R for Photobiology

A handbook

Pedro J. Aphalo
and
Andreas Albert

Git: tag ‘none’, committed with hash 1a40e56 on 2014-11-27 00:32:57 +0200
by Pedro J. Aphalo
PDF created 27th November 2014

Contents

Preface

This is just a very early draft of a short book that will accompany the release of the suite of R packages for photobiology (r4photobiology).

Acknowledgements

We thank Titta Kotilainen, Stefano Catola, Paula Salonen, David Israel, Neha Rai, Tendry Randriamanana and . . . for very useful comments and suggestions. We specially thank Matt Robson for exercising the packages with huge amounts of spectral data and giving detailed feedback on problems, and in particular for describing needs and proposing new features.

List of abbreviations and symbols

For quantities and units used in photobiology we follow, as much as possible, the recommendations of the Commission Internationale de l'Éclairage as described by (Sliney 2007).

Symbol	**Definition**
α	(%).
Δe	water vapour pressure difference (Pa).
ϵ	emittance ($\mathrm{W\,m^{-2}}$).
λ	wavelength (nm).
θ	solar zenith angle (degrees).
ν	frequency (Hz or $\mathrm{s^{-1}}$).
ρ	(%).
σ	Stefan-Boltzmann constant.
τ	(%).
χ	water vapour content in the air ($\mathrm{g\,m^{-3}}$).
A	(absorbance units).
ANCOVA	analysis of covariance.
ANOVA	analysis of variance.
BSWF	.
c	speed of light in a vacuum.
CCD	charge coupled device, a type of light detector.
CDOM	coloured dissolved organic matter.
CFC	chlorofluorocarbons.
c.i.	confidence interval.
CIE	Commission Internationale de l'Éclairage; or erythemal action spectrum standardized by CIE.
CTC	closed-top chamber.
DAD	diode array detector, linear light detector based on photodiodes.
DBP	dibutylphthalate.
DC	direct current.
DIBP	diisobutylphthalate.
DNA(N)	UV action spectrum for 'naked' DNA.
DNA(P)	UV action spectrum for DNA in plants.
DOM	dissolved organic matter.
DU	Dobson units.
e	water vapour partial pressure (Pa).
E	(energy) irradiance ($\mathrm{W\,m^{-2}}$).
$E(\lambda)$	spectral (energy) irradiance ($\mathrm{W\,m^{-2}\,nm^{-1}}$).

E_0	fluence rate, also called scalar irradiance ($\mathrm{W\,m^{-2}}$).
ESR	early stage researcher.
FACE	free air carbon-dioxide enhancement.
FEL	a certain type of 1000 W incandescent lamp.
FLAV	UV action spectrum for accumulation of flavonoids.
FWHM	full-width half-maximum.
GAW	Global Atmosphere Watch.
GEN	generalized plant action spectrum, also abreviated as GPAS (Caldwell 1971).
GEN(G)	mathematical formulation of GEN by (Green et al. 1974) .
GEN(T)	mathematical formulation of GEN by (Thimijan et al. 1978).
h	Planck's constant.
h'	Planck's constant per mole of photons.
H	exposure, frequently called dose by biologists ($\mathrm{kJ\,m^{-2}\,d^{-1}}$).
H^{BE}	biologically effective (energy) exposure ($\mathrm{kJ\,m^{-2}\,d^{-1}}$).
$H_{\mathrm{p}}^{\mathrm{BE}}$	biologically effective photon exposure ($\mathrm{mol\,m^{-2}\,d^{-1}}$).
HPS	high pressure sodium, a type of discharge lamp.
HSD	honestly signifcant difference.
k_{B}	Boltzmann constant.
L	radiance ($\mathrm{W\,sr^{-1}\,m^{-2}}$).
LAI	leaf area index, the ratio of projected leaf area to the ground area.
LED	light emitting diode.
LME	linear mixed effects (type of statistical model).
LSD	least significant difference.
n	number of replicates (number of experimental units per treatment).
N	total number of experimental units in an experiment.
N_{A}	Avogadro constant (also called Avogadro's number).
NIST	National Institute of Standards and Technology (U.S.A.).
NLME	non-linear mixed effects (statistical model).
OTC	open-top chamber.
PAR	, 400–700 nm.
	measured as energy or photon irradiance.
PC	polycarbonate, a plastic.
PG	UV action spectrum for plant growth.
PHIN	UV action spectrum for photoinhibition of isolated chloroplasts.
PID	(control algorithm).
PMMA	polymethylmethacrylate.
PPFD	, another name for
	PAR photon irradiance (Q_{PAR}).
PTFE	polytetrafluoroethylene.
PVC	polyvinylchloride.
q	energy in one photon ('energy of light').
q'	energy in one mole of photons.
Q	photon irradiance ($\mathrm{mol\,m^{-2}\,s^{-1}}$ or $\mathrm{\mu mol\,m^{-2}\,s^{-1}}$).
$Q(\lambda)$	spectral photon irradiance ($\mathrm{mol\,m^{-2}\,s^{-1}\,nm^{-1}}$ or $\mathrm{\mu mol\,m^{-2}\,s^{-1}\,nm^{-1}}$).
r_0	distance from sun to earth.
RAF	(nondimensional).
RH	relative humidity (%).
s	energy effectiveness (relative units).

$s(\lambda)$	spectral energy effectiveness (relative units).
s^{p}	quantum effectiveness (relative units).
$s^{\mathrm{p}}(\lambda)$	spectral quantum effectiveness (relative units).
s.d.	standard deviation.
SDK	software development kit.
s.e.	standard error of the mean.
SR	spectroradiometer.
t	time.
T	temperature.
TUV	tropospheric UV.
U	electric potential difference or voltage (e.g. sensor output in V).
UV	ultraviolet radiation (λ = 100–400 nm).
UV-A	ultraviolet-A radiation (λ = 315–400 nm).
UV-B	ultraviolet-B radiation (λ = 280–315 nm).
UV-C	ultraviolet-C radiation (λ = 100–280 nm).
$\mathrm{UV}^{\mathrm{BE}}$	biologically effective UV radiation.
UTC	coordinated universal time, replaces GMT in technical use.
VIS	radiation visible to the human eye ($\approx$ 400–700 nm).
WMO	World Meteorological Organization.
VPD	water vapour pressure deficit (Pa).
WOUDC	World Ozone and Ultraviolet Radiation Data Centre.

