photobiology Version 0.4.0
User Guide

Pedro J. Aphalo

November 3, 2014

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 (photobiologyUV), visible radiation (photobiologyVIS), or based on Phytochrome (photobiologyPhy), Cryptochrome (photobiologyCry) (both photoreceptors present in plants), and spectral data for filters (photobiologyFilters). In the future it will be submitted to CRAN (Comprehensive R archive network), it is meanwhile available from `https://bitbucket.org/aphalo/photobiology/downloads`. There is also a public Git repository at `https://bitbucket.org/aphalo/photobiology` from where the source code of the current an earlier versions can be cloned.

2 Installation and use

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

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

```
library(photobiology)
library(photobiologyWavebands)
library(photobiologyFilters)
library(ggplot2)
library(ggtern)
```

3 Data formats

The package provides two sets of functions for many operations: functions programmed following a functional paradigm, and functions using an object-oriented paradigm. The former functions take as arguments numeric vectors

and are probably faster. The later ones take 'spectra' objects as arguments, are easier to use, and at least at the moment, to some extent slower. For 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.

3.1 Example spectral data

A few data objects are included in the package for use in examples. Two simulated solar spectra, `sun.spct` and `sun.daily.spct`, ...

3.2 Using numeric vectors

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

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

3.3 Using 'spectra' objects

This package defines a family of objects based on data tables (data frame compatible objects) which impose some restrictions on the naming of the vectors, something that allows the second set of functions to 'find' the data when passed one of these objects as argument. In addition, as the data is checked when the object is built, there is no need to test for the validity of the data each time a calculation is carried out. The other advantage of using `spct` objects, is that specialized versions of generic functions like `print` and operators like `+` can be defined for spectra. `___.spct` objects are derived from data.table objects, which in turn are derived from data.frame objects. In this package we define a *generic* spectrum type of object, derived from data table, from which specialized types of spectra are derived. This 'parenthood' hierarchy means that spectra objects can be used almost anywhere where a `data.frame` or `data.table` is expected. Many functions defined in package `data.table` and useful when working with spectra are re-exported by the current package, so that in most cases there is no need to explicitly include `library(data.table)` in your scripts.

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.

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

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

a.df # not modified!

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

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

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

a.dt # not modified!

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

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

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

a.dt # modified!

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

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

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

a.dt # not modified!

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

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

3.4 Spectral data assumptions

An assumption of the package is that wavelengths are always expressed in nanometres (1 nm $= 1 \cdot 10^{-9}$m). If the data to be analysed uses different units for wavelengths, e.g. Ånstrom (1 Å $= 1 \cdot 10^{-10}m$), the values need to be rescaled before any calculations. **Not doing so will yield completely wrong results!**

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.

If our spectral irradiance data is in $\mathrm{W\,m^{-2}\,nm^{-1}}$, and the wavelength in nm, as in the case of 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 output by our model of spectroradiometer is in $\mathrm{mW\,m^{-2}\,nm^{-1}}$, and the wavelengths are in Ångstrom then to obtain the effective irradiance in $\mathrm{W\,m^{-2}}$ we will need to rescale the data.

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

In the example above, we take advantage of the behavior 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.

It is very 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.

3.5 Setting and querying the classe of a spectrum object

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

```
a.spct <- data.table(w.length = 300:305, y = rep(1,6))
class(a.spct)

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

class.spct(a.spct)

## character(0)

is.any.spct(a.spct)

## [1] FALSE

a.spct <- setGenSpct(a.spct)
class(a.spct)

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

class.spct(a.spct)

## [1] "generic.spct"

is.any.spct(a.spct)

## [1] TRUE

is.generic.spct(a.spct)

## [1] TRUE

is.filter.spct(a.spct)

## [1] FALSE

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, a columns called `w.length` (wavelength values expressed in nm) and `s.e.irrad` ($\mathrm{W\,m^{-2}\,nm^{-1}}$) and\or `s.q.irrad` ($\mathrm{mol\,m^{-2}\,s^{-1}\,nm^{-1}}$) must be present.

```
b.spct <- setSourceSpct(data.table(w.length = 300:305, s.e.irrad = rep(1,6)))
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 can be used when printing the results.

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

## [1] "day"

class(b.d.spct)

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

b.d.spct

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

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

```
c.spct <- setFilterSpct(data.table(w.length = 300:305, Tfr = rep(1,6)))
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
```

`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 = rep(1,6)))
class(d.spct)

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

d.spct

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

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

```
e.spct <- setChromaSpct(data.table(w.length = 300:305, x = rep(1,6),
                                   y = rep(1,6), z = rep(1,6)))
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, and if not with NAs.

