photobiology Version 0.3.2
User Guide

Pedro J. Aphalo

June 21, 2014

1 Introduction

> This user guide is out of date. What is included is correct, but new functionality is not yet described.

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 photobiologyPhy 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(photobiologyUV)
library(photobiologyVIS)
```

3 Spectral data

If our spectral irradiance data is in W m-2 nm-1, and the wavelength in nm, as in the case of the Macam spectroradiometer, the functions will return the effective irradiance in W m-2. In this example we calculated a biologically effective irradiance.

If, for example, the spectral irradiance output by our model of spectroradiometer is in mW m-2 nm-1, and the wavelengths are in Ångstrom then to obtain the effective irradiance in W m-2we will need to convert the units.

```
energy_irradiance(wavelength/10,irrad/1000)
```

In this example, 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.

If the spectral irradiance is in mW m-2 nm-1then values should be multiplied by 10 to convert them to W m-2 nm-1.

It is very important to make sure that the wavelengths are in nanometres as this is what the functions expect. If the wavelengths are in the wrong units, the action spectra will be wrongly calculated, and the returned value for effective irradiance will be completely wrong.

Here we just use the example data supplied with the package.

4 Defining wavebands

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

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

```
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)
```

5 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.

We first load some data to play with.

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

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 wavebdand objects: `print.waveband()`, `range.waveband()`, etc.

```
PAR()

## PAR
## low (nm) 400
## high (nm) 700
```

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

```
## List of 10
##  $ low      : num 400
##  $ high     : num 700
##  $ weight   : NULL
##  $ 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"
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
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 NAME", wb.label="ANY LABEL"))

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

Now the example calculations.

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

## range.400.700
##     0.0008938

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

##   my_PARx
## 0.0008938

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

##       PAR
## 0.0008938

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

## range.250.400.wtd
##         2.037e-07

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

## range.250.400.wtd
##         2.037e-07
```

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

## range.250.400.wtd
##         2.037e-07

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

## range.250.400.wtd
##           0.08178

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

## range.250.400.wtd
##           0.08178

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

## range.250.400.wtd
##           0.08178

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

## CIE98.298
##   0.08178

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

## CIE98.298
##   0.08178

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

## CIE98.300
##    0.1261
```

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
##   0.08178   0.08178   0.12608

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

##   Red.ISO  Blue.ISO Green.ISO
##     79.61     37.58     49.30
```

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.

```r
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
##         28.32         69.63         68.53         58.54
## range.700.800
##         43.90

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

## range.400.500 range.500.600 range.600.700
##         35.40         34.84         29.76

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

## range.400.500 range.500.600 range.600.700
##         29.41         35.14         35.45
```

6 Calculating ratios

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.

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

## [1] 0.001708

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

## [1] 0.00299
```

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.

```r
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] 4.544e-06
```

```
print(PPFD.mol.photon * 1e6)

## [1] 1000

print(PAR.energy)

## [1] 220.1
```

The 'ratio' functions 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.