Part I

Preliminaries

Part II

Cookbook of calculations

CHAPTER 1

Radiation physics

Abstract

In this chapter we explain how to code some optics and physics computations in R.

1.1 Packages used in this chapter

For executing the examples listed in this chapter you need first to load the following packages from the library:

```
library(ggplot2)

## Loading required package:  methods

library(photobiologygg)

## Loading required package:  photobiology
## Loading required package:  lubridate
## Loading required package:  data.table
##
## Attaching package:  'data.table'
##
## The following objects are masked from 'package:lubridate':
##
##     hour, mday, month, quarter, wday, week,
##     yday, year
##
## Loading required package:  photobiologyWavebands
## Loading required package:  proto
## Loading required package:  splus2R
## Loading required package:  plyr
##
## Attaching package:  'plyr'
##
```

```
## The following object is masked from 'package:lubridate':
##
##     here

library(photobiology)
library(photobiologyFilters)
```

1.2 Introduction

1.3 Task: black body emission

The emitted spectral radiance (L_s) is described by Planck's law of black body radiation at temperature T, measured in degrees Kelvin (K):

$$L_s(\lambda, T) = \frac{2hc^2}{\lambda^5} \cdot \frac{1}{e^{(hc/k_B T\lambda)} - 1} \tag{1.1}$$

with Boltzmann's constant $k_B = 1.381 \times 10^{-23}$ J K^{-1}, Planck's constant $h = 6.626 \times 10^{-34}$ J s and speed of light in vacuum $c = 2.998 \times 10^8$ m s^{-1}.

We can easily define an R function based on the equation above, which returns W sr^{-1} m^{-3}:

```
h <- 6.626e-34 # J s-1
c <- 2.998e8 # m s-1
kB <- 1.381e-23 # J K-1
black_body_spectrum <- function(w.length, Tabs) {
  w.length <- w.length * 1e-9 # nm -> m
  ((2 * h * c^2) / w.length^5) *
    1 / (exp((h * c / (kB * Tabs * w.length))) - 1)
}
```

We can use the function for calculating black body emission spectra for different temperatures:

```
black_body_spectrum(500, 5000)

## [1] 1.212443e+13
```

The function is vectorized:

```
black_body_spectrum(c(300,400,500), 5000)

## [1] 3.354907e+12 8.759028e+12 1.212443e+13
```

```
black_body_spectrum(500, c(4500,5000))

## [1] 6.387979e+12 1.212443e+13
```

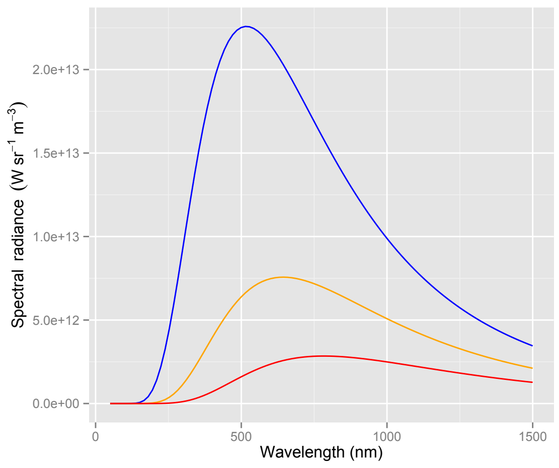
We aware that if two vectors are supplied, then the elements in each one are matched and recycled[1]:

[1] Exercise: calculate each of the four values individually to work out how the two vectors are being used.

```
black_body_spectrum(c(500, 500, 600, 600), c(4500,5000)) # tricky!

## [1] 6.387979e+12 1.212443e+13 7.474587e+12
## [4] 1.277769e+13
```

We can use the function defined above for plotting black body emission spectra for different temperatures. We use `ggplot2` and directly plot a function using `stat_function`, using `args` to pass the additional argument giving the absolute temperature to be used. We plot three lines using three different temperatures (5600 K, 4500 K, and 3700 K):

```
ggplot(data=data.frame(x=c(50,1500)), aes(x)) +
  stat_function(fun=black_body_spectrum,
                args = list(Tabs=5600),
                colour="blue") +
  stat_function(fun=black_body_spectrum,
                args = list(Tabs=4500),
                colour="orange") +
  stat_function(fun=black_body_spectrum,
                args = list(Tabs=3700),
                colour="red") +
  labs(y=expression(Spectral~~radiance~~(W~sr^-1~m^-3)),
       x="Wavelength (nm)")
```



Wien's displacement law, gives the peak wavelength of the radiation emitted by a black body as a function of its absolute temperature.

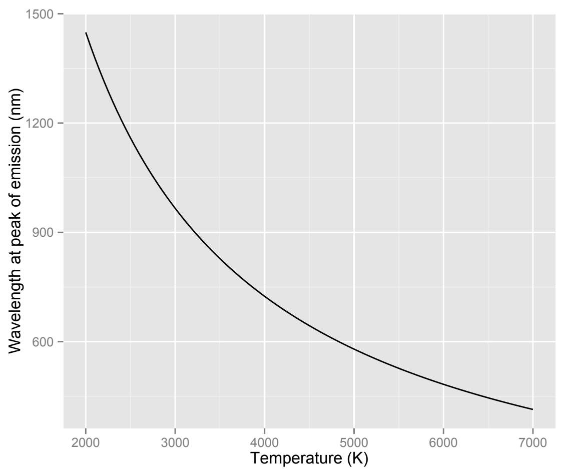
$$\lambda_{max} \cdot T = 2.898 \times 10^6 \,\mathrm{nm\,K} \tag{1.2}$$

A function implementing this equation takes just a few lines of code:

```
k.wein <- 2.8977721e6 # nm K
black_body_peak_wl <- function(Tabs) {
  k.wein / Tabs
}
```

It can be used to plot the temperature dependence of the location of the wavelength at which radiance is at its maximum:

```
ggplot(data=data.frame(Tabs=c(2000,7000)), aes(x=Tabs)) +
  stat_function(fun=black_body_peak_wl) +
  labs(x="Temperature (K)",
       y="Wavelength at peak of emission (nm)")
```



```
try(detach(package:photobiologyFilters))
try(detach(package:photobiologygg))
try(detach(package:photobiology))
try(detach(package:ggplot2))
```

CHAPTER 2

Astronomy

Abstract

In this chapter we explain how to code some astronomical computations in R.

