN.G())

## [1] 293.000 313.299
```

Special `subset` methods for spectral objects have been implemented. These are used by default, and in contrast to `subset.data.table` preserve the attributes used by this package. The only difference, in the current implementation, compared to base R `subset()` is that the `select` arguments, is quietly ignored—this argument in base R can be used to select columns.

In contrast to `trim_spct`, `subset` never interpolates or inserts *hinges*. On the other hand, the `subset` argument accepts any logical expression and can be consequently used to do subsetting, for example, based on factors.

```
subset(sun.spct, s.e.irrad > 0.2)

##      w.length s.e.irrad    s.q.irrad
##   1:      324 0.2075508 5.621282e-07
##   2:      325 0.2168055 5.890059e-07
##  ---
## 474:      799 0.4185850 2.795738e-06
## 475:      800 0.4069055 2.721132e-06

subset(sun.spct, w.length > 600)

##      w.length s.e.irrad    s.q.irrad
##   1:      601 0.6295837 3.162962e-06
##   2:      602 0.6305890 3.173284e-06
##  ---
## 199:      799 0.4185850 2.795738e-06
## 200:      800 0.4069055 2.721132e-06

subset(sun.spct, c(TRUE, rep(FALSE, 99)))

##    w.length    s.e.irrad    s.q.irrad
## 1:      293 2.609665e-06 6.391730e-12
## 2:      393 2.422023e-01 7.956770e-07
## 3:      493 7.382081e-01 3.042228e-06
## 4:      593 6.106598e-01 3.027053e-06
## 5:      693 4.870913e-01 2.821692e-06
## 6:      793 4.201554e-01 2.785153e-06

subset(sun.spct, c(1:5, 10))

##    w.length    s.e.irrad    s.q.irrad
## 1:      293 2.609665e-06 6.391730e-12
## 2:      294 6.142401e-06 1.509564e-11
## 3:      295 2.176175e-05 5.366385e-11
## 4:      296 6.780119e-05 1.677626e-10
## 5:      297 1.533491e-04 3.807181e-10
## 6:      302 3.922583e-03 9.902505e-09

selector <- 1:5
subset(sun.spct, selector)

##    w.length    s.e.irrad    s.q.irrad
## 1:      293 2.609665e-06 6.391730e-12
## 2:      294 6.142401e-06 1.509564e-11
## 3:      295 2.176175e-05 5.366385e-11
## 4:      296 6.780119e-05 1.677626e-10
## 5:      297 1.533491e-04 3.807181e-10
```

8.2 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.

8.3 Normalizing a spectrum

Function normalize permits normalizing a spectrum to one at an arbitrary wavelength (nm) or to the wavelength of either the maximum or the minimum spectral value.

```
normalize(sun.spct)

##      w.length    s.e.irrad
##   1:      293 3.180721e-06
##   2:      294 7.486502e-06
##  ---
## 507:      799 5.101812e-01
## 508:      800 4.959460e-01
```

Which is equivalent to

```
normalize(sun.spct, norm = "max")

##      w.length    s.e.irrad
##   1:      293 3.180721e-06
##   2:      294 7.486502e-06
##  ---
## 507:      799 5.101812e-01
## 508:      800 4.959460e-01
```

We can also supply an arbitrary wavelength within the range of the data, and interpolation will be used if needed to calculate the multiplier but no insertion will be done on the spectral data. Consequently, a spectral value equal to one will not necessarily be generated when supplying a wavelength value as argument.

```
normalize(sun.spct, norm = 600.3)

## Warning in setTimeUnit(new.spct, time.unit.spct):  Overrriding existing
'time.unit' 'second' with 'unknown' may invalidate data!

##      w.length    s.e.irrad
##   1:      293 4.100701e-06
##   2:      294 9.651870e-06
##  ---
## 507:      799 6.577441e-01
## 508:      800 6.393916e-01
```

It is also possible to supply a range within which the normalization wavelength will be searched.

```
normalize(sun.spct, range = PAR(), norm = "max")
```

```
##      w.length    s.e.irrad
##   1:      293 3.180721e-06
##   2:      294 7.486502e-06
##  ---
## 507:      799 5.101812e-01
## 508:      800 4.959460e-01
```

As is the case for other functions, the argument "unit.out" can be used to change the type output from the default.

8.4 Rescaling a spectrum

Function f_scale rescales a spectrum by dividing each spectral data value by a summary calculated with a function (f) selected by a character string, either the integrated ”total” or the average or ”mean” value over a range of wavelengths.

```
fscale(sun.spct)

##      w.length    s.e.irrad
##   1:      293 4.916305e-06
##   2:      294 1.157157e-05
##  ---
## 507:      799 7.885654e-01
## 508:      800 7.665627e-01
```

```
fscale(sun.spct, f = "total")

##      w.length    s.e.irrad
##   1:      293 9.696855e-09
##   2:      294 2.282361e-08
##  ---
## 507:      799 1.555356e-03
## 508:      800 1.511958e-03
```

It is also possible to supply a range within which integration or averaging will be done.

```
fscale(sun.spct, range = PAR(), f = "mean")

##      w.length    s.e.irrad
##   1:      293 3.981501e-06
##   2:      294 9.371307e-06
##  ---
## 507:      799 6.386246e-01
## 508:      800 6.208056e-01
```

As is the case for other functions, the argument "unit.out" can be used to change the type output from the default.

In addition to the character constants "total" and "mean", it is possible to pass any suitable R function, built-in or user defined, through parameter f plus additional named arguments to it.

8.5 Wavelength interpolation

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
```