```
f.spct <- setReflectorSpct(data.table(w.length = 300:305, Rpc = rep(100,6)))
class(f.spct)

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

f.spct

##    w.length Rpc Rfr
## 1:      300 100   1
## 2:      301 100   1
## 3:      302 100   1
## 4:      303 100   1
## 5:      304 100   1
## 6:      305 100   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 = rep(1,6)))

## Warning in check.reflector.spct(x):  No reflectance data found in filter.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 `w.length` column is present, and a `wl` column is found, it is renamed to `w.length`.

```
h.spct <- setReflectorSpct(data.table(wl = 300:305, Rfr = rep(1,6)))
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
```

Here we just use the example data supplied with 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
##   3:      295 2.176175e-05 5.366385e-11
##   4:      296 6.780119e-05 1.677626e-10
##   5:      297 1.533491e-04 3.807181e-10
##  ---
## 504:      796 4.080616e-01 2.715219e-06
## 505:      797 4.141204e-01 2.758995e-06
## 506:      798 4.236281e-01 2.825879e-06
## 507:      799 4.185850e-01 2.795738e-06
## 508:      800 4.069055e-01 2.721132e-06
```

3.6 Conversions

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

```
b.spct

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

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

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

b1.spct

##    w.length    s.q.irrad
## 1:      300 2.507767e-06
## 2:      301 2.516127e-06
```

```
## 3:      302 2.524486e-06
## 4:      303 2.532845e-06
## 5:      304 2.541204e-06
## 6:      305 2.549564e-06

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

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

b2.spct

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

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

3.7 Remapping a spectrum to different wavelengths

Converting spectra available at a given set of wavelengths values to a different one, is frequently needed when operating with several spectra of different origin. One can increase the *apparent* resolution by interpolation, and reduce it by local averaging or smoothing and resampling. The same function works on all `spct` objects, interpolating every column except `w.length` and replacing in this last column the old wavelength values with the new ones supplied as argument. The optional argument `fill.value` control what value is assigned to wavelengths in the new data that are outside the range of the new 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
##    3:    400.2 0.6117187 2.046445e-06
##    4:    400.3 0.6135257 2.053010e-06
##    5:    400.4 0.6153326 2.059575e-06
##   ---
##  997:    499.6 0.7244659 3.025562e-06
##  998:    499.7 0.7243740 3.025784e-06
##  999:    499.8 0.7242821 3.026006e-06
## 1000:    499.9 0.7241901 3.026228e-06
## 1001:    500.0 0.7240982 3.026450e-06
```

3.8 Summaries

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

```
integrate_spct(sun.spct)

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

average_spct(sun.spct)

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

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

```
range(sun.spct)

## [1] 293 800

min(sun.spct)

## [1] 293

max(sun.spct)

## [1] 800

midpoint(sun.spct)

## [1] 546.5

spread(sun.spct)

## [1] 507

stepsize(sun.spct)

## [1] 1 1
```

Method definitions for `summary` and 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 with using the correct units.

```
summary(sun.spct)

## wavelength ranges from 293 to 800 nm
## largest wavelength step size is 1 nm
## spectral irradiance ranges from 2.61e-06 to 0.8205 W m-2 nm-1
## energy irradiance is 269.1 W m-2
## photon irradiance is 1255 umol s-1 m-2
```

```
summary(sun.daily.spct)

## wavelength ranges from 290 to 800 nm
## largest wavelength step size is 1 nm
## spectral irradiance ranges from 0 to 32.61 kJ d-1 m-2 nm-1
## energy irradiance is 10.93 MJ m-2
## photon irradiance is 51.39 mol d-1 m-2
```

4 Using operators with spectra

The basic math 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.

```
sun.spct / sun.spct

##      w.length s.e.irrad
##   1:      293         1
##   2:      294         1
##   3:      295         1
##   4:      296         1
##   5:      297         1
##  ---
## 504:      796         1
## 505:      797         1
## 506:      798         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
##   3:      295 4.352350e-05
##   4:      296 1.356024e-04
##   5:      297 3.066981e-04
##  ---
## 504:      796 8.161233e-01
## 505:      797 8.282407e-01
## 506:      798 8.472561e-01
## 507:      799 8.371699e-01
## 508:      800 8.138111e-01
```

Operators are also defined for operations between an 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
```

```
##   3:      295 4.352350e-05
##   4:      296 1.356024e-04
##   5:      297 3.066981e-04
##  ---
## 504:      796 8.161233e-01
## 505:      797 8.282407e-01
## 506:      798 8.472561e-01
## 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
##   3:      295 0.000000e+00
##   4:      296 6.780119e-05
##   5:      297 0.000000e+00
##  ---
## 504:      796 4.080616e-01
## 505:      797 0.000000e+00
## 506:      798 4.236281e-01
## 507:      799 0.000000e+00
## 508:      800 4.069055e-01
```

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

5 Defining wavebands

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

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

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

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

6 Calculating irradiance or exposure

There is one basic function for these calculations `irradiance()`, it 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.

Then we compare the calculations based on the different wavebands defined in the previous section and the predefined functions in the packages photobiologyVIS and photobiologyUV. The predefined functions have the advantage of allowing the specification of parameters to modify the `w.band` created. In the example bellow, we use this to set the normalization wavelength.

This is how `CIE()` is defined in photobiologyUV.