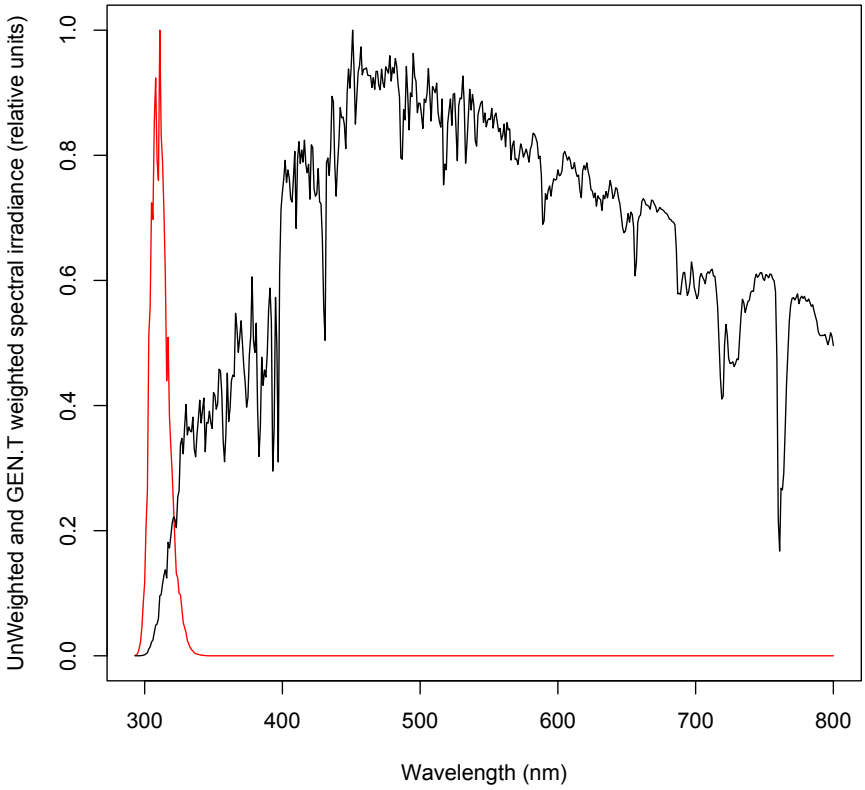
```
ratios <- photon_irradiance(w.length, s.e.irrad, list(UVC=UVC(), UVB=UVB(), UVA=UVA())) / photon_irradiance(w.len
ratios

##      UVC      UVB      UVA
## 0.000000 0.001708 0.093931

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

##  UVC:PAR  UVB:PAR  UVA:PAR
## 0.000000 0.001708 0.093931
```

7 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.

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")
```



8 Position of the sun in the sky

In photobiology research we sometimed 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] "2014-06-21 17:50:40 EEST"
##
## $azimuth
## [1] 265.3
##
```

```
## $elevation
## [1] 51.79
##
## $diameter
## [1] 0.5247
##
## $distance
## [1] 1.016
```

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

```
## $time
## [1] "2014-01-01 UTC"
##
## $azimuth
## [1] 182
##
## $elevation
## [1] -66.96
##
## $diameter
## [1] 0.5423
##
## $distance
## [1] 0.9833
```

9 Calculating times of sunrise and sunset

```
day_night()
```

```
## $sunrise
## [1] "2014-06-21 09:01:43 EEST"
##
## $noon
## [1] "2014-06-21 15:01:45 EEST"
##
## $sunset
## [1] "2014-06-21 21:01:49 EEST"
##
## $daylength
## Time difference of 12 hours
##
## $nightlength
## Time difference of 12 hours
```

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

```
## $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 13.78 hours
##
## $nightlength
## Time difference of 10.22 hours
```

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

```
## $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 12.77 hours
##
## $nightlength
## Time difference of 11.23 hours
```

10 Calculating equivalent RGB colous for display

Two functions allow calculation of simulated colour of light sources as R colour definitions. Three different functions are avaialble, 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 represeted as a range of wavelength, based on the assumption of a flat energy irradiance accreoss the range. By default CIE coordinates for *typical* human vision are used, but the functions have a parameter that can be used for suppying a different chromaticity definition.

Examples for monochromatic light:

```
w_length2rgb(550) # green
```

```
##    550 nm
## "#00FF00"
```

```
w_length2rgb(630) # red
```

```
##    630 nm
## "#FF0000"
```

```
w_length2rgb(380) # UVA
```

```
##    380 nm
## "#000000"

w_length2rgb(750) # far red

##    750 nm
## "#000000"

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

##    550 nm    630 nm    380 nm    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)

## Warning:  Using only extreme wavelength values.

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

w_length_range2rgb(sun.data$w.length)

## Warning:  Using only extreme wavelength values.

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

w_length_range2rgb(550)

## Warning:  Calculating RGB values for monochromatic light.

##    550 nm
## "#00FF00"
```