2.1 Packages used in this chapter

For executing the examples listed in this chapter you need first to load the following packages from the library:

```
library(photobiology)
library(lubridate)
library(ggplot2)
library(ggmap)
```

2.2 Introduction

This chapter deals with calculations that require times and/or dates as arguments. One could use R's built-in functions for POSIXct but package `lubridate` makes working with dates and times, much easier. Package `lubridate` defines functions for decoding dates represented as character strings, and for manipulating dates and doing calcualtions on dates. Each one of the different functions shown in the code chunk below can decode dates in different formats as long as the year, month and date order in the string agrees with the name of the function:

```
ymd("20140320")

## [1] "2014-03-20 UTC"
```

```
ymd("2014-03-20")

## [1] "2014-03-20 UTC"

ymd("14-03-20")

## [1] "2014-03-20 UTC"

ymd("2014-3-20")

## [1] "2014-03-20 UTC"

ymd("2014/3/20")

## [1] "2014-03-20 UTC"

dmy("20032014")

## [1] "2014-03-20 UTC"

mdy("03202014")

## [1] "2014-03-20 UTC"
```

For astronomical calculations we need as argument the geographical coordinates. It is, of course, possible to enter latitude and longitude values recorded with a GPS instrument or manually obtained from a map. However, when the location is searchable through Google Maps, it is also possible to obtain the coordinates by means of a query from within R using packages `RgoogleMaps`, or package `ggmap`, as done here. When inputing coordinate values manually, they should in degrees as numeric values (in other words the fractional part is given as part of floating point numberin degrees, and not as separate integers representing minutes and seconds of degree).

```
geocode("Helsinki")

##        lon      lat
## 1 24.94102 60.17332

geocode("Viikinkaari 1, 00790 Helsinki, Finland")

##        lon     lat
## 1 25.01673 60.2253
```

2.3 Task: calculating the length of the photoperiod

In function `day_night` from our `photobiology` package we use function `sun_angles`, which is an edited version of function `sunAngle` from package `ode`, to calculate the altitude or elevation of the sun. We first find local solar noon by finding the maximal solar elevation, and then search for sunrise in the first half of the day and for sunset in the second half, defined based on the local solar noon. Sunset and sunrise are by default based on a solar elevation angle equal to zero. The argument `twilight` can be used to set the angle according to different conventions.

In the examples we use `geocode` to get the latitude and longitude of cities. `geocode` accepts any valid Google Maps search terms, including street addresses, and postal codes within cities. `day_night` returns a list containing the times at sunrise, sunset and noon, and day- and night lengths. This first example is for Buenos Aires on two different dates, by use of the optional argument `tz` we request the results to be expressed in local time for Buenos Aires.

```
geo_code_BA <- geocode("Buenos Aires")
geo_code_BA

##          lon       lat
## 1 -58.38159 -34.60372

day_night(ymd("2013-12-21"),
          lon = geo_code_BA[["lon"]],
          lat = geo_code_BA[["lat"]],
          tz="America/Argentina/Buenos_Aires")

## $day
## [1] "2013-12-21 UTC"
##
## $sunrise
## [1] "2013-12-21 05:42:00 ART"
##
## $noon
## [1] "2013-12-21 12:51:46 ART"
##
## $sunset
## [1] "2013-12-21 20:01:32 ART"
##
## $daylength
## Time difference of 14.32535 hours
##
## $nightlength
## Time difference of 9.674649 hours

day_night(ymd("2013-06-21"),
          lon = geo_code_BA[["lon"]],
          lat = geo_code_BA[["lat"]],
          tz="America/Argentina/Buenos_Aires")

## $day
## [1] "2013-06-21 UTC"
##
## $sunrise
## [1] "2013-06-21 08:04:57 ART"
##
## $noon
## [1] "2013-06-21 12:55:32 ART"
##
## $sunset
## [1] "2013-06-21 17:45:49 ART"
##
## $daylength
## Time difference of 9.681105 hours
```

```
##
## $nightlength
## Time difference of 14.3189 hours
```

We here repeat the same calculations for Munich on the same days —note that the output for December is in "EET" time coordinates, and for June it is in "EEST", i.e. in 'winter-' and 'summer time' coordinates.

```
geo_code_Mu <- geocode("Munich")
geo_code_Mu

##         lon      lat
## 1 11.58198 48.13513

day_night(ymd("2013-12-21"),
          lon = geo_code_Mu[["lon"]],
          lat = geo_code_Mu[["lat"]],
          tz="Europe/Berlin")

## $day
## [1] "2013-12-21 UTC"
##
## $sunrise
## [1] "2013-12-21 08:07:27 CET"
##
## $noon
## [1] "2013-12-21 12:11:49 CET"
##
## $sunset
## [1] "2013-12-21 16:16:11 CET"
##
## $daylength
## Time difference of 8.145512 hours
##
## $nightlength
## Time difference of 15.85449 hours

day_night(ymd("2013-06-21"),
          lon = geo_code_Mu[["lon"]],
          lat = geo_code_Mu[["lat"]],
          tz="Europe/Berlin")

## $day
## [1] "2013-06-21 UTC"
##
## $sunrise
## [1] "2013-06-21 05:19:41 CEST"
##
## $noon
## [1] "2013-06-21 13:15:29 CEST"
##
## $sunset
## [1] "2013-06-21 21:11:16 CEST"
##
## $daylength
## Time difference of 15.85966 hours
##
## $nightlength
## Time difference of 8.140341 hours
```