```
CIE <- function(norm=298) {
  new_waveband(w.low=250, w.high=400,
               weight="SWF", SWF.e.fun=CIE.e.fun, SWF.norm=298,
               norm=norm, hinges=c(249.99, 250, 298, 328, 399.99, 400),
               wb.name=paste("CIE98", as.character(norm), sep="."))
}
```

The generic functions `print()`, `min()`, `max()`, `range()`, `midpoint()` and `labels()` call the corresponding special functions defined for waveband objects: `print.waveband()`, `range.waveband()`, etc.

```
PAR()

## PAR
## low (nm) 400
## high (nm) 700
## weighted none
```

```r
str(PAR())
```

```
## List of 10
##  $ low      : num 400
##  $ high     : num 700
##  $ weight   : chr "none"
##  $ SWF.e.fun: NULL
##  $ SWF.q.fun: NULL
##  $ SWF.norm : NULL
##  $ norm     : NULL
##  $ hinges   : num [1:4] 400 400 700 700
##  $ name     : chr "PAR"
##  $ label    : chr "PAR"
##  - attr(*, "class")= chr [1:2] "waveband" "list"
```

```r
range(PAR())
```

```
## [1] 400 700
```

```r
spread(PAR())
```

```
## [1] 300
```

```r
min(PAR())
```

```
## [1] 400
```

```r
max(PAR())
```

```
## [1] 700
```

```r
midpoint(PAR()) # gives wavelength at center
```

```
## [1] 550
```

```r
normalization(PAR())
```

```
## [1] NA
```

```r
color(PAR()) # equivalent RGB color definitions
```

```
## $CMF
##   PAR.CMF
## "#735B57"
##
## $CC
##    PAR.CC
## "#8D4C3D"
```

```r
labels(PAR())
```

```
## $label
## [1] "PAR"
##
## $name
## [1] "PAR"
```

```
CIE()
```

```
## CIE98.298
## low (nm) 250
## high (nm) 400
## weighted SWF
## normalized at 298 nm
```

```
str(CIE())
```

```
## List of 10
##  $ low      : num 250
##  $ high     : num 400
##  $ weight   : chr "SWF"
##  $ SWF.e.fun:function (w.length)
##  $ SWF.q.fun:function (w.length)
##  $ SWF.norm : num 298
##  $ norm     : num 298
##  $ hinges   : num [1:6] 250 250 298 328 400 ...
##  $ name     : chr "CIE98.298"
##  $ label    : chr "CIE98"
##  - attr(*, "class")= chr [1:2] "waveband" "list"
```

```
labels(CIE())
```

```
## $label
## [1] "CIE98"
##
## $name
## [1] "CIE98.298"
```

```
labels(CIE(300))
```

```
## $label
## [1] "CIE98"
##
## $name
## [1] "CIE98.300"
```

```
normalization(CIE())
```

```
## [1] 298
```

```
normalization(CIE(300))
```

```
## [1] 300
```

```
labels(UVB())
```

```
## $label
## [1] "UVB"
##
## $name
## [1] "UVB.ISO"
```

```
labels(UVB("medical"))
```

```
## $label
## [1] "UVB"
##
## $name
## [1] "UVB.medical"

labels(UVB("ISO"))

## $label
## [1] "UVB"
##
## $name
## [1] "UVB.ISO"
```

```
labels(new_waveband(100, 200))

## $label
## [1] "range.100.200"
##
## $name
## [1] "range.100.200"

labels(new_waveband(100, 200, wb.name="ANY NAME"))

## $label
## [1] "ANY NAME"
##
## $name
## [1] "ANY NAME"

labels(new_waveband(100, 200, wb.name="ANY SHORT NAME",
                    wb.label="ANY LONG LABEL"))

## $label
## [1] "ANY LONG LABEL"
##
## $name
## [1] "ANY SHORT NAME"
```

6.1 Irradiances from vectors

Here we describe the functions that use vectors as arguments. In the next section we describe similar functions for `spct` objects.

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

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

##          PAR
## 0.0008937598

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

##          PAR
## 0.0008937598
```

The recommended practice is to use `with`, as above. But to keep code lines shorter and simpler we attach the data frame `sun.data` to make the contained vectors visible. The use of `w.length` and `s.e.irrad` and7or `s.q.irrad` is just our convention, and vectors of any name can be supplied as arguments.

```
attach(sun.data)
```

Some examples comparing different definitions of wavebands for PAR. Photon irradiance is expressed in $\mathrm{mol\,m^{-2}\,s^{-1}}$.

```
photon_irradiance(w.length, s.e.irrad, my_PAR)

## range.400.700
##  0.0008937598

photon_irradiance(w.length, s.e.irrad, my_PARx)

##      my_PARx
## 0.0008937598

photon_irradiance(w.length, s.e.irrad, PAR())

##          PAR
## 0.0008937598

photon_irradiance(w.length, s.e.irrad, PAR()) * 1e6

##      PAR
## 893.7598
```