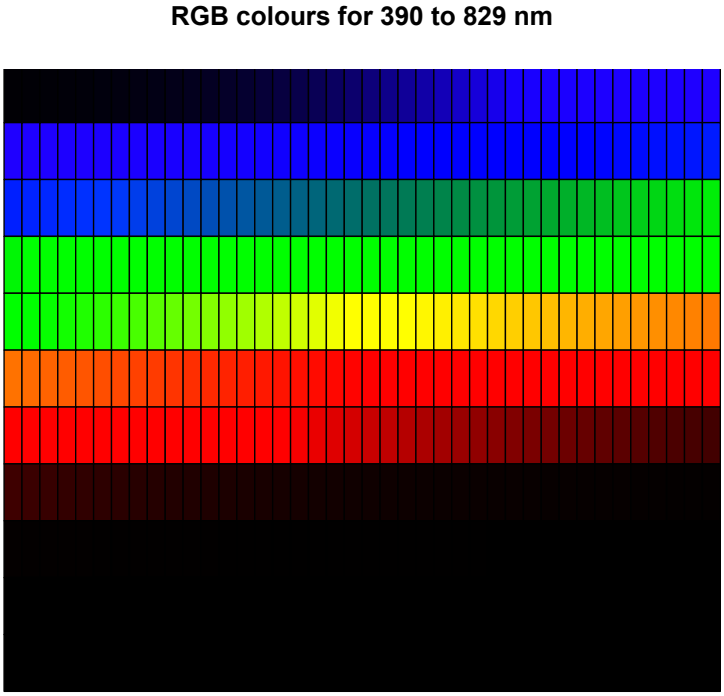
Examples for spectra, in this case the solar spectrum:

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

## [1] "#544F4B"

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

## [1] "#544F4B"

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

## [1] "#59534F"

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

```
## [1] "#B63C37"

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

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

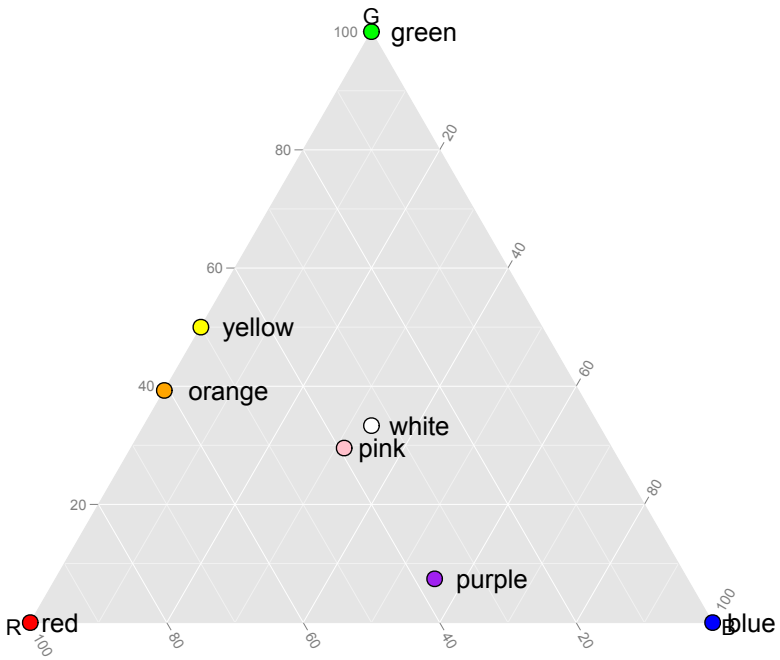
Here we plot the RGB colours for the range covered by the CIE 2006 proposed standard calculated at each 1 nm step:

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

my.colors <- w_length2rgb(wl[1]:wl[2])
# my.colors <- w_length2rgb(wl[1]:wl[2], sens=ciexyzCC2.data)
# my.colors <- w_length2rgb(wl[1]:wl[2], sens=ciexyzCMF2.data)

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)])
}
```