As a final example, we calculate day length based on different definitions of twilight for Helsinki, at the equinox:

```
geo_code_He <- geocode("Helsinki")
geo_code_He

##        lon      lat
## 1 24.94102 60.17332

day_night(ymd("2013-09-21"),
          lon = geo_code_He[["lon"]], lat = geo_code_He[["lat"]])

## $day
## [1] "2013-09-21 UTC"
##
## $sunrise
## [1] "2013-09-21 07:08:45 EEST"
##
## $noon
## [1] "2013-09-21 13:12:49 EEST"
##
## $sunset
## [1] "2013-09-21 19:16:23 EEST"
##
## $daylength
## Time difference of 12.12728 hours
##
## $nightlength
## Time difference of 11.87272 hours

day_night(ymd("2013-09-21"),
          lon = geo_code_He[["lon"]], lat = geo_code_He[["lat"]],
          twilight="civil")

## $day
## [1] "2013-09-21 UTC"
##
## $sunrise
## [1] "2013-09-21 07:57:16 EEST"
##
## $noon
## [1] "2013-09-21 13:12:49 EEST"
##
## $sunset
## [1] "2013-09-21 18:28:02 EEST"
##
## $daylength
## Time difference of 10.51275 hours
##
## $nightlength
## Time difference of 13.48725 hours

day_night(ymd("2013-09-21"),
          lon = geo_code_He[["lon"]], lat = geo_code_He[["lat"]],
          twilight="nautical")

## $day
## [1] "2013-09-21 UTC"
##
```

```
## $sunrise
## [1] "2013-09-21 08:47:20 EEST"
##
## $noon
## [1] "2013-09-21 13:12:49 EEST"
##
## $sunset
## [1] "2013-09-21 17:38:05 EEST"
##
## $daylength
## Time difference of 8.845828 hours
##
## $nightlength
## Time difference of 15.15417 hours

day_night(ymd("2013-09-21"),
          lon = geo_code_He[["lon"]], lat = geo_code_He[["lat"]],
          twilight="astronomical")

## $day
## [1] "2013-09-21 UTC"
##
## $sunrise
## [1] "2013-09-21 09:41:31 EEST"
##
## $noon
## [1] "2013-09-21 13:12:49 EEST"
##
## $sunset
## [1] "2013-09-21 16:44:00 EEST"
##
## $daylength
## Time difference of 7.041373 hours
##
## $nightlength
## Time difference of 16.95863 hours
```

2.4 Task: calculating the position of the sun

`sun_angles` not only returns solar elevation, but all the angles defining the position of the sun. The time argument to `sun_angles` is internally converted to UTC (universal time coordinates, which is equal to GMT) time zone, so time defined for any time zone is valid input. The time zone used for the output is by default that currently in use in the computer on which R is running, but we can easily specify the time coordinates used for the output with parameter `tz`, using any string accepted by package `lubridate`.

```
geo_code_Jo <- geocode("Joensuu")
geo_code_Jo

##        lon      lat
## 1 29.76353 62.60109

my_time <- ymd_hms("2014-05-29 18:00:00", tz="EET")
sun_angles(my_time,
         lon = geo_code_Jo[["lon"]], lat = geo_code_Jo[["lat"]])
```

```
## $time
## [1] "2014-05-29 18:00:00 EEST"
##
## $azimuth
## [1] 267.585
##
## $elevation
## [1] 25.81887
##
## $diameter
## [1] 0.5260482
##
## $distance
## [1] 1.013595
```

We can calculate the current position of the sun, in this case giving the position of the sun in the sky of Joensuu when this .PDF file was generated.

```
sun_angles(now(),
         lon = geo_code_Jo[["lon"]], lat = geo_code_Jo[["lat"]])

## $time
## [1] "2014-11-27 20:14:11 EET"
##
## $azimuth
## [1] 293.3916
##
## $elevation
## [1] -35.14582
##
## $diameter
## [1] 0.5403949
##
## $distance
## [1] 0.9866859
```

2.5 Task: plotting sun elevation through a day

Function `sun_angles` described above is vectorized, so it is very easy to calculate the position of the sun throughout a day at a given location on Earth. The example here uses sun only elevation, plotted for Helsinki through the course of 23 June 2014. We first a vector of times, using `seq` which can not only be used with numbers, but also with dates. Note that `by` is specified as a string.

```
opts_chunk$set(opts_fig_wide)
```

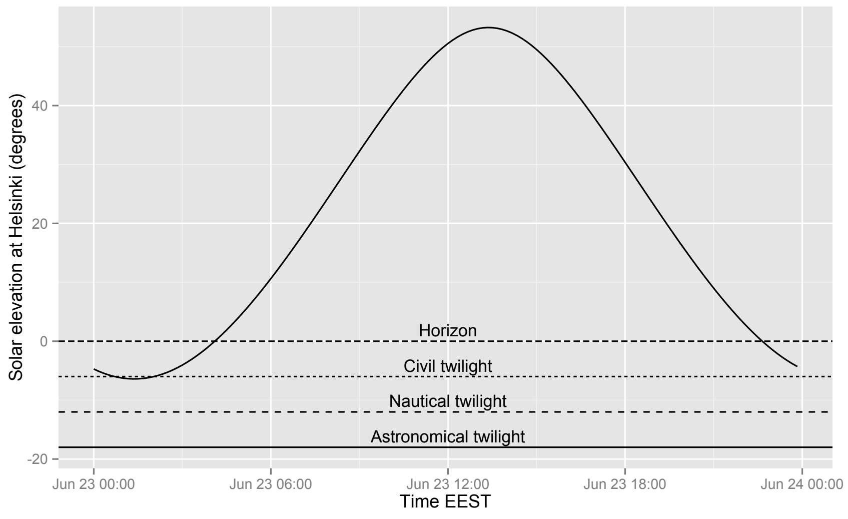
```
hours <- seq(from=ymd("2014-06-23", tz="EET"),
             by="10 min",
             length=24 * 6)
elevations <- sun_angles(hours,
            lon = geo_code_He[["lon"]],
            lat = geo_code_He[["lat"]])$elevation
sun_elev_hel <- data.frame(time_eet = hours,
                              elevation = elevations,
                              location = "Helsinki",
                              lon = geo_code_He[["lon"]],
                             lat = geo_code_He[["lat"]])
```

We also create a small data frame with data for plotting and labeling the different twilight conventions.

```
twilight <-
  data.frame(angle = c(0, -6, -12, -18),
             label = c("Horizon", "Civil twilight",
                       "Nautical twilight",
                       "Astronomical twilight"),
             time = rep(ymd_hms("2014-06-23 12:00:00",
                                tz="EET"),
                        4) )
```

We draw a plot using the data frames created above.

```
ggplot(sun_elev_hel,
       aes(x = time_eet, y = elevation)) +
  geom_line() +
  geom_hline(data=twilight,
             aes(yintercept = angle, linetype=factor(label))) +
  annotate(geom="text",
           x=twilight$time, y=twilight$angle,
           label=twilight$label, vjust=-0.4, size=4) +
  labs(y = "Solar elevation at Helsinki (degrees)",
       x = "Time EEST")
```