Some examples comparing different definitions of wavebands for erythemal UV-B (CIE98) irradiance in $\mathrm{W\,m^{-2}}$.

```
photon_irradiance(w.length, s.e.irrad, my_CIE_1)

## range.250.400.wtd
##      2.037119e-07

photon_irradiance(w.length, s.e.irrad, my_CIE_2)

## range.250.400.wtd
##      2.037119e-07

photon_irradiance(w.length, s.e.irrad, my_CIE_3)

## range.250.400.wtd
##      2.037119e-07

energy_irradiance(w.length, s.e.irrad, my_CIE_1)

## range.250.400.wtd
##        0.08177754

energy_irradiance(w.length, s.e.irrad, my_CIE_2)
```

```
## range.250.400.wtd
##        0.08177754

energy_irradiance(w.length, s.e.irrad, my_CIE_3)

## range.250.400.wtd
##        0.08177754

energy_irradiance(w.length, s.e.irrad, CIE())

##  CIE98.298
## 0.08177754

energy_irradiance(w.length, s.e.irrad, CIE(298))

##  CIE98.298
## 0.08177754

energy_irradiance(w.length, s.e.irrad, CIE(300))

## CIE98.300
## 0.1260765
```

Lists of wavebands are also accepted as argument.

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

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

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

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

The are also 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.

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

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

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

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

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

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

Now we detach the data frame.

```
detach(sun.data)
```

6.2 Irradiances from spectra

The code using `spct` objects is much simpler.

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

##      PAR
## 196.7004
## attr(,"time.unit")
## [1] "second"

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

##      PAR
## 893.7598
## attr(,"time.unit")
## [1] "second"

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

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

7 Calculating ratios

7.1 Ratios from vectors

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

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

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.

We attach again the data frame.

```
attach(sun.data)
```

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

## [1] 0.001708359

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

## [1] 0.002989897
```

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 <- photons_energy_ratio(w.length, s.e.irrad, PAR())

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

## [1] 4.543762e-06

print(PPFD.mol.photon * 1e6)

## [1] 1000

print(PAR.energy)

## [1] 220.082
```

The 'ratio' functions for vectors do not accept lists of waveband objects as the 'irradiance' functions do. This is a feature, as otherwise it would be too easy to make mistakes. It is possible to use the 'irradiance' functions to calculate several ratios in one go, or use the functions that calculate ratios from spectral objects.

```
ratios <- photon_irradiance(w.length, s.e.irrad,
                            list(UVC=UVC(), UVB=UVB(), UVA=UVA())) /
  photon_irradiance(w.length, s.e.irrad, PAR())
ratios

##         UVC         UVB         UVA
## 0.000000000 0.001708359 0.093930758

names(ratios) <- paste(names(ratios), ":PAR", sep="")
ratios

##     UVC:PAR     UVB:PAR     UVA:PAR
## 0.000000000 0.001708359 0.093930758
```

7.2 Ratios from spectra

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

## UVB.ISO:PAR(q:q)
##      0.001708359
## attr(,"time.unit")
## [1] "second"

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

## UVC.ISO:UV.ISO(q:q) UVB.ISO:UV.ISO(q:q) UVA.ISO:UV.ISO(q:q)
##          0.00000000          0.01786255          0.98213745
## attr(,"time.unit")
## [1] "second"

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

## UVB.ISO:UV.ISO(q:q)    UVB.ISO:PAR(q:q)
##         0.017862550         0.001708359
## attr(,"time.unit")
## [1] "second"
```

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

## UVB.ISO:PAR(e:e)
##      0.002989897
## attr(,"time.unit")
## [1] "second"

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

## UVC.ISO:UV.ISO(e:e) UVB.ISO:UV.ISO(e:e) UVA.ISO:UV.ISO(e:e)
##          0.00000000          0.02076332          0.97923668
## attr(,"time.unit")
## [1] "second"
```

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

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

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

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

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

## e:q(PAR)
##   220082
## attr(,"time.unit")
## [1] "second"

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

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

7.3 Tagging observations in a spectrum

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

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

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

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

## $time.unit
## [1] "second"
##
## $wb.key.name
## [1] "Bands"
##
## $wl.color
## [1] TRUE
##
## $wb.color
## [1] TRUE
##
## $wb.num
## [1] 1
##
## $wb.colors
## $wb.colors[[1]]
##   PAR.CMF
## "#735B57"
##
##
## $wb.names
## [1] "PAR"
```

```
##
## $wb.list
## $wb.list[[1]]
## PAR
## low (nm) 400
## high (nm) 700
## weighted none
```

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