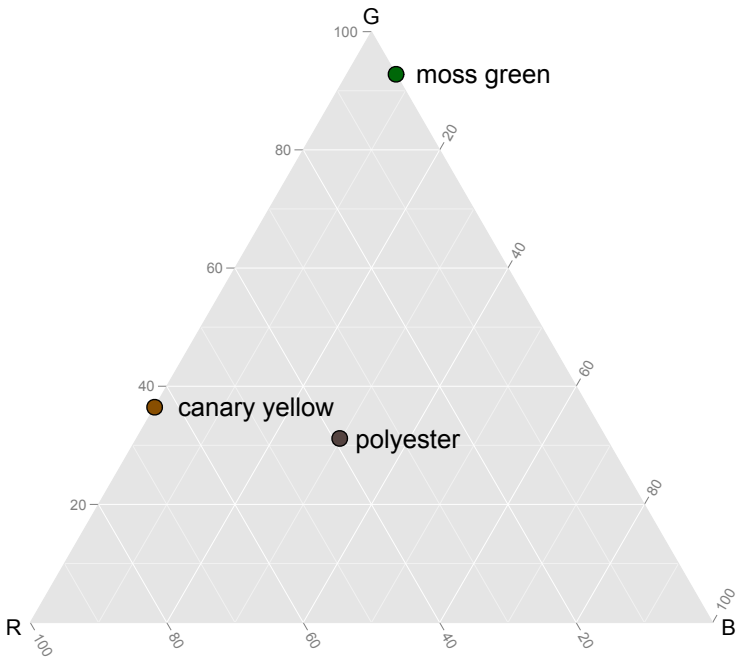
Given a color in any of the above ways, yields RGB values that can be used to locate the position of any colour on Maxwell's triangle. First using R's predifined colour.

```
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
  labs(x = "R", y="G", z="B") + scale_fill_identity()
maxwell.tern
```



Now using colours calculated from filter transmittance spectra.

```
yellow.filter <- with(canary.yellow.new.dt, s_e_irrad2rgb(w.length, Tfr, sens=ciexyzCMF2.data))
green.filter <- with(moss.green.new.dt, s_e_irrad2rgb(w.length, Tfr, sens=ciexyzCMF2.data))
polyester.filter <- with(polyester.new.dt, s_e_irrad2rgb(w.length, Tfr, sens=ciexyzCMF2.data))
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
maxwell.tern <- ggtern(data=test.data, aes(x=R, y=G, z=B, fill=colour, label=labels)) + geom_point(shape=21, size
  labs(x = "R", y="G", z="B")  + scale_fill_identity()
maxwell.tern
```



11 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.

12 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.data)

##    w.length s.e.irrad s.q.irrad
## 1:      293 2.610e-06 6.392e-12
## 2:      294 6.142e-06 1.510e-11
## 3:      295 2.176e-05 5.366e-11
## 4:      296 6.780e-05 1.678e-10
## 5:      297 1.533e-04 3.807e-10
## 6:      298 3.670e-04 9.141e-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.data <- subset(sun.data, w.length >= 297)
head(subsetted.sun.data)

##    w.length s.e.irrad s.q.irrad
## 1:      297 0.0001533 3.807e-10
## 2:      298 0.0003670 9.141e-10
## 3:      299 0.0007845 1.961e-09
## 4:      300 0.0012646 3.171e-09
## 5:      301 0.0026237 6.602e-09
## 6:      302 0.0039226 9.903e-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.

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

##   w.length   s.irrad
## 1      297 0.0001533
## 2      298 0.0003670
## 3      299 0.0007845
## 4      300 0.0012646
## 5      301 0.0026237
## 6      302 0.0039226
```

```
tail(trimmed.sun.data)

##     w.length s.irrad
## 499      795  0.4147
## 500      796  0.4081
## 501      797  0.4141
## 502      798  0.4236
## 503      799  0.4186
## 504      800  0.4069
```

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

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

##   w.length   s.irrad
## 1      297 0.0001533
## 2      298 0.0003670
## 3      299 0.0007845
## 4      300 0.0012646
## 5      301 0.0026237
## 6      302 0.0039226

tail(trimmed.both.sun.data)

##     w.length s.irrad
## 249      545  0.7272
## 250      546  0.6993
## 251      547  0.7119
## 252      548  0.6937
## 253      549  0.7021
## 254      550  0.7047
```

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

##   w.length   s.irrad
## 1      293        NA
## 2      294        NA
## 3      295        NA
## 4      296        NA
## 5      297 0.0001533
## 6      298 0.0003670
```

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

##   w.length   s.irrad
## 1      293        NA
```

```
## 2      294        NA
## 3      295        NA
## 4      296        NA
## 5      297 0.0001533
## 6      298 0.0003670

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

##     w.length s.irrad
## 503      795      NA
## 504      796      NA
## 505      797      NA
## 506      798      NA
## 507      799      NA
## 508      800      NA
```

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

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

##   w.length   s.irrad
## 1      293 0.0000000
## 2      294 0.0000000
## 3      295 0.0000000
## 4      296 0.0000000
## 5      297 0.0001533
## 6      298 0.0003670
```

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

13 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.