2.6 Task: plotting day length through the year

For this we first need to generate a sequence of dates. We use `seq` as in the previous section, but instead of supplying a length as argument we supply an ending time. Instead of giving `by` in minutes as above, we now use days:

```
days <- seq(from=ymd("2014-01-01"), to=ymd("2014-12-31"),
            by="3 day")
```

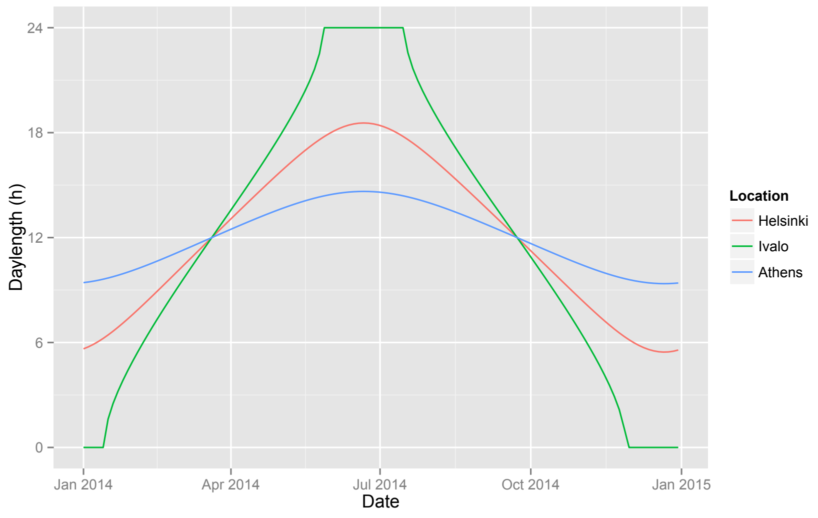
To calculate the length of each day, we need to use an explicit loop as function `day_night` is not vectorized. We repeat the calculations for three locations at different latitudes, then row bind the data frames into a single data frame. Each individual data frame contains information to identify the sites:

```
len_days <- length(days)
photoperiods <- numeric(len_days)
geo_code_He <- geocode("Helsinki")
for (i in 1:len_days) {
  day_night.ls <- day_night(days[i],
                            lon = geo_code_He[["lon"]],
                            lat = geo_code_He[["lat"]],
                            tz="EET")
  photoperiods[i] <-
    as.numeric(day_night.ls[["daylength"]],
               units="hours")
}
daylengths_hel <-
  data.frame(day = days,
             daylength = photoperiods,
             location="Helsinki",
             lon = geo_code_He[["lon"]],
             lat = geo_code_He[["lat"]])
geo_code_Iv <- geocode("Ivalo")
for (i in 1:len_days) {
  day_night.ls <- day_night(days[i],
                            lon = geo_code_Iv[["lon"]],
                            lat = geo_code_Iv[["lat"]],
                            tz="EET")
  photoperiods[i] <-
    as.numeric(day_night.ls[["daylength"]],
               units="hours")
}
daylengths_ivalo <-
  data.frame(day = days,
             daylength = photoperiods,
             location="Ivalo",
             lon = geo_code_Iv[["lon"]],
             lat = geo_code_Iv[["lat"]])
geo_code_At <- geocode("Athens, Greece")
for (i in 1:len_days) {
  day_night.ls <- day_night(days[i],
                            lon = geo_code_At[["lon"]],
                            lat = geo_code_At[["lat"]],
                            tz="EET")
  photoperiods[i] <-
    as.numeric(day_night.ls[["daylength"]],
               units="hours")
}
daylengths_athens <-
  data.frame(day = days,
             daylength = photoperiods,
             location="Athens",
             lon = geo_code_At[["lon"]],
             lat = geo_code_At[["lat"]])
```

```
daylengths <- rbind(daylengths_hel,
                    daylengths_ivalo,
                    daylengths_athens)
```

Once we have the data available, plotting is simple:

```
ggplot(daylengths,
       aes(x = day, y = daylength, colour=factor(location))) +
  geom_line() +
  scale_y_continuous(breaks=c(0,6,12,18,24), limits=c(0,24)) +
  labs(x = "Date", y = "Daylength (h)", colour="Location")
```



```
try(detach(package:photobiology))
try(detach(package:lubridate))
try(detach(package:ggmap))
try(detach(package:ggplot2))
```

CHAPTER 3

Basic operations on spectra

Abstract

In this chapter we describe the objects used to store data and functions and operators for basic operations. We also give some examples of operating on these objects and their components using normal R functions and operators.

3.1 Packages used in this chapter

For executing the examples listed in this chapter you need first to load the following packages from the library:

```
library(photobiology)
library(photobiologyWavebands)
library(photobiologyFilters)
library(photobiologyLEDs)
```

3.2 Introduction

The suite uses object-oriented programming for its higher level ‘user-friendly’ syntax. Objects are implemented using “S3” classes. The two main distinct kinds of objects are different types of spectra, and wavebands. Spectral objects contain, as their name implies, spectral data. Wavebands contain the information needed to calculate irradiance, non-weighted or weighted (effective), and a name and a label to be used in output printing. Functions and operators are defined for operations on these objects, alone and in combination. We will first describe spectra, and then wavebands, in each case describing operators and functions. Towards the end of the chapter we describe

3.3 Spectra

3.3.1 How are spectra stored?

For spectra the classes are a specialization of `data.table` which are in turn a specialization of `data.frame`. This means that they are compatible with functions that operate on these classes.

The suite defines a `generic.spct` class, from which two specialized classes, '`filter.spct`, `reflector.spct`, `source.spct`, `response.spct` and `chroma.spct` are derived. Having this class structure allows us to create special methods and operators, which use the same names than the generic ones defined by R itself, but take into account the special properties of spectra. Each spectrum object can hold only one spectrum.

Objects of class `generic.spct` one one mandatory component `w.length`, containing wavelength values in nm.

Objects of class `source.spct` have two mandatory components `w.length`, and `s.e.irrad`, and an optional one, `s.q.irrad`. They are expected to contain data expressed always in the same units: nm, for `w.length`, $\mathrm{W\,m^{-2}\,nm^{-1}}$ for `s.e.irrad`, and $\mathrm{mol\,m^{-2}\,s^{-1}\,nm^{-1}}$ for `s.q.irrad`. Objects can have a "comment" attribute with a textual description. Additional columns are ignored, but not deleted, unless the operation applied could invalidate them.