```
wb2tagged_spct(UV_bands())
wb2rect_spct(UV_bands())
```

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 ot the wavebands.

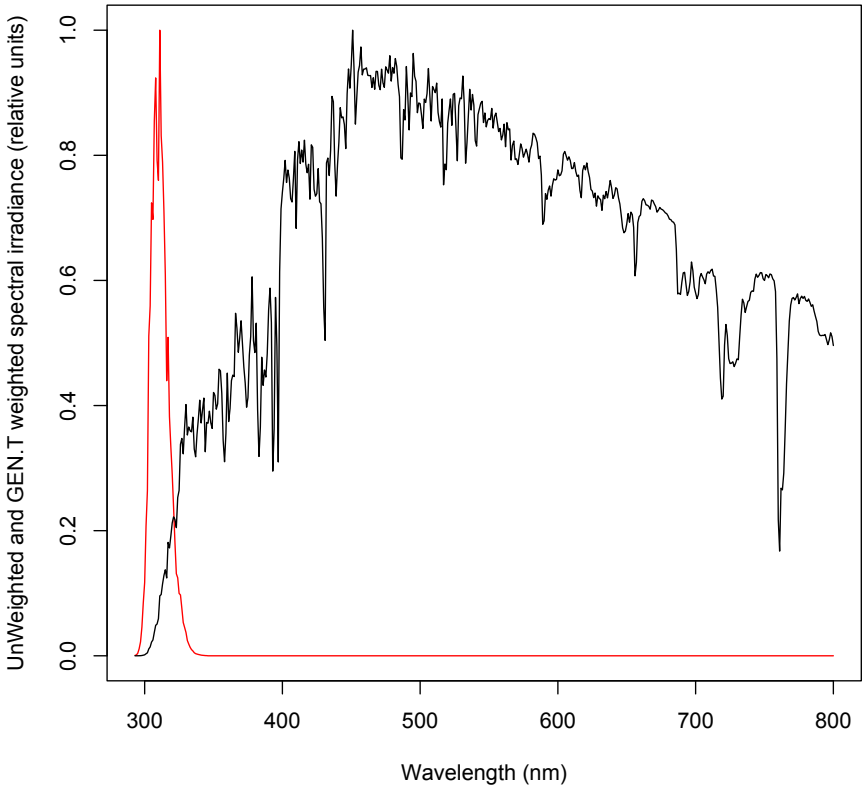
8 Calculating weighted spectral irradiances

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

8.1 Weighted spectral irradiance from vectors

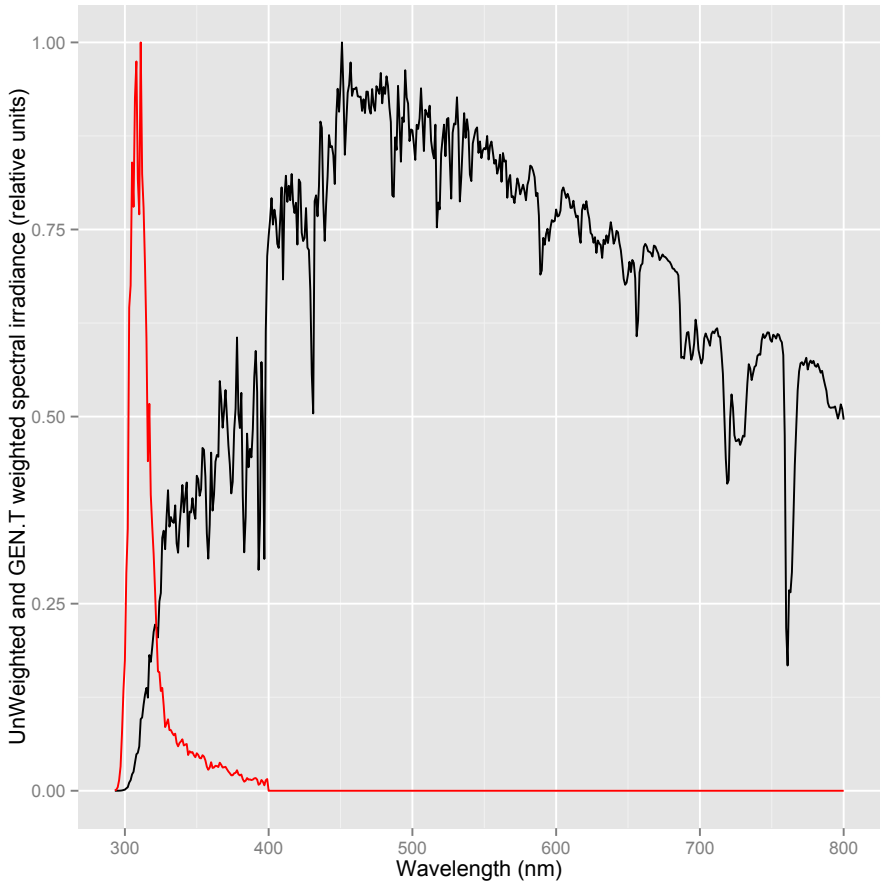
We here plot weighted and unweighted irradiances using simulated solar spectral irradiance data.