8.6 Trimming

> 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 spectral 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 `low.limit` and/or `high.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
```

8.7 Convoluting weights

It is very instructive to look at weighted spectral data to understand how effective irradiances are calculated. Plotting effective spectral irradiance data can be very instructive when analyzing the interaction of photoreceptors and ambient radiation. It can also illustrate what a large effect that small measuring errors can have on the estimated effective irradiances or exposures when SWFs have a steep slope.

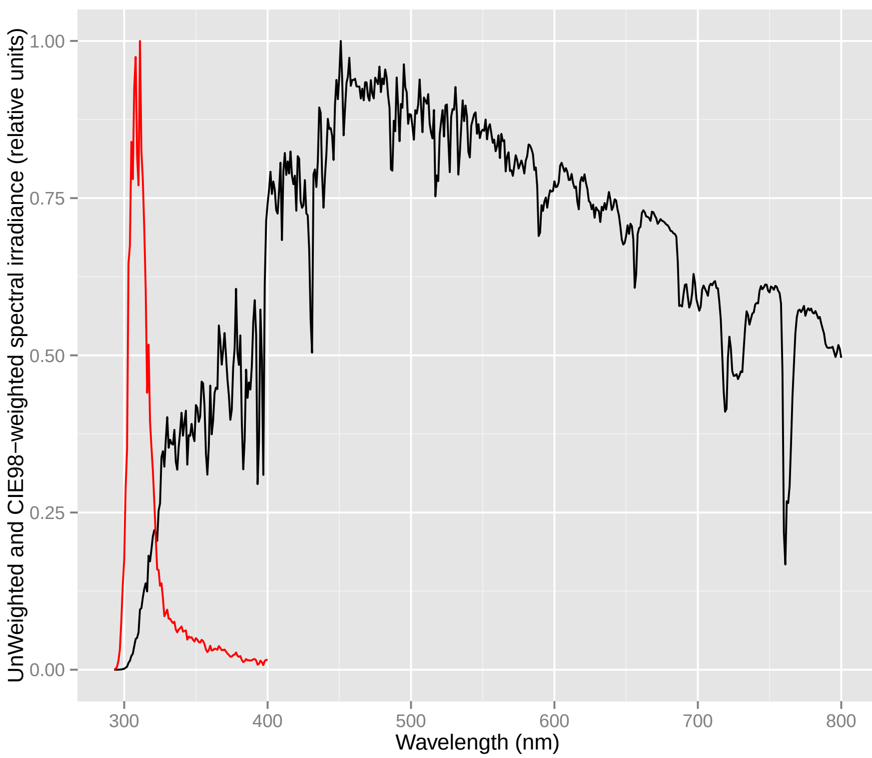
8.7.1 Individual spectra

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.irrad
##   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.

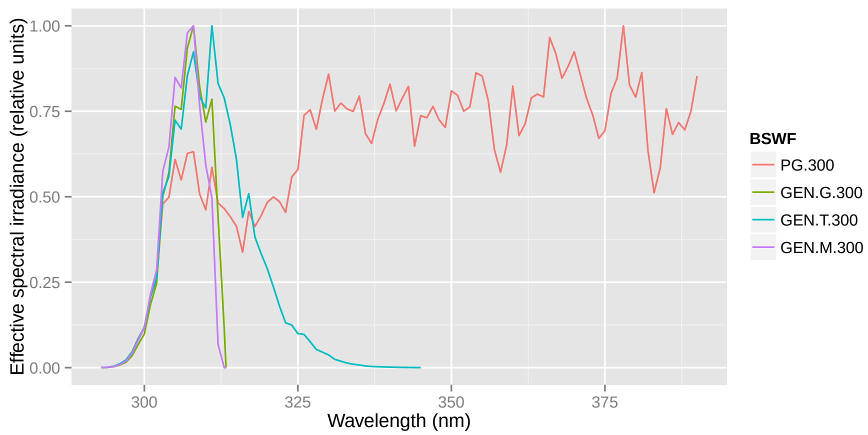
```
ggplot(data = sun.spct, aes(x = w.length, y = s.e.irrad/max(s.e.irrad))) +
  geom_line() +
  geom_line(data = sun.spct * CIE(), colour = "red") +
  labs(x = "Wavelength (nm)",
       y = "UnWeighted and CIE98-weighted spectral irradiance (relative units)")
```



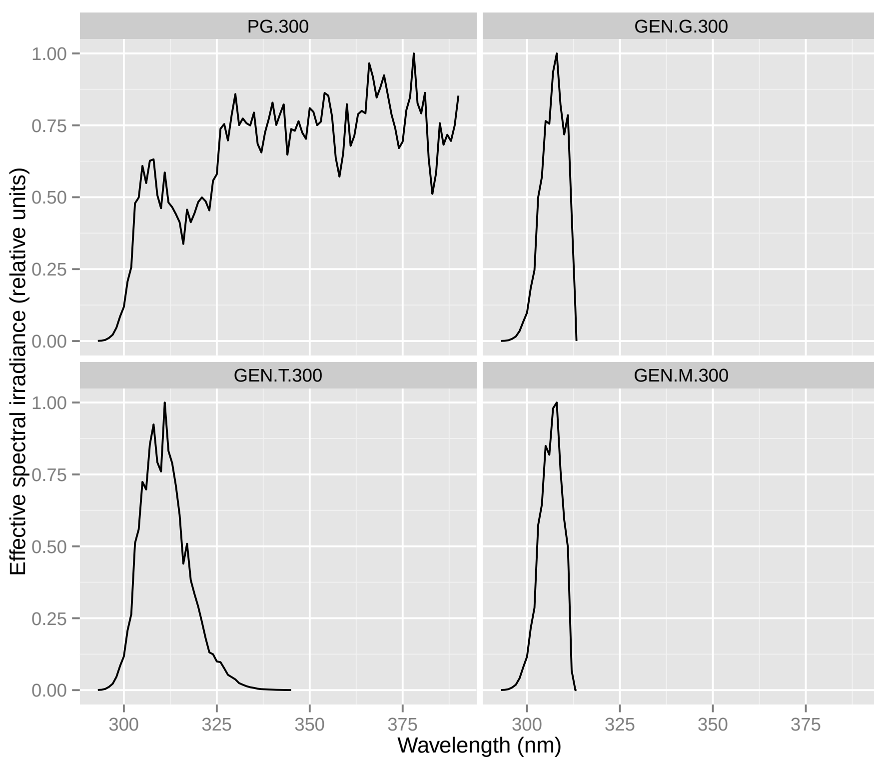
An alternative is as follows, where it is easier to use other aesthetics, or plot additional curves as shown in the chunk below.

```
combined.spct <- rbindspct(list(normalize(sun.spct * PG(), range = c(290,400)),
                                normalize(sun.spct * GEN.G(), range = c(290,400)),
                                normalize(sun.spct * GEN.T(), range = c(290,400)),
                                normalize(sun.spct * GEN.M(), range = c(290,400))
                                )
                           )
```

```
ggplot(data = combined.spct, aes(x = w.length, y = s.e.irrad, colour = BSWF)) +
  geom_line() +
  labs(x = "Wavelength (nm)",
       y = "Effective spectral irradiance (relative units)")
```



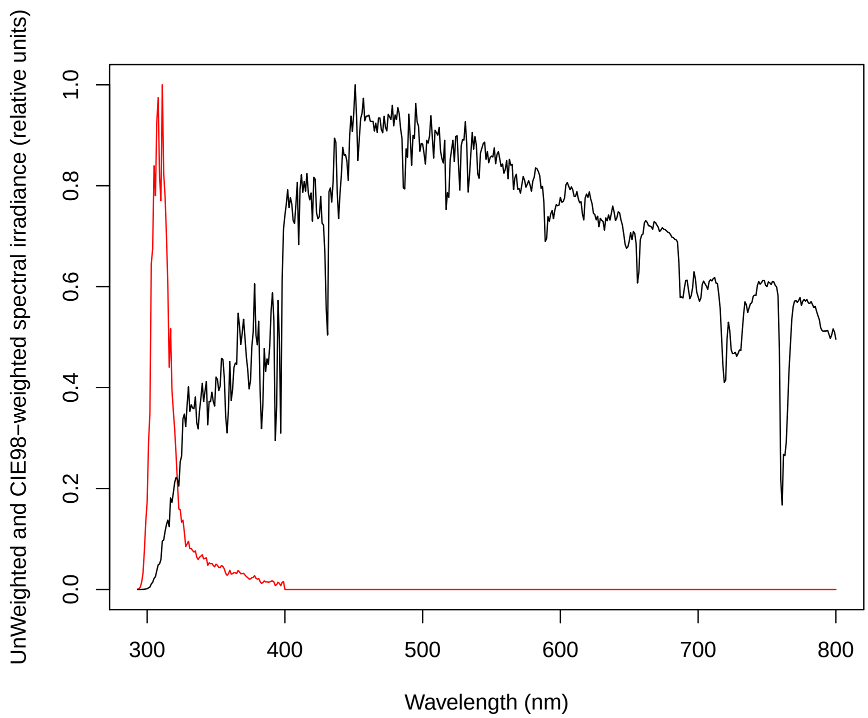

```
ggplot(data = combined.spct, aes(x = w.length, y = s.e.irrad)) +
  geom_line() + facet_wrap(~BSWF, ncol = 2) +
  labs(x = "Wavelength (nm)",
       y = "Effective spectral irradiance (relative units)")
```



8.7.2 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)")
lines(s.e.irrad/max(s.e.irrad) ~ w.length, col = "black", data = sun.spct)
```



8.8 Tagging with bands and colours

We call tagging, to the process of adding reference information to spectral data. For example we can add a factor indicating regions or bands in the spectrum. We can add also information on the colour, as seen by humans, for each observed value, or for individual regions or bands of the spectrum. In most cases this additional information is used for annotations when plotting the spectral data.

8.8.1 Individual spectra

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' a 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
```

9 Summaries

Summaries can be calculated both from individual spectral objects (Table 9) and from collections of spectral objects (Table 10). They return a *simpler* object than the spectral data in their arguments. For example a vector of numeric values, possibly of length one, in the case of individual spectra, or a data frame containing one row of summary data for each spectrum the collection of multiple spectra supplied as argument.

Table 9: Summary methods for spectra. Key: + available, – not available, f available in the future.

methods	source	response	filter	reflector	object	chroma
irrad, e_irrad, q_irrad	+	–	–	–	–	–
fluence, e_fluence, q_fluence	+	–	–	–	–	–
ratio, e_ratio, q_ratio	+	–	–	–	–	–
qe_ratio, eq_ratio	+	–	–	–	–	–
response, e_response, q_response	–	+	–	–	–	–
transmittance	–	–	+	–	+	–
absorptance	–	–	+	–	+	–
absorbance	–	–	+	–	+	–
range, min, max	+	+	+	+	+	+
stepsize, spread, midpoint	+	+	+	+	+	+
labels	+	+	+	+	+	+
summary	+	+	+	+	+	+
peaks	+	+	+	+	+	+
valleys	+	+	+	+	+	+
integrate_spct	+	+	+	+	+	+
average_spct	+	+	+	+	+	+
color	+	–	+	+	–	+

Table 10: Summary methods for collections of spectra. Key: + available, – not available, f available in the future.

methods	source	response	filter	reflector	object	chroma
f_mspct	+	+	+	+	+	+
irrad, e_irrad, q_irrad	+	–	–	–	–	–
fluence, e_fluence, q_fluence	+	–	–	–	–	–
ratio, e_ratio, q_ratio	+	–	–	–	–	–
qe_ratio, eq_ratio	+	–	–	–	–	–
response, e_response, q_response	–	+	–	–	–	–
transmittance	–	–	+	–	+	–
absorptance	–	–	+	–	+	–
absorbance	–	–	+	–	+	–
range, min, max	+	+	+	+	+	+
stepsize, spread, midpoint	+	+	+	+	+	+
labels	–	–	–	–	–	–
summary	–	–	–	–	–	–
peaks	f	f	f	f	f	f
valleys	f	f	f	f	f	f
integrate_spct	f	f	f	f	f	f
average_spct	f	f	f	f	f	f
color	f	f	f	f	f	f

9.1 Summary

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)

##     w.length       s.e.irrad           s.q.irrad
##  Min.   :293.0   Min.   :0.0000026   Min.   :6.000e-12
##  1st Qu.:419.8   1st Qu.:0.4201147   1st Qu.:2.124e-06
##  Median :546.5   Median :0.5847238   Median :2.951e-06
##  Mean   :546.5   Mean   :0.5301739   Mean   :2.474e-06
##  3rd Qu.:673.2   3rd Qu.:0.6695883   3rd Qu.:3.156e-06
##  Max.   :800.0   Max.   :0.8204633   Max.   :3.375e-06
```

```
summary(sun.spct * CIE())

##     w.length       s.e.irrad
##  Min.   :293.0   Min.   :2.610e-06
##  1st Qu.:319.8   1st Qu.:1.034e-04
##  Median :346.5   Median :2.195e-04
##  Mean   :346.5   Mean   :7.579e-04
##  3rd Qu.:373.2   3rd Qu.:6.291e-04
##  Max.   :400.0   Max.   :4.710e-03
```

```
summary(sun.daily.spct)

##     w.length       s.e.irrad       s.q.irrad
##  Min.   :290.0   Min.   :    0   Min.   :0.00000
##  1st Qu.:417.5   1st Qu.:17916   1st Qu.:0.08357
##  Median :545.0   Median :24048   Median :0.12152
##  Mean   :545.0   Mean   :21399   Mean   :0.10068
##  3rd Qu.:672.5   3rd Qu.:26879   3rd Qu.:0.12928
##  Max.   :800.0   Max.   :32608   Max.   :0.13813
```