Objects of class `filter.spct` have two mandatory components `w.length`, and `Tfr` and two optional components,`Tpc` and A. They are expected to contain data expressed always in the same units: nm, for `w.length`, a fraction of one for `Tfr`, and % for `Tpc`. Absorbance A values are expected to be expressed based on $\log_{10}$. Objects have a "comment" attribute with a textual description.

Objects of class `reflector.spct` have two mandatory components `w.length`, and `Rfr` and one optional components,`Rpc`. They are expected to contain data expressed always in the same units: nm, for `w.length`, a fraction of one for `Rfr`, and % for `Rpc`. Objects have a "comment" attribute with a textual description.

Objects of class `chroma.spct` have four mandatory components `w.length`, and `x, y, z` giving the chromaticity coordinates for trichromic vision.

Objects of class `response.spct` have two mandatory components `w.length`, and `response` giving the spectral response.

3.3.2 How can the user create spectra from his own data

If the data is already stored in a data frame or data table, or even a list, and if the components have one of the recognized "standard" names, specific `setGenericSpct`, `setSourceSpct`, `setFilterSpct`, `setReflectorSpct` commands can be used to change the class attribute and check that the object is valid. These functions have the same semantics as `setDT` and `setDF` from package `data.table`, they modify their argument directly—the argument is passed by *reference* instead of by *copy* as is usual in R. As `sun.data` is part of the package, we need to make a copy before modifying it, with one's own data frames or data tables this step is not need.

We can create a new object from two vectors,

```
source.spct(sun.data$w.length, s.e.irrad = sun.data$s.e.irrad)
```

or make a copy of a data frame or a data table and convert it into a source spectrum,

```
as.source.spct(sun.data)
```

or convert an existing data frame or data table into a source spectrum[1].

```
my_sun.spct <- sun.data
setSourceSpct(my_sun.spct)
```

We can query the class of an object.

```
class(my_sun.spct)

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

is.source.spct(my_sun.spct)

## [1] TRUE
```

Similar functions are available for filter, reflector and response spectral objects.

Table 3.1 lists the different 'names' understood by the constructor functions which take a data frame as argument, and the required and optional components of the different spectral object classes.

3.3.3 What operators are available for operations between spectra?

All operations with spectral objects affect only the required components listed in Table 3.1, all optional components are deleted, while unrecognized components are left alone. There will be seldom need to add numerical components, and the user should take into account that the paradigm of the suite is that each spectrum is stored as a separate object. However, it is allowed, and possibly useful to have factors as components with levels identifying different bands, or color vectors with RGB values. Ancillary information information useful for presentation and plotting might sometimes be useful.

Several operators are defined for spectral objects. Using operators is an easy and familiar way of doing calculations, but operators are rather inflexible (they can take at most two arguments, the operands) and performance is slower than with functions with additional parameters that allow optimizing the algorithm. The operators are defined so that an operation between two `filter.spct` objects yields another `filter.spct` object, an operation

[1] In this case we need to copy sun.data because this data frame is protected as a member of the package. This is rarely needed with user's data.

Table 3.1: Names of spectral object components, and the additional names recognized during automatic spectral object creation, and the units of expression.

Class	required	optional	recognized	units
generic.spct	w.length	—	wl, wavelength	nm
source.spct	w.length	—	wl, wavelength	nm
	s.e.irrad	—	irradiance	$\mathrm{W\,m^{-2}\,nm^{-1}}$
	—	s.q.irrad	—	$\mathrm{mol\,m^{-2}\,s^{-1}\,nm^{-1}}$
filter.spct	w.length	—	wl, wavelength	nm
	Tfr	—	—	$x/1$
	—	Tpc	transmittance	%
	—	A	absorbance	a.u. $\log_{10}$-based
reflector.spct	w.length	—	wl, wavelength	
	Rfr	—	—	$x/1$
	—	Rpc	reflectance	%
response.spct	w.length	—	wl, wavelength	nm
	response	—	response	arbitrary u.
chroma.spct	w.length	—	wl, wavelength	nm
	x, y, z	—	X, Y, Z	relative u.

between two `reflector.spct` yields a `reflector.spct` object, and operations between a `filter.spct` object and a `source.spct`, between a `reflector.spct` and a `source.spct`, or between two `source.spct` objects yield `source.spct` objects. The object returned contains data only for the overlapping region of wavelengths. The objects do NOT need to have values at the same wavelengths, as interpolation is handled transparently. All four basic maths operations are supported with any combination of spectra, and the user is responsible for deciding which calculations make sense and which not. Operations can be concatenated and combined. The unary negation operator is also implemented.

For example we can convolute the emission spectrum of a light source and the transmittance spectrum of a filter.

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

##       w.length    s.e.irrad
##   1:       293 7.828995e-09
##   2:       294 1.842720e-08
##   3:       295 6.528525e-08
##   4:       296 2.034036e-07
##   5:       297 4.600472e-07
##  ---
## 504:       796 3.705200e-01
## 505:       797 3.764354e-01
## 506:       798 3.855015e-01
## 507:       799 3.809123e-01
## 508:       800 3.706909e-01
```

3.3.4 What operators are available for operations between spectra and numeric vectors?

The same four basic math operators plus power ('^') are defined for the case when the first term or factor is a spectrum and the second one a numeric vector, possibly of length one. Recycling rules apply. These operations do not alter `w.length`, just the other *required* components such as spectral irradiance and transmittance. The optional components are deleted as they can be recalculated if needed. Unrecognized 'user' components are left unchanged.

For example we can divide an spectrum by a numeric value (a vector of length 1, which gets recycle). The value returned is a spectral object of the same type as the firt argument.

```
sun.spct / 2

##      w.length    s.e.irrad
##   1:       293 1.304833e-06
##   2:       294 3.071200e-06
##   3:       295 1.088087e-05
##   4:       296 3.390059e-05
##   5:       297 7.667453e-05
##  ---
## 504:       796 2.040308e-01
## 505:       797 2.070602e-01
## 506:       798 2.118140e-01
## 507:       799 2.092925e-01
## 508:       800 2.034528e-01
```

3.3.5 What unary math functions are available for spectra?

Logarithms (`log, log10`), square root (`sqrt`) and exponentiation (`exp`) are defined for spectra. These functions are not applied on `w.length`, but instead to the other mandatory component `s.e.irrad`, `Rfr` or `Tfr`. Any optional numeric components are discarded. (Other user-supplied components should remain unchanged, but this needs further checking!)