```
weighted.s.e.irrad <- s.e.irrad * calc_multipliers(w.length, GEN.T())
plot(weighted.s.e.irrad/max(weighted.s.e.irrad) ~ w.length, type="l", col="red",
     xlab="Wavelength (nm)",
     ylab="UnWeighted and GEN.T weighted spectral irradiance (relative units)")
lines(s.e.irrad/max(s.e.irrad) ~ w.length, col="black")
```



Or using spectral objects and ggplot.

```
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 GEN.T weighted spectral irradiance (relative units)")
```



8.2 Weighted spectral irradiance from an unweighted source spectrum

The multiplication operator is defined for operation between a `source.spct` and a `waveband`.

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

##      w.length    s.e.irrad
##   1:      293 2.609665e-06
##   2:      294 6.142401e-06
##   3:      295 2.176175e-05
##   4:      296 6.780119e-05
##   5:      297 1.533491e-04
##  ---
## 504:      796 0.000000e+00
## 505:      797 0.000000e+00
```

```
## 506:      798 0.000000e+00
## 507:      799 0.000000e+00
## 508:      800 0.000000e+00
```

9 Position of the sun in the sky

In photobiology research we sometimes need to calculate the position on the sun at arbitrary locations and positions. The function `sun_angles` returns the azimuth in degrees eastwards, altitude in degrees above the horizon, solar disk diameter in degrees and sun to earth distance in astronomical units. The time should be 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] "2014-11-03 00:15:32 EET"
##
## $azimuth
## [1] 308.4216
##
## $elevation
## [1] -62.47793
##
## $diameter
## [1] 0.5374272
##
## $distance
## [1] 0.9921345

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
```

10 Calculating times of sunrise and sunset

```
day_night()
```

```
## $day
## [1] "2014-11-03 EET"
##
## $sunrise
## [1] "2014-11-03 07:43:33 EET"
##
## $noon
## [1] "2014-11-03 13:43:28 EET"
##
## $sunset
## [1] "2014-11-03 19:43:33 EET"
##
## $daylength
## Time difference of 12.00001 hours
##
## $nightlength
## Time difference of 11.99999 hours
```

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

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

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

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

11 Calculating equivalent RGB colours for display

Two functions allow calculation of simulated colour of light sources as R colour definitions. Three different functions are available, one for monochromatic light taking as argument wavelength values, and one for polychromatic light taking as argument spectral energy irradiances and the corresponding wave length values. The third function can be used to calculate a representative RGB colour for a band of the spectrum represented as a range of wavelength, based on the assumption of a flat energy irradiance across the range. By default CIE coordinates for *typical* human vision are used, but the functions have a parameter that can be used for supplying a different chromaticity definition.

Examples for monochromatic light:

```
w_length2rgb(550) # green

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

w_length2rgb(630) # red

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

w_length2rgb(380) # UVA

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

w_length2rgb(750) # far red

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

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

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

Examples for wavelength ranges:

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

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

w_length_range2rgb(400:700)

## Using only extreme wavelength values.

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

```
w_length_range2rgb(sun.spct$w.length)

## Using only extreme wavelength values.

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

w_length_range2rgb(550)

## Calculating RGB values for monochromatic light.

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

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

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

## [1] "#544F4B"

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

## [1] "#544F4B"

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

## [1] "#59534F"

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

## [1] "#B63C37"

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

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

Examples with `source.spct` objects.

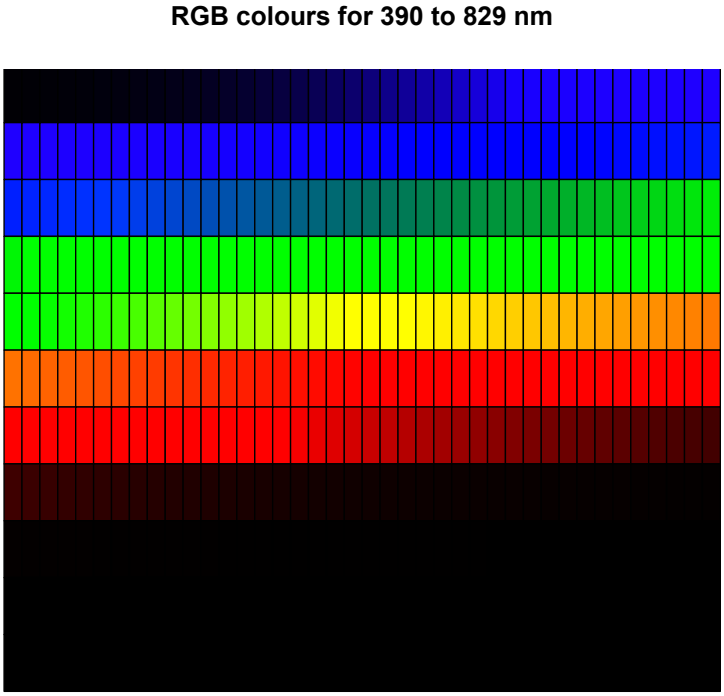
```
rgb_spct(sun.spct)