9.2 Wavelength

9.2.1 Individual spectra

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. They operate on every numeric column in the spectrum object given as argument.

```
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 summary functions are available for spectra, but redefined to return wavelength based summaries in nanometres (nm).

```
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
```

9.2.2 Collections of spectra

> Not all summary methods are yet implemented for collections of spectra, because this is a new set of classes, just added to the package. See Table 10 where methods be implemented in the *future* are marked with 'f'. Functions `f_mspct` or `plyr::ldply` can be used to apply a function to all the spectra in a collection and obtain the results in a data frame object. Other *apply* functions or `for` loops can also be used if needed.

Collections of spectra can be useful not only for time-series of spectra or spectral images, but also when dealing with a small group of related spectra. In the example below we show how to use a collection of spectra for calculating summaries. The spectra in a collection do **not** need to have been measured at the same wavelength values, or have the same number of rows or even of columns. Consequently, in many cases applying the wavelength summary functions described above to collections of spectra can be useful. The value returned is a data frame, with a number of data columns equal to the length of the returned value by the corresponding method for individual spectra.

```
filtered_sun <-
  source_mspct(
    lapply(list(none = clear.spct,
                ug1 = ug1.spct, ug11 = ug11.spct,
                gg400 = gg400.spct,
                og550 = og550.spct,
                rg665 = rg665.spct, rg830 = rg830.spct),
           `*`,
           y = sun.spct))
range(filtered_sun)

##   spct.idx min.wl max.wl
## 1     none    293    800
## 2      ug1    293    800
## 3     ug11    293    800
## 4    gg400    293    800
## 5    og550    293    800
## 6    rg665    293    800
## 7    rg830    293    800
```

9.3 Peaks and valleys

9.3.1 Individual spectra

Functions `peaks` and `valleys` take spectra as first argument and return a subset of the spectral object data corresponding to local maxima and local minima of the measured variable. `span` defines the width of the 'window' used as a number of observations.

```
peaks(sun.spct, span = 51)

##    w.length s.e.irrad
## 1:      451 0.8204633
## 2:      495 0.7899872
## 3:      747 0.5025733

valleys(sun.spct, span = 51)

##    w.length s.e.irrad
## 1:      358 0.2544907
## 2:      393 0.2422023
## 3:      431 0.4136900
## 4:      487 0.6511654
## 5:      517 0.6176652
## 6:      589 0.5658760
## 7:      656 0.4982959
## 8:      719 0.3366514
## 9:      761 0.1373212
```

In the case of `source_spct` and `response_spct` methods `unit.out` can be used to force peaks to be searched using either energy or photon based spectral irradiance. The default is energy, or the option `"photobiology.radiation.unit"` if set.

```
peaks(sun.spct, span = 51, unit.out = "photon")

##    w.length    s.q.irrad
## 1:      451 3.093155e-06
## 2:      495 3.268822e-06
## 3:      531 3.374912e-06
## 4:      621 3.355564e-06
## 5:      668 3.337584e-06
## 6:      712 3.017948e-06
## 7:      754 3.156213e-06
```

In the case of `filter_spct` methods `unit.out` can be used to force peaks to be searched using either spectral transmittance or spectral absorbance. The default is transmittance, or the option `"photobiology.filter.qty"` if set.

```
peaks(ug1.spct)

##    w.length   Tfr
## 1:      360 0.830
## 2:      750 0.470
## 3:     1400 0.033
## 4:     1600 0.034
## 5:     2700 0.180

peaks(ug1.spct, filter.qty = "absorbance")

##    w.length        A
## 1:     1250 1.585027
## 2:     1500 1.522879
## 3:     1800 1.698970

valleys(ug1.spct, filter.qty = "absorbance")

##    w.length          A
## 1:      360 0.08092191
## 2:      750 0.32790214
## 3:     1400 1.48148606
## 4:     1600 1.46852108
## 5:     2700 0.74472749
```

It is possible to approximately set the width of the windows in nanometres by using function `step_size`. As `stepsize` returns a range, we use `mean` in this example, although one could also use an index (1 or 2), or `min` or `max` depending on needs.

```
peaks(sun.spct, span = 21 / mean(stepsize(sun.spct)))

##     w.length s.e.irrad
##  1:      354 0.3758625
```

```
##  2:      366 0.4491898
## ---
## 17:      747 0.5025733
## 18:      774 0.4746771
```

Low level functions `find_peaks`, `get_peaks` and `get_valleys` take numeric vectors as arguments.

9.4 Irradiance

9.4.1 Individual spectra

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

```
irrad(sun.spct)

##    Total
## 269.1249
## 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
## 196.6343
## 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
## 0.0008941352
## 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
## 894.1352
## 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 option `photobiology.radiation.unit` is set.

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

##      PAR
## 196.6343
## 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
## 0.0008941352
## 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
## 196.6343
## attr(,"time.unit")
## [1] "second"
## attr(,"radiation.unit")
## [1] "energy irradiance total"

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

##      PAR
## 894.1352
## attr(,"time.unit")
## [1] "second"
## attr(,"radiation.unit")
## [1] "photon irradiance total"

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

##      PAR
## 36.31755
## attr(,"time.unit")
## [1] "day"
## attr(,"radiation.unit")
## [1] "photon irradiance total"
```

It is also possible to supply a time unit to use as basis of expression for the returned value, but be aware that conversion into a loger time unit is only valid for sources like lamps, which have an output the remains constant in time.

```
irrad(sun.spct, PAR(), time.unit = "hour")

##      PAR
## 707883.4
## attr(,"time.unit")
## [1] "hour"
## attr(,"radiation.unit")
## [1] "energy irradiance total"

irrad(sun.spct, PAR(), time.unit = duration(8, "hours"))

##     PAR
## 5663067
## attr(,"time.unit")
## [1] "28800s (~8 hours)"
## attr(,"radiation.unit")
## [1] "energy irradiance total"
```

Using a shorter time unit than the original, yields an average value re-expressed on a new time unit base.

```
irrad(sun.daily.spct, PAR(), time.unit = "second")

##      PAR
## 92.16251
## attr(,"time.unit")
## [1] "second"
## attr(,"radiation.unit")
## [1] "energy 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
## 196.6343
## 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
##       196.6343
## 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))) # W m-2 (BE)

##  CIE98.298.tr.lo  CIE98.298.tr.lo  CIE98.300.tr.lo
##       0.08181415       0.08181415       0.12613292
## attr(,"time.unit")
## [1] "second"
## attr(,"radiation.unit")
## [1] "energy irradiance total"

my_wavebands <- list(Red(), Blue(), Green())
e_irrad(sun.spct, my_wavebands) # W m-2

##    Red.ISO   Blue.ISO  Green.ISO
##   79.38161   37.55208   49.26861
## 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
##      0.6445035     27.9841813
## 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
##      0.6445035     27.9841813
## 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
##    0.002396624    0.104060802
## 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
##      0.2396624     10.4060802
## attr(,"time.unit")
## [1] "second"
## attr(,"radiation.unit")
## [1] "energy irradiance contribution.pc"
```

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

##  UVB.ISO.tr.lo        UVA.ISO
##     0.02251251     0.97748749
## 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
##       2.251251      97.748749
## 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
##     0.02929561     0.32922566
## attr(,"time.unit")
## [1] "second"
## attr(,"radiation.unit")
## [1] "energy irradiance average"
```

9.4.2 Collections of spectra

Collections of spectra can be useful not only for time-series of spectra or spectral images, but also when dealing with a small group of related spectra. In the example below we show how to use a collection of spectra for estimating irradiances under different filters set up in sunlight.