```
log10(sun.spct)

##      w.length  s.e.irrad
##   1:       293 -5.5834152
##   2:       294 -5.2116619
##   3:       295 -4.6623062
##   4:       296 -4.1687627
##   5:       297 -3.8143189
##  ---
## 504:       796 -0.3892742
## 505:       797 -0.3828734
## 506:       798 -0.3730153
## 507:       799 -0.3782164
## 508:       800 -0.3905064
```

3.3.6 What 'summary' functions are available for spectra?

The R functions `summary`, `print` work using their `data.table` definitions, however, there are special versions of `range`, `min`, `max` that when applied to spectra return values corresponding to wavelengths, two other generic functions defined in the suite give additional summaries of spectra `spread`, `midpoint`.

3.3.7 Examples

Package `phobiologyFilters` makes available many different filter spectra, from which we choose Schott filter GG400. Package `photobiology` makes available one example solar spectrum. Using these data we will simulate the filtered solar spectrum.

```
filtered_sun.spct <- sun.spct * gg400.spct
filtered_sun.spct

##      w.length    s.e.irrad
##   1:      293 2.609665e-11
##   2:      294 6.142401e-11
##   3:      295 2.176175e-10
##   4:      296 6.780119e-10
##   5:      297 1.533491e-09
##  ---
## 504:      796 3.958198e-01
## 505:      797 4.016967e-01
## 506:      798 4.109192e-01
## 507:      799 4.060274e-01
## 508:      800 3.946984e-01
```

The GG440 data is for internal transmittance, consequently the results above would be close to the truth only for filters treated with an anti-reflexion multicoating. Let's assume a filter with 9% reflectance across all wavelengths (a coarse approximation for uncoated glass):

```
filtered_uncoated_sun.spct <- sun.spct * gg400.spct * (100 - 9) / 100
filtered_uncoated_sun.spct

##      w.length    s.e.irrad
##   1:      293 2.374795e-11
##   2:      294 5.589585e-11
##   3:      295 1.980319e-10
##   4:      296 6.169908e-10
##   5:      297 1.395476e-09
##  ---
## 504:      796 3.601960e-01
## 505:      797 3.655440e-01
## 506:      798 3.739365e-01
## 507:      799 3.694849e-01
## 508:      800 3.591755e-01
```

Calculations related to filters will be explained in detail in chapter **??**. This is just an example of how the operators work, even when, as in this example, the wavelength values do not coincide bertween the two spectra.

3.3.8 Task: uniform scaling of a spectrum

As noted above operators are available for `generic.scpt`, `source.spct`, `filter.spct` and `reflector.spct` objects, and 'recycling' takes places when needed:

```
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

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
```

All four basic binary operators (`+, -, *, /`) can be used in the same way, but when operating between a spectrum an a numeric value the spectrum should be the first term or factor. If an operation on a "source.spct" would yield different values for data on energy and photon basis, only the value based on energy data is returned in `s.e.irrad` and `s.q.irrad` is set to NA.

3.3.9 Task: simple operations between two spectra

```
filtered_sun.spct <- ug1.spct * sun.spct
filtered_sun.spct

##      w.length    s.e.irrad
##   1:      293 2.286067e-07
##   2:      294 6.191540e-07
##   3:      295 2.480839e-06
##   4:      296 8.624311e-06
##   5:      297 2.153021e-05
##  ---
## 504:      796 1.069122e-01
## 505:      797 1.072572e-01
## 506:      798 1.084488e-01
## 507:      799 1.059020e-01
## 508:      800 1.017264e-01
```

All four basic binary operators (`+`, `-`, `*`, `/`) can be used in the same way, and they can be combined into equations.

3.3.10 Task: arithmetic operations within one spectrum

If data for two spectra are available for the same wavelength values, then we can simply use the built in R mat operators on vectors (e.g. when only individual vectors are available, or a data frame). These operators are vectorized, which means that an addition between two vectors adds the elements at each position. A non-nonsensical example follows using R syntax on a data frame, returning a vector.

Using data frame syntax on a data frame, data table or spectral object, returning a vector:

```
# not run
with(sun.spct, s.e.irrad^2 / w.length)
```

Using data table syntax on a data table or spectral object, returning a vector:

```
# not run
sun.spct[ , s.e.irrad^2 / w.length]
```

Using data table syntax, adding the result to the `data.table` object, or a `____.spct` object:

```
# run
my_sun.spct <- copy(sun.spct)
my_sun.spct[ , result := s.e.irrad^2 / w.length]

##      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
##            result
##   1: 2.324352e-14
##   2: 1.283302e-13
##   3: 1.605335e-12
##   4: 1.553041e-11
##   5: 7.917823e-11
##  ---
## 504: 2.091888e-04
## 505: 2.151765e-04
## 506: 2.248882e-04
## 507: 2.192908e-04
## 508: 2.069651e-04
```

3.3.11 Task: other operations between two spectra

If data for two spectra are available for the same wavelength values, then we can simply use the built in R math operators. These operators are vectorized, which means that an addition between two vectors adds the elements at the same index position in the two vectors with data, in this case for two different spectra. So, they do not differ from the examples in the previous section for normal R syntax. Data table syntax is no longer so convenient in this case.

In contrast to the previous case, operations using built-in R operators cannot be done if the wavelengths in two spectral data sets are not matched. In this situation is when functions and operators defined in package `photobiology` come to the rescue by transparently making the two operand spectra compatible by interpolation. The result they return includes all the individual wavelength values (the set union of the wavelengths from the two spectra in the region where they overlap). The functions are `sum_spectra`, `subt_spectra`, `prod_spectra`, `div_spectra`, and `oper_spectra`. Here is a very simple hypothetical example:

```
# not run
out1.dt <- sum_spectra(spc1$w.length, spc2$w.length,
                       spc1$s.e.irrad, spc2$s.e.irrad)
```

We can achieve the same result, with simpler syntax, using spectral objects and the corresponding operators. The actual computations are done in both cases by the same code, but the example below adds some “syntactic sugar” to make the script code more readable.