## [1] "#544F4B"

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

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

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

```
color(sun.spct)

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

color(sun.spct * rg630.spct)

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

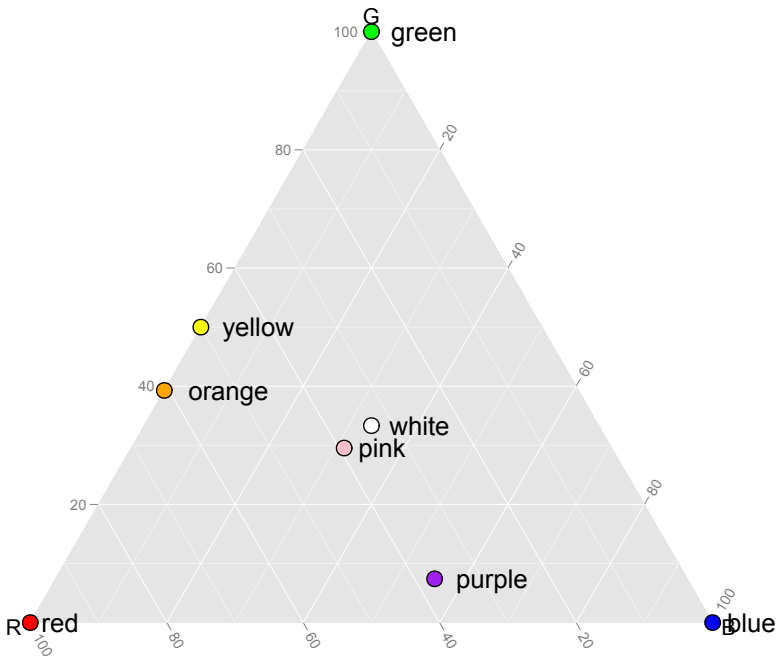
Here we plot the RGB colours for the range covered by the CIE 2006 proposed standard calculated at each 1 nm step:

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

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

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

plot( c(1,colCount), c(0,rowCount), type="n", ylab="", xlab="",
      axes=FALSE, ylim=c(rowCount,0))
title(paste("RGB colours for", as.character(wl[1]),
            "to", as.character(wl[2]), "nm"))

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



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

```
colours <- c("red", "green", "yellow", "white", "orange",
             "blue", "pink", "purple")
rgb.values <- col2rgb(colours)
test.data <-
  data.frame(colour=colours,
             R=rgb.values[1, ], G=rgb.values[2, ], B=rgb.values[3, ])
maxwell.tern <- ggtern(data=test.data,
                       aes(x=R, y=G, z=B, label=colour, fill=colour)) +
  geom_point(shape=21, size=4) + geom_text(hjust=-0.3) +
  labs(x = "R", y="G", z="B") + scale_fill_identity()
maxwell.tern
```



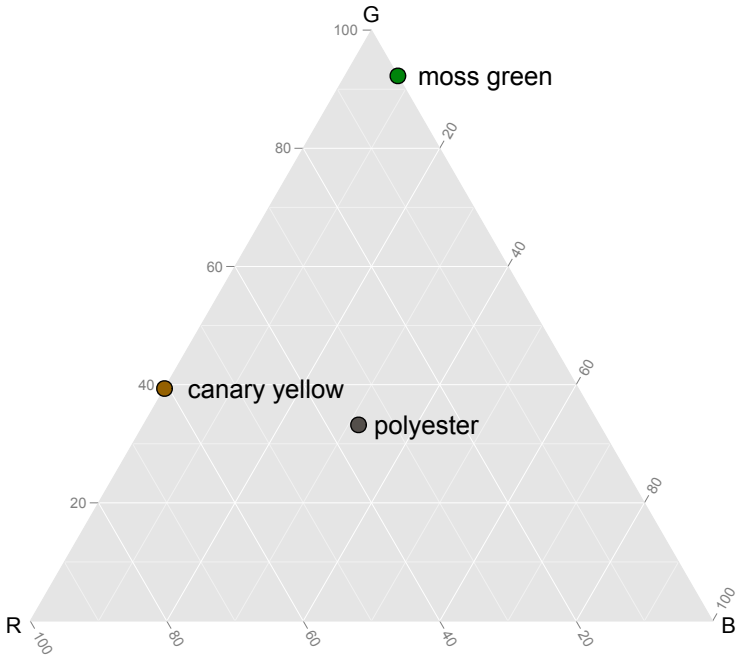
We simulate the spectra of filtered sunlight by multiplying the solar spectrum by filter transmittance spectra.

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

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

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

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



12 Auxiliary functions

Sometimes it is needed to add (parallel sum) between two spectra, even if the two spectra have been measured at different wavelengths or wavelength steps. This can happen for example when we would like to calculate the spectrum of a combination of two light sources from the individual spectra. The function `sum_spectra()` can be used in such a case. It's use is described in User Guide of the photobiologyLamps package.

A function `interpolate_spectrum` is also included to facilitate interpolation

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

13 Dealing with real 'noisy' spectral data

The first thing to do is to think whether any part of the spectral measurements can be *a priori* known to be equal to zero. For example for the solar spectrum at ground level it is safe to assume that the spectral irradiance is zero for all wavelengths shorter than 293 nm. If the data are noisy, it is best to discard these data before calculating any effective UV doses.

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)

##    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:      298 3.669677e-04 9.141345e-10
```