One thing to remember, is that operators in R are just functions with special names and call syntax. They can also be called with the usual function call syntax by enclosing their *name* in backquotes. We use this trick to use the multiplication operator `*` in a call to `lapply` which returns a list, which we convert into a `source_multi_spct` object. After this we just call the `irrad` method on the *collection of spectra* and obtain the result as a data frame with one row per spectrum and one column by waveband.

```r
filtered_sun <-
  source_mspct(
    lapply(list(none = clear.spct,
                ug1 = ug1.spct, ug11 = ug11.spct,
                gg400 = gg400.spct,
                og550 = og550.spct,
                rg665 = rg665.spct, rg830 = rg830.spct),
           `*`,
           y = sun.spct))
irrad(filtered_sun, list(UVA(), VIS()))

##   spct.idx irrad_UVA.ISO irrad_VIS.ISO
```

```
## 1     none  2.798418e+01  2.318635e+02
## 2      ug1  1.821686e+01  1.275260e+01
## 3     ug11  1.815903e+01  8.748845e+00
## 4    gg400  1.100092e+00  2.169464e+02
## 5    og550  2.798418e-04  1.156291e+02
## 6    rg665  2.798418e-04  4.455555e+01
## 7    rg830  2.798418e-04  2.875945e-03
```

9.4.3 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
## 0.0008941352

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

##          PAR
## 0.0008941352
```

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
## 0.08181415 0.08181415 0.12613292

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

##   Red.ISO  Blue.ISO Green.ISO
##  79.38161  37.55208  49.26861
```

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
##      28.62665      69.69042      68.48951      58.45435      43.86192

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
##      35.44164      34.83091      29.72745

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
##      29.45287      35.13414      35.41299
```

9.5 Fluence

The code using `spct` objects is simple, to integrate the whole spectrum we can use code similar to that for irradiances, but we need to always supply `exposure.time` as it has no default. The exposure time is a `lubridate::duration`, but any argument accepted by `as.duration` can also be used.

```
fluence(sun.spct, exposure.time = duration(1, "hours"))

##    Total
## 968849.6
## attr(,"radiation.unit")
## [1] "energy fluence (J m-2)"
## attr(,"exposure.duration")
## [1] "3600s (~1 hours)"

fluence(sun.spct, exposure.time = 3600) # seconds

## converting 'time.unit' 3600 into a lubridate::duration
```

```
##    Total
## 968849.6
## attr(,"radiation.unit")
## [1] "energy fluence (J m-2)"
## attr(,"exposure.duration")
## [1] 3600
```

```
fluence(sun.spct, exposure.time = hms("01:00:00"))

## converting 'time.unit' 1H 0M 0S into a lubridate::duration
## estimate only:  convert periods to intervals for accuracy

##    Total
## 968849.6
## attr(,"radiation.unit")
## [1] "energy fluence (J m-2)"
## attr(,"exposure.duration")
## [1] "1H 0M 0S"
```

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

```
fluence(sun.spct, PAR(), unit.out = "photon",
        exposure.time = duration(1, "hours"))

##      PAR
## 3.218887
## attr(,"radiation.unit")
## [1] "photon fluence (mol m-2)"
## attr(,"exposure.duration")
## [1] "3600s (~1 hours)"
```

```
fluence(sun.spct, CIE(),
        exposure.time = duration(30, "minutes"))

##  CIE98.298.tr.lo
##         147.2655
## attr(,"radiation.unit")
## [1] "energy fluence (J m-2)"
## attr(,"exposure.duration")
## [1] "1800s (~30 minutes)"
```

```
fluence(sun.spct, CIE(),
        exposure.time = duration(0.5, "hours"))

##  CIE98.298.tr.lo
##         147.2655
## attr(,"radiation.unit")
## [1] "energy fluence (J m-2)"
## attr(,"exposure.duration")
## [1] "1800s (~30 minutes)"
```

9.6 Photon and energy ratios

9.6.1 Individual 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)
##              0.001873704
## 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)
##                         0.01936927                        0.98063073
## 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)
##                        0.019369272                       0.001873704
## attr(,"radiation.unit")
## [1] "q:q ratio"
```

Function `e_ratio` returns energy ratios.

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

##  UVB.ISO.tr.lo: PAR(q:q)
##              0.001873704
## attr(,"radiation.unit")
## [1] "q:q ratio"

e_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)
##                         0.01936927                        0.98063073
## attr(,"radiation.unit")
## [1] "q:q ratio"
```

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

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

##    q:e( PAR)
## 4.547199e-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)
##    3.968590e-06    4.469290e-06    5.682783e-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)
##  219915.6
## 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)
##        251978.7        223749.2        175970.1
## 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)
## 4.547199e-06
## attr(,"radiation.unit")
## [1] "q:e ratio"

PPFD.mol.photon * 1e6

## [1] 1000

PAR.energy

## q:e( PAR)
##  219.9156
## attr(,"radiation.unit")
## [1] "q:e ratio"
```

9.6.2 Collections of spectra

```
q_ratio(filtered_sun, list(UVB(), UVA()), PAR())

##   spct.idx q_ratio_UVB.ISO.tr.lo:PAR(q:q) q_ratio_UVA.ISO:PAR(q:q)
## 1     none                   1.873704e-03             9.486218e-02
## 2      ug1                   2.732910e-01             2.315138e+01
## 3     ug11                   1.665022e-01             5.844385e+00
## 4    gg400                   1.937804e-08             4.223096e-03
## 5    og550                   3.627087e-08             1.836328e-06
## 6    rg665                   1.655514e-07             8.381562e-06
## 7    rg830                   1.873704e-03             9.486218e-02
```

9.6.3 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`.

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
```

9.7 Normalized difference indexes

9.8 Individual spectra

These indexes are frequently used to summarize reflectance data, for example in remote rensing the NDVI (normalized difference vegetation index). Here we give an *unusual* example to demonstrate that function `normalized_diff_ind()` can be used to calculate, or define any similar index.

```r
normalized_diff_ind(sun.spct,
                    waveband(c(400, 700)), waveband(c(700, 1100)),
                    irrad)

## NDI  irrad  [ 400.700 ] - [ 700.1100 ]
##                              0.6352382
```

9.9 Transmittance, reflectance, absorptance and absorbance

9.9.1 Individual spectra

The functions `transmittance`, and `absorbance` take `filter_spct` as argument, while function `reflectance` takes `reflector_spct` objects as argument. Functions `transmittance`, `reflectance` and `absorptance` are implemented for `object_spct`. They return as default an average value for these quantities **assuming** a light source with a flat spectral energy output.

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

##     UVB.ISO     UVA.ISO         PAR
## 0.007671429 0.782682353 0.920245000
## attr(,"Tfr.type")
## [1] "total"
## attr(,"radiation.unit")
## [1] "transmittance average"
```

```r
transmittance(Solidago_lower_adax.spct, Plant_bands())

##   UVA.ISO.tr.lo    Blue.Sellaro   Green.Sellaro     Red.Smith20
##     0.009810439     0.044690005     0.173413734     0.087330122
##  FarRed.Smith20
##     0.437553025
## attr(,"Tfr.type")
## [1] "total"
## attr(,"radiation.unit")
## [1] "transmittance average"
```

```r
reflectance(Solidago_lower_adax.spct, Plant_bands())

##   UVA.ISO.tr.lo    Blue.Sellaro   Green.Sellaro     Red.Smith20
##      0.05013476      0.05708517      0.13744800      0.06682849
##  FarRed.Smith20
##      0.39845150
## attr(,"Rfr.type")
## [1] "total"
## attr(,"radiation.unit")
## [1] "reflectance average"
```

```
absorptance(Solidago_lower_adax.spct, Plant_bands())

##   UVA.ISO.tr.lo    Blue.Sellaro   Green.Sellaro     Red.Smith20
##       0.9400548       0.8982248       0.6891383       0.8458414
##  FarRed.Smith20
##       0.1639955
## attr(,"Afr.type")
## [1] "total"
## attr(,"radiation.unit")
## [1] "absorptance average"
```

```
reflectance(gold.spct, VIS_bands())

##  Purple.ISO    Blue.ISO   Green.ISO  Yellow.ISO  Orange.ISO     Red.ISO
##   0.3328451   0.3663725   0.7024582   0.8449222   0.8742509   0.9362462
## attr(,"Rfr.type")
## [1] "total"
## attr(,"radiation.unit")
## [1] "reflectance average"
```