```
out2.spct <- sun.spct + sun.spct
out3.spct <- e2q(sun.spct + sun.spct)
out3.spct

##      w.length    s.e.irrad    s.q.irrad
##   1:      293 5.219330e-06 1.278346e-11
##   2:      294 1.228480e-05 3.019128e-11
##   3:      295 4.352350e-05 1.073277e-10
##   4:      296 1.356024e-04 3.355251e-10
##   5:      297 3.066981e-04 7.614363e-10
##  ---
## 504:      796 8.161233e-01 5.430438e-06
## 505:      797 8.282407e-01 5.517990e-06
## 506:      798 8.472561e-01 5.651759e-06
## 507:      799 8.371699e-01 5.591475e-06
## 508:      800 8.138111e-01 5.442264e-06
```

In both cases only spectral energy irradiance is calculated during the summing operation, while in the second example, it is simple to convert the returned spectral energy irradiance values into spectral photon irradiance. `out1.data` is a "data.table", while the second will be a spectrum of a class dependent on the classes of spc1 and spc2. Obviously, the second calculation will be slower, but in most cases unnoticeable so[2].

The function oper_spectra takes the operator to use as an argument, and this abstraction both simplifies the package code, and also makes it easy for users to add other operators if needed:

```
out.data <- oper_spectra(spc1$w.length, spc2$w.length,
                         spc1$s.e.irrad, spc2$s.e.irrad,
                         bin.oper=`^`)
```

and yields one spectrum to a power of a second one. Such additional functions are not predefined, as I cannot think of any use for them. oper_spectra is used internally to define the functions for the four basic maths operators, and the corresponding operators.

3.3.12 Task: trimming a spectrum

This is basically a subsetting operation, but our functions operate only based on wavelengths, while R subset is more general. On the other hand, our functions trim_spct and trim_tails add a few 'bells and whistles'. The trimming is based on wavelengths and by default the cut points are inserted by interpolation, so that the spectrum returned includes the limits given as arguments. In addition, by default the trimming is done by deleting both spectral irradiance and wavelength values outside the range delimited by the limits (just like subset does), but through parameter fill the values outside the limits can be replaced by any value desired (most commonly NA or 0.) It is possible to supply only one, or both of low.limit and high.limit, depending on the desired trimming, or use a waveband definition. If the limits are outside the original data set, then the output spectrum is expanded and the tails filled with the value given as argument for fill.

```
trim_spct(my_sun.spct, UV())

## Warning in trim_spct(my_sun.spct, UV()):  Not trimming short
end as low.limit is outside spectral data range.

trim_spct(my_sun.spct, UV(), fill=0)
trim_spct(my_sun.spct, low.limit=400)
trim_spct(my_sun.spct, low.limit=250, fill=0.0)
```

trim_tails can be used for trimming spectra when data is available as vectors. We here present different examples for both functions, we encourage readers to try to reproduce all examples using both functions.

```
# not run
with(sun.data,
     trim_tails(w.length, s.e.irrad,
                low.limit=300))
```

[2]The reason behind keeping e2q as a separately called function is that otherwise calculations would be slowed-down by doing the conversion when it is not needed, either at intermediate steps in the calculation, or when the user has no use for the result

```
with(sun.data,
     trim_tails(w.length, s.e.irrad,
                low.limit=300, fill=NULL))
with(sun.data,
     trim_tails(w.length, s.e.irrad,
                low.limit=300, fill=NA))
with(sun.data,
     trim_tails(w.length, s.e.irrad,
                low.limit=300, fill=0.0))
```

If the limits are outside the range of the input spectral data, and `fill` is set to a value other than NULL the output is expanded up to the limits and filled.

```
# not run
with(sun.data,
     trim_tails(w.length, s.e.irrad,
                low.limit=300, high.limit=1000))
with(sun.data,
     trim_tails(w.length, s.e.irrad,
                low.limit=300, high.limit=1000, fill=NA))
with(sun.data,
     trim_tails(w.length, s.e.irrad,
                low.limit=300, high.limit=1000, fill=0.0))
```

3.3.13 Task: conversion from energy to photon base

The energy of a quantum of radiation in a vacuum, q, depends on the wavelength, λ, or frequency[3], ν,

$$q = h \cdot \nu = h \cdot \frac{c}{\lambda} \tag{3.1}$$

with the Planck constant $h = 6.626 \times 10^{-34}$ J s and speed of light in vacuum $c = 2.998 \times 10^{8}$ m s^{-1}. When dealing with numbers of photons, the equation (3.1) can be extended by using Avogadro's number $N_A = 6.022 \times 10^{23}$ mol^{-1}. Thus, the energy of one mole of photons, q', is

$$q' = h' \cdot \nu = h' \cdot \frac{c}{\lambda} \tag{3.2}$$

with $h' = h \cdot N_A = 3.990 \times 10^{-10}$ J s mol^{-1}.

Function `as_quantum` converts W m^{-2} into *number of photons* per square meter per second, and `as_quantum_mol` does the same conversion but returns mol m^{-2} s^{-1}. Function `as_quantum` is based on the equation 3.1 while `as_quantum_mol` uses equation 3.2. To obtain µmol m^{-2} s^{-1} we multiply by 10^6:

```
as_quantum_mol(550, 200) * 1e6

## [1] 919.5147
```

[3]Wavelength and frequency are related to each other by the speed of light, according to $\nu = c/\lambda$ where c is speed of light in vacuum. Consequently there are two equivalent formulations for equation 3.1.

The calculation above is for monochromatic light (200 $\mathrm{W\,m^{-2}}$ at 550 nm).

The functions are vectorized, so they can be applied to whole spectra (when data are available as vectors), to convert $\mathrm{W\,m^{-2}\,nm^{-1}}$ to $\mathrm{mol\,m^{-2}\,s^{-1}\,nm^{-1}}$:

```
head(sun.data$s.e.irrad, 10)

##  [1] 2.609665e-06 6.142401e-06 2.176175e-05
##  [4] 6.780119e-05 1.533491e-04 3.669677e-04
##  [7] 7.845430e-04 1.264554e-03 2.623718e-03
## [10] 3.922583e-03

s.q.irrad <- with(sun.data