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):

```
subsetted.sun.spct <- subset(sun.spct, w.length >= 297)
head(subsetted.sun.spct)

##    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
```

And then just use the sub-setted data in your calculations.

If the data are not in a dataframe, then there are 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()`, and just use the same data as in earlier examples. First we delete all data for wavelengths shorter than 293 nm.

13.1 Trimming spectra stored as vectors

```
trimmed.sun.spct <-
  trim_tails(w.length, s.e.irrad, low.limit=297)
head(trimmed.sun.spct)

##    w.length      s.irrad
## 1:      297 0.0001533491
## 2:      298 0.0003669677
## 3:      299 0.0007845430
## 4:      300 0.0012645540
## 5:      301 0.0026237179
## 6:      302 0.0039225827

tail(trimmed.sun.spct)

##    w.length   s.irrad
## 1:      795 0.4146957
## 2:      796 0.4080616
## 3:      797 0.4141204
## 4:      798 0.4236281
## 5:      799 0.4185850
## 6:      800 0.4069055
```

This function returns a new data.frame and uses always the same variable names for the columns.

```
trimmed.both.sun.spct <-
  trim_tails(w.length, s.e.irrad, low.limit=297, high.limit=550)
head(trimmed.both.sun.spct)

##    w.length      s.irrad
## 1:      297 0.0001533491
## 2:      298 0.0003669677
## 3:      299 0.0007845430
## 4:      300 0.0012645540
## 5:      301 0.0026237179
## 6:      302 0.0039225827

tail(trimmed.both.sun.spct)

##    w.length   s.irrad
## 1:      545 0.7272464
## 2:      546 0.6993041
## 3:      547 0.7119013
## 4:      548 0.6936565
## 5:      549 0.7020853
## 6:      550 0.7046551
```

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.na.sun.spct <- trim_tails(w.length, s.e.irrad, low.limit=297, fill=NA)
head(trimmed.na.sun.spct)

##    w.length      s.irrad
```

```
## 1:      293           NA
## 2:      294           NA
## 3:      295           NA
## 4:      296           NA
## 5:      297 0.0001533491
## 6:      298 0.0003669677
```

```
trimmed.both.na.sun.spct <-
  trim_tails(w.length, s.e.irrad, low.limit=297, high.limit=400, fill=NA)
head(trimmed.both.na.sun.spct)

##    w.length      s.irrad
## 1:      293           NA
## 2:      294           NA
## 3:      295           NA
## 4:      296           NA
## 5:      297 0.0001533491
## 6:      298 0.0003669677

tail(trimmed.both.na.sun.spct)

##    w.length s.irrad
## 1:      795      NA
## 2:      796      NA
## 3:      797      NA
## 4:      798      NA
## 5:      799      NA
## 6:      800      NA
```

In addition to NA we can supply an arbitrary numeric value.

```
trimmed.zero.sun.spct <-
  trim_tails(w.length, s.e.irrad, low.limit=297, fill=0.0)
head(trimmed.zero.sun.spct)

##    w.length      s.irrad
## 1:      293 0.0000000000
## 2:      294 0.0000000000
## 3:      295 0.0000000000
## 4:      296 0.0000000000
## 5:      297 0.0001533491
## 6:      298 0.0003669677
```

```
detach(sun.data)
```

13.2 Trimming spectral objects

We can use as above `low.limit`, `high.limit`, and `fill` as in the examples above.

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

We can in addition do the trimming with a `waveband` object.

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

By default trim_spct trims its argument in place, but as sun.spct is protected as part of the package, we need to first make a copy.

```
my_sun.spct <- copy(sun.spct)
trim_spct(my_sun.spct, PAR())
```

14 Optimizing performance

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

In the case of doing calculations repeatedly on the same spectrum, a small improvement in performance can be achieved by setting the parameter `check.spectrum=FALSE` for all but the first call to `irradiance()`, or `photon_irradiance()`, or `energy_irradiance()`, or the equivalent function for ratios. It is also possible to set this parameter to FALSE in all calls, and do the check beforehand by explicitly calling `check_spectrum()`.

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

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