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

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

##  UVC.ISO.tr.lo        UVB.ISO        UVA.ISO
##    0.006483333    0.007671429    0.782682353
## attr(,"Tfr.type")
## [1] "total"
## attr(,"radiation.unit")
## [1] "transmittance average"

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

##  UVC.ISO.tr.lo        UVB.ISO        UVA.ISO
##         0.5835         0.2685        66.5280
## attr(,"Tfr.type")
## [1] "total"
## attr(,"radiation.unit")
## [1] "transmittance total"

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

##  UVC.ISO.tr.lo        UVB.ISO        UVA.ISO
##   0.0013427591   0.0006178763   0.1530952496
## attr(,"Tfr.type")
## [1] "total"
## attr(,"radiation.unit")
## [1] "transmittance contribution"

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

##  UVC.ISO.tr.lo        UVB.ISO        UVA.ISO
##     0.13427591     0.06178763    15.30952496
## attr(,"Tfr.type")
## [1] "total"
## attr(,"radiation.unit")
## [1] "transmittance contribution.pc"
```

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

##  UVC.ISO.tr.lo        UVB.ISO        UVA.ISO
##    0.008659840    0.003984862    0.987355298
## attr(,"Tfr.type")
## [1] "total"
## attr(,"radiation.unit")
## [1] "transmittance relative"

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

##  UVC.ISO.tr.lo        UVB.ISO        UVA.ISO
##      0.8659840      0.3984862     98.7355298
## attr(,"Tfr.type")
## [1] "total"
## attr(,"radiation.unit")
## [1] "transmittance relative.pc"

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

##  UVC.ISO.tr.lo        UVB.ISO        UVA.ISO
##    0.006483333    0.007671429    0.782682353
## 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`.

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

##    UVB.ISO    UVA.ISO        PAR
## 2.30895546 0.12895143 0.03610019
## attr(,"Tfr.type")
## [1] "total"
## attr(,"radiation.unit")
## [1] "absorbance average"
```

Function `absorbance` also has an additional argument `quantity`, with default `"average"`, which can take values controlling the output.

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

##  UVC.ISO.tr.lo        UVB.ISO        UVA.ISO
##    0.006483333    0.007671429    0.782682353
## attr(,"Tfr.type")
## [1] "total"
## attr(,"radiation.unit")
## [1] "transmittance average"

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

##  UVC.ISO.tr.lo        UVB.ISO        UVA.ISO
##         0.5835         0.2685        66.5280
```

```
## attr(,"Tfr.type")
## [1] "total"
## attr(,"radiation.unit")
## [1] "transmittance total"

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

##  UVC.ISO.tr.lo        UVB.ISO        UVA.ISO
##   0.0013427591   0.0006178763   0.1530952496
## attr(,"Tfr.type")
## [1] "total"
## attr(,"radiation.unit")
## [1] "transmittance contribution"

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

##  UVC.ISO.tr.lo        UVB.ISO        UVA.ISO
##     0.13427591     0.06178763    15.30952496
## attr(,"Tfr.type")
## [1] "total"
## attr(,"radiation.unit")
## [1] "transmittance contribution.pc"

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

##  UVC.ISO.tr.lo        UVB.ISO        UVA.ISO
##    0.008659840    0.003984862    0.987355298
## attr(,"Tfr.type")
## [1] "total"
## attr(,"radiation.unit")
## [1] "transmittance relative"

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

##  UVC.ISO.tr.lo        UVB.ISO        UVA.ISO
##      0.8659840      0.3984862     98.7355298
## attr(,"Tfr.type")
## [1] "total"
## attr(,"radiation.unit")
## [1] "transmittance relative.pc"

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

##  UVC.ISO.tr.lo        UVB.ISO        UVA.ISO
##    0.006483333    0.007671429    0.782682353
## 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
##       2.506497
## 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
##       2.506497      82.127831      92.059897
## attr(,"time.unit")
## [1] "second"
## attr(,"radiation.unit")
## [1] "energy irradiance total"
```

9.9.2 Collections of spectra

Here we construct a collection of filter spectra, and then we calculate the transmittance of these filters for two wavebands, obtaining the results as a data frame, with one row per filter, and one column per waveband.

```
filters <-
  filter_mspct(
    list(clear = clear.spct,
         ug1 = ug1.spct, ug11 = ug11.spct,
         gg400 = gg400.spct,
         og550 = og550.spct,
         rg665 = rg665.spct, rg830 = rg830.spct))
transmittance(filters, list(UVA(), VIS()))

##   spct.idx transmittance_UVA.ISO transmittance_VIS.ISO
## 1    clear            1.00000000          1.000000e+00
## 2      ug1            0.65514706          7.671034e-02
## 3     ug11            0.69538235          5.161079e-02
## 4    gg400            0.02923176          9.199580e-01
## 5    og550            0.00001000          5.381038e-01
## 6    rg665            0.00001000          2.404373e-01
## 7    rg830            0.00001000          1.315789e-05
```

9.10 Integrated response

9.10.1 Individual spectra

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

e_response(Vital_BW_20.spct)

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

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

##    Total
## 8.174317
## attr(,"time.unit")
## [1] "second"
## 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(,"time.unit")
## [1] "second"
## attr(,"radiation.unit")
## [1] "energy response total"

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

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

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

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

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

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

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

9.10.2 Collections of spectra

```
sensors <-
  response_mspct(
    list(BW20 = Vital_BW_20.spct,
         Berger = Berger_UV_erythemal.spct))
response(sensors, list(UVC(), UVB(), UVA()), quantity = "contribution")

##   spct.idx response_UVC.ISO.tr.lo response_UVB.ISO response_UVA.ISO.tr.hi
## 1     BW20            0.050622869        0.9417168            0.007660344
## 2   Berger            0.003949164        0.9790677            0.016983168
```

9.11 Integration over wavelengths

When we need to integrate some *non-standard* `numeric` variable stored in a spectral object we can use functions `integrate_spct` or `average_spct`.

9.11.1 Calculation from individual spectra

We can integrate the values of arbitrary `numeric` 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 derived 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(), use.hinges = TRUE)

##  UVA.ISO
## 27.98418
## 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`, returning a total value no longer expressed per nanometre.

```
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.

The function `average_spct` integrates every column holding numeric values from a spectrum object, except for `w.length`, and divides the result by the *spread* or width of the wavelength range integrated, returning a value expressed in the same units as the spectral data.

```
average_spct(my.sun.spct)

##      e.irrad      q.irrad          one
## 5.308183e-01 2.476007e-06 1.000000e+00
```

10 Handling '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 because 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.

10.0.1 Trimming of noisy regions

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 wavelengths 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 (≈ sub-setted) spectral data object in further calculations or plotting.

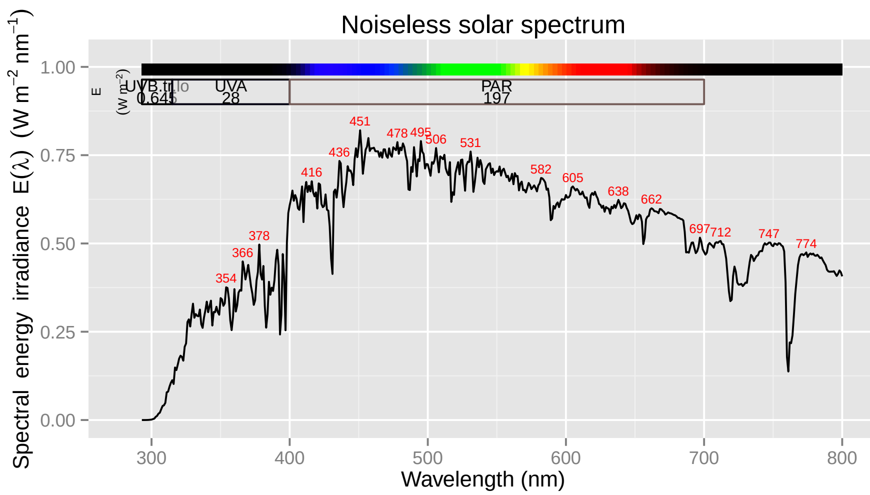
If the data are in a data frame, instead of a spct object then **subset** or indexing can be used. If the data are available as vectors, different options: 1) create a data frame from your data, 2) use the function **trim_tails()** from this package, or 3) just use R commands. Here we give examples of 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.

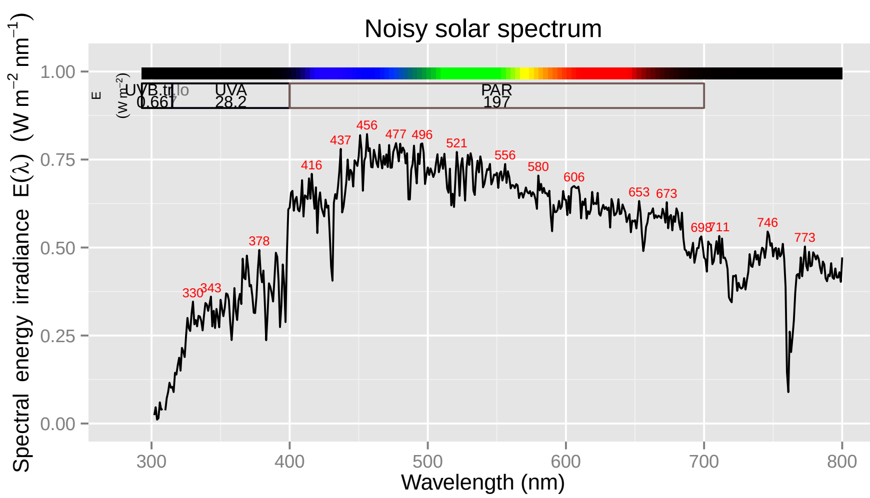
10.0.2 Smoothing

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.

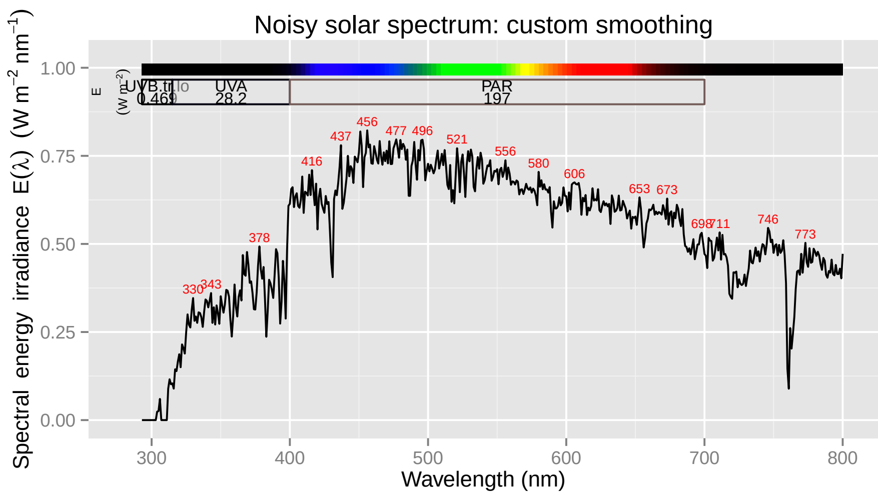
```
noisy.sun.spct <- sun.spct +
  rnorm(length(sun.spct$w.length), sd = 0.04) *
  irrad(sun.spct, quantity = "average")
plot(sun.spct) + labs(title = "Noiseless solar spectrum")
plot(noisy.sun.spct) + labs(title = "Noisy solar spectrum")

## Warning in range_check(x, strict.range = strict.range):  Negative spectral
energy irradiance values; minimun s.e.irrad = -0.036
```





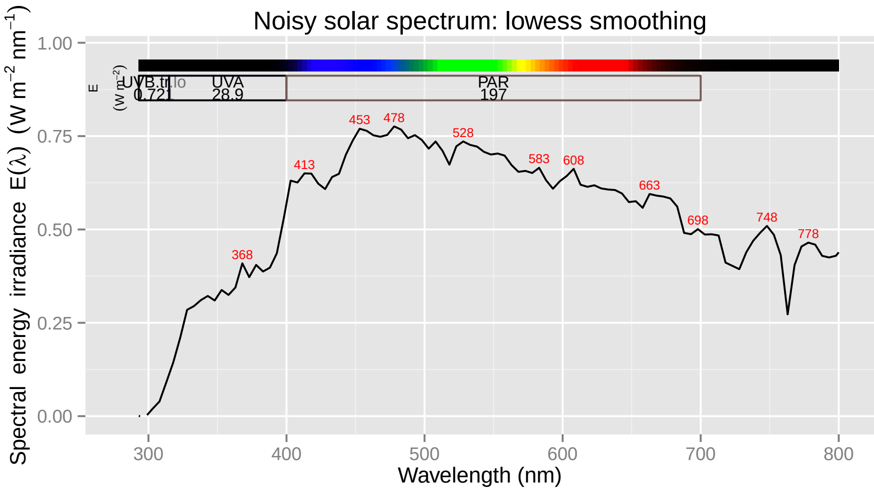
The default "custom'' method is our own, and is suitable for small amounts of noise, as it only applies smoothing to low signal regions of the spectrum, and also forces to zero those regions which are 'detected' to contain mostly noise. The strength parameter should be used to adjust the sensitivity 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")
```

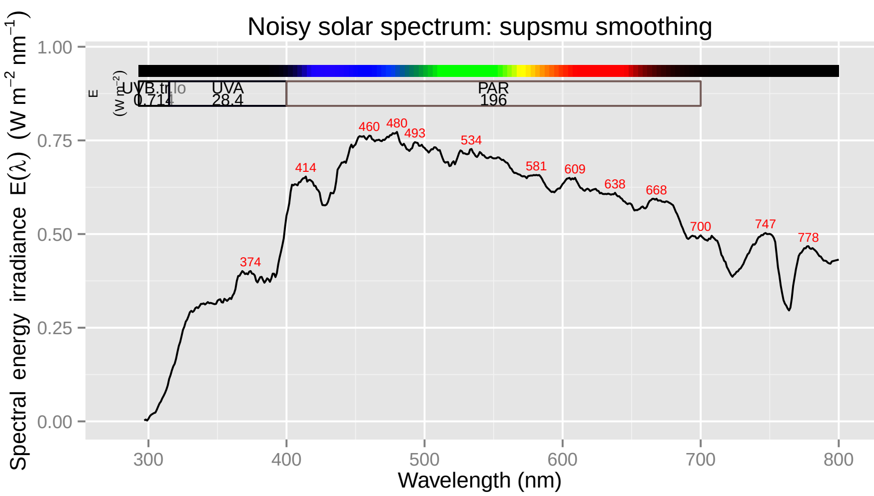


Methods "lowess" and "supsmu" are general purpose methods, which with their default values for the parameters tend to smooth spectra very aggressively. They remove a significant portion of the spectral detail but could be useful when the data is very noisy or when the overall shape of a spectrum is of 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 = -0.0014
## Warning in range_check(x, strict.range = strict.range):  Negative spectral
energy irradiance values; minimun s.e.irrad = -0.0014
```

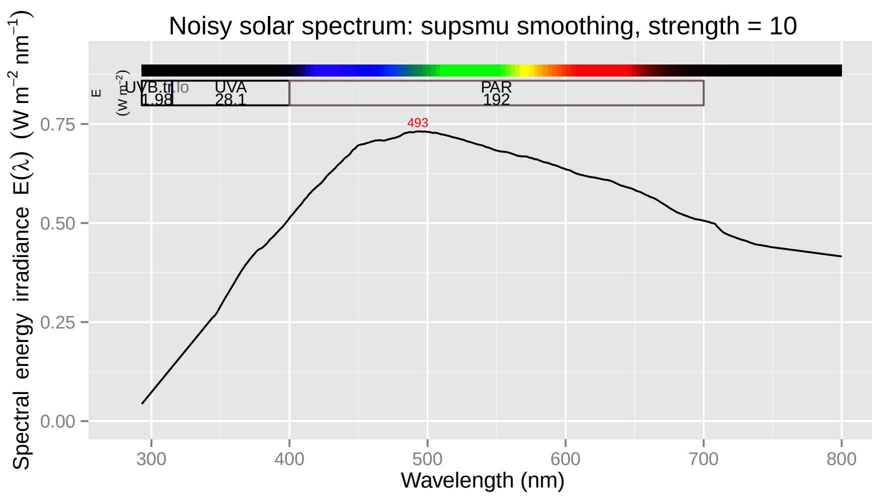



```
plot(smooth_spct(noisy.sun.spct, method = "supsmu")) +
  labs(title = "Noisy solar spectrum: supsmu smoothing")

## Warning in range_check(x, strict.range = strict.range):  Negative spectral
energy irradiance values; minimun s.e.irrad = -0.008
## Warning in range_check(x, strict.range = strict.range):  Negative spectral
energy irradiance values; minimun s.e.irrad = -0.008
## Warning:  Removed 4 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")
```



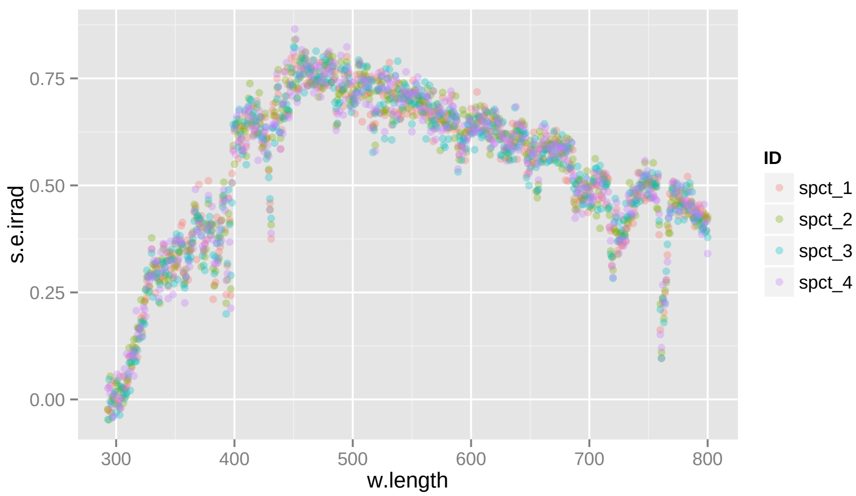
Function `smooth_spct` is generic with specializations for `source_spct`, `filter_spct`, `reflector_spct`, and `response_spct`.

10.0.3 Parallel averaging and summaries

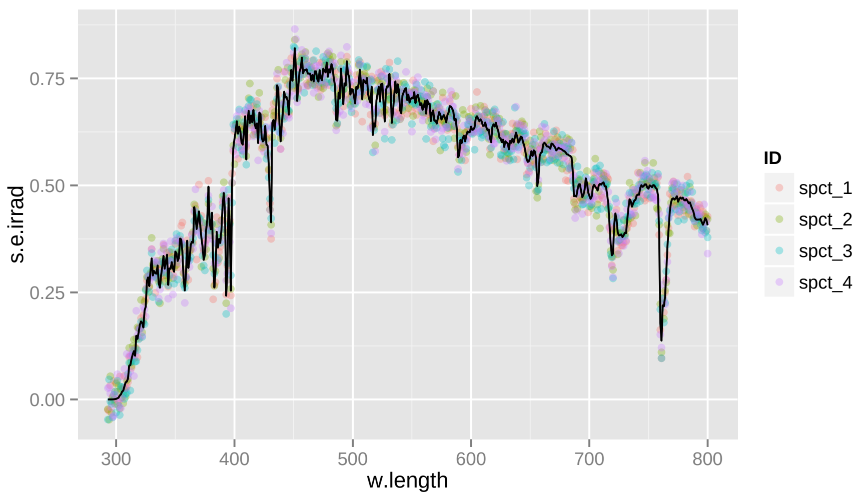
It is quite easy to calculate 'parallel' summary quantities using `data.table` syntax and function `rbindspct`. However, one should be careful with the handling of `NA` values, and specially make sure that all spectra have values for spectral irradiance at the same wavelengths. Similar code to that given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

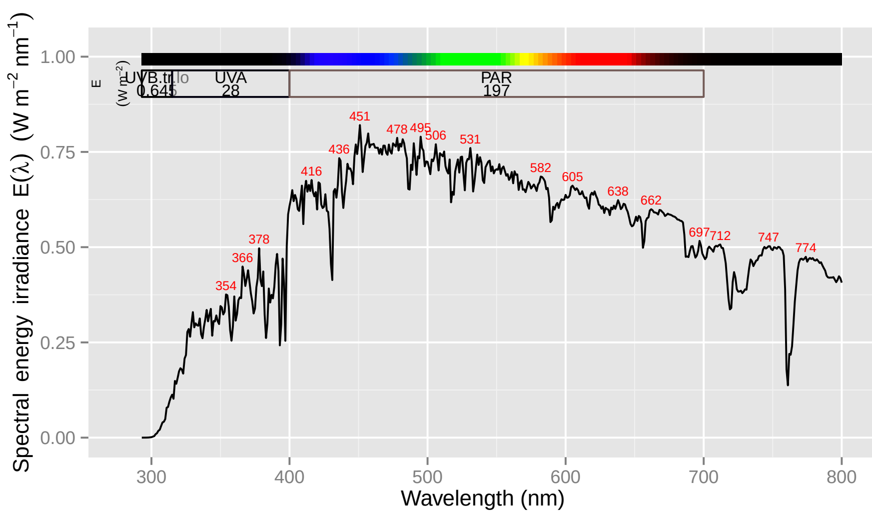
For the `source_spct` example we generate four independent noisy replicates of the same solar spectrum, to demonstrate parallel summaries. Under normal use we would use `rbindspct` to bind the different true replicate measured spectra.

```
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 easily modified as needed, but they do not preserve all the attributes of the original spectra.

11 Astronomy

11.1 Position of the sun

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] "2015-07-19 00:46:51 EEST"
##
```

```
## $azimuth
## [1] 323.6478
##
## $elevation
## [1] -25.81009
##
## $diameter
## [1] 0.5246564
##
## $distance
## [1] 1.016284

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
```

11.2 Times of sunrise, solar noon and sunset

Functions `sunrise_time`, `sunset_time`, `noon_time`, `day_length` and `night_length` have all the same parameter signature. In addition, function `day_night` returns a list containing all the quantities returned by the other functions. They are all vectorized for the `date` parameter.

We create a vector of dates to use in the examples—default time zone of `ymd` is UTC or GMT.

```
dates <- seq(from = ymd("2015-03-01"), to = ymd("2015-07-1"), length.out = 3)
```

Default latitude is zero (the Equator), the default longitude is zero (Greenwich), and default time zone for the functions in the `photobiology` package is `"UTC"`. Be also aware that for summer dates the times are expressed accrodingly. In the examples below this can be recognized for example, by the time zone being reported as EEST instead of EET for Eastern Europe.

```
noon_time(dates, tz = "UTC", lat =  60)

## [1] "2015-03-01 12:12:48 UTC" "2015-05-01 11:57:17 UTC"
## [3] "2015-07-01 12:03:44 UTC"

noon_time(dates, tz = "CET", lat =  60)
```

```
## [1] "2015-03-01 13:12:48 CET"  "2015-05-01 13:57:17 CEST"
## [3] "2015-07-01 14:03:44 CEST"
```

```r
day_night(dates, lat =  60)

## $day
## [1] "2015-03-01" "2015-05-01" "2015-07-01"
##
## $sunrise
## [1] "2015-03-01 07:06:26 UTC" "2015-05-01 04:06:51 UTC"
## [3] "2015-07-01 02:52:50 UTC"
##
## $noon
## [1] "2015-03-01 12:12:48 UTC" "2015-05-01 11:57:17 UTC"
## [3] "2015-07-01 12:03:44 UTC"
##
## $sunset
## [1] "2015-03-01 17:19:30 UTC" "2015-05-01 19:49:05 UTC"
## [3] "2015-07-01 21:14:09 UTC"
##
## $daylength
## [1] 10.21778 15.70382 18.35536
##
## $nightlength
## [1] 13.782215  8.296180  5.644636
```

The default for `date` is the current day.

```r
sunrise_time(lat = 60)

## [1] "2015-07-19 03:20:30 UTC"
```

Both latitude and longitude can be supplied, but be aware that if the returned value is desired in the local time coordinates, the time zone should match the longitude.

```r
sunrise_time(today(tzone = "UTC"), tz = "UTC", lat = 60, lon = 0)

## [1] "2015-07-18 03:18:33 UTC"

sunrise_time(today(tzone = "EET"), tz = "EET", lat = 60, lon = 25)

## [1] "2015-07-19 04:40:22 EEST"
```

Southern hemisphere latitudes are given as negative numbers.

```r
sunrise_time(dates, lat =  60)

## [1] "2015-03-01 07:06:26 UTC" "2015-05-01 04:06:51 UTC"
## [3] "2015-07-01 02:52:50 UTC"

sunrise_time(dates, lat = -60)

## [1] "2015-03-01 05:18:13 UTC" "2015-05-01 07:47:52 UTC"
## [3] "2015-07-01 09:14:28 UTC"
```

The angle used in the twilight calculation can be supplied, either as the name of a standard definition, or as an angle in degrees (negative for sun positions below the horizon). Positive angles can be used when the time of sun occlusion behind a building, mountain, or other obstacle needs to be calculated.

```
sunrise_time(today(tzone = "EET"), tz = "EET", lat = 60, lon = 25,
             twilight = "civil")

## [1] "2015-07-19 03:16:31 EEST"

sunrise_time(today(tzone = "EET"), tz = "EET", lat = 60, lon = 25,
             twilight = -10)

## [1] NA

sunrise_time(today(tzone = "EET"), tz = "EET", lat = 60, lon = 25,
             twilight = +12)

## [1] "2015-07-19 06:36:13 EEST"
```

Parameter `unit.out` can be used to obtain the returned value expresed as time-of-day in hours, minutes, or seconds since midnight.

```
sunrise_time(today(tzone = "EET"), tz = "EET", lat = 60, lon = 25,
             unit.out = "hour")

## [1] 4.672843
```

Functions `day_length` and `night_length` return by default the length of time in hours.

```
day_length(dates, lat = 60)

## [1] 10.21778 15.70382 18.35536

night_length(dates, lat = 60)

## [1] 13.782215  8.296180  5.644636
```

Function `day_night` returns a list.

```
day_night(dates, lat = 60)

## $day
## [1] "2015-03-01" "2015-05-01" "2015-07-01"
##
## $sunrise
## [1] "2015-03-01 07:06:26 UTC" "2015-05-01 04:06:51 UTC"
## [3] "2015-07-01 02:52:50 UTC"
##
## $noon
## [1] "2015-03-01 12:12:48 UTC" "2015-05-01 11:57:17 UTC"
## [3] "2015-07-01 12:03:44 UTC"
##
```

```
## $sunset
## [1] "2015-03-01 17:19:30 UTC" "2015-05-01 19:49:05 UTC"
## [3] "2015-07-01 21:14:09 UTC"
##
## $daylength
## [1] 10.21778 15.70382 18.35536
##
## $nightlength
## [1] 13.782215  8.296180  5.644636

day_night(dates, lat = 60, unit.out = "hour")

## $day
## [1] "2015-03-01" "2015-05-01" "2015-07-01"
##
## $sunrise
## [1] 7.107340 4.114251 2.880713
##
## $noon
## [1] 12.21353 11.95495 12.06228
##
## $sunset
## [1] 17.32512 19.81807 21.23608
##
## $daylength
## [1] 10.21778 15.70382 18.35536
##
## $nightlength
## [1] 13.782215  8.296180  5.644636
```

12 RGB colours

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.

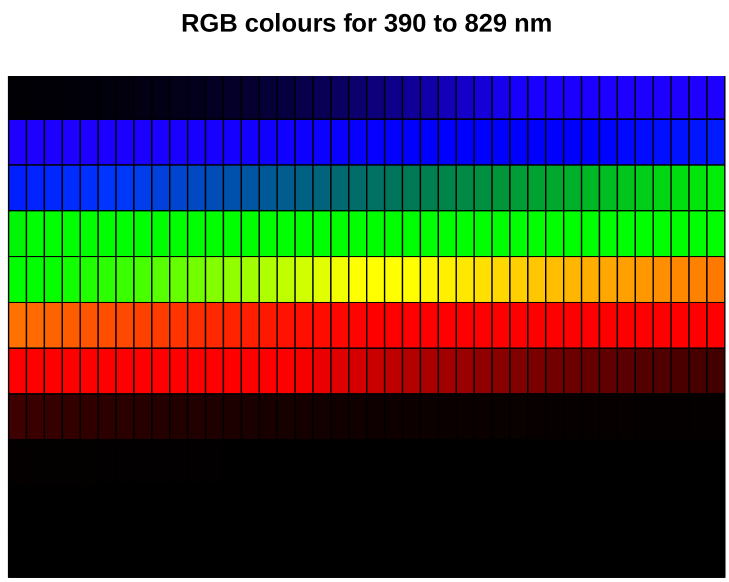
```
rgb_spct(sun.spct)

## [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"
```

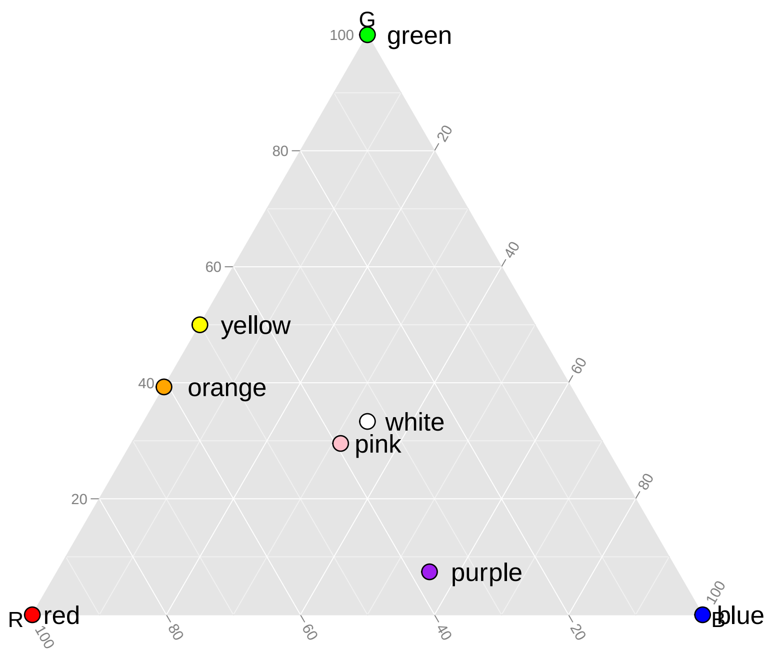
Here we plot the RGB colours for the range covered by the CIE 2006 proposed standard calculated at each 1 nm step:

```
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
```



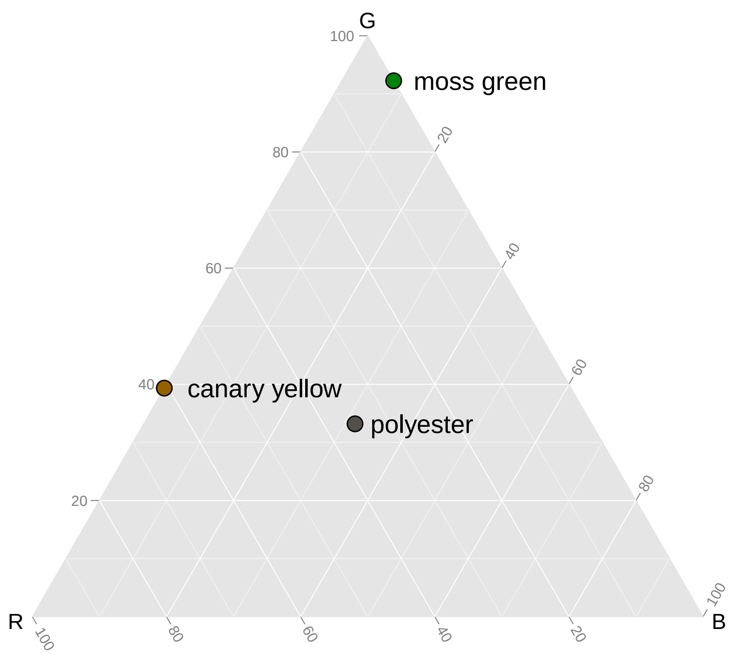
We simulate the spectra of filtered sunlight by multiplying the solar spectrum by filter transmittance spectra.

```
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
```



13 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 per-

Table 11: Data sets included in the package: spectra. The CIE standard illuminant data in this package are normalized to one at $\lambda = 560\,\mathrm{nm}$, while in the CIE standard they are normalized to 100 at the same wavelength.

Object	class	units	data description
sun.spct	source_spct	$\mathrm{W\,m^{-2}\,nm^{-1}}$	solar spectral irradiance
sun.daily.spct	source_spct	$\mathrm{J\,m^{-2}\,d^{-1}\,nm^{-1}}$	solar spectral exposure
sun.data	data.frame	$\mathrm{W\,m^{-2}\,nm^{-1}}$	solar spectral irradiance
sun.daily.data	data.frame	$\mathrm{J\,m^{-2}\,d^{-1}\,nm^{-1}}$	solar spectral exposure
D65.illuminant.spct	source_spct	(norm. 560 nm)	CIE standard
A.illuminant.spct	source_spct	(norm. 560 nm)	CIE standard

Table 12: Data sets included in the package: chromaticity data

Object	class	data description
ciexyzCC2.spct	chroma_spct	human chromaticity coordinates 2°
ciexyzCC10.spct	chroma_spct	human chromaticity coordinates 10°
ciexyzCMF2.spct	chroma_spct	human colour matching function 2°
ciexyzCMF10.spct	chroma_spct	human colour matching function 10°
ciev2.spct	chroma_spct	human luminous efficiency 2°
ciev10.spct	chroma_spct	human luminous efficiency 10°
beesxyzCMF.spct	chroma_spct	bee colour matching function

formance 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.

14 Example data

A few example spectra are included in this package for use in examples and vignettes, and testing (Tables 11 and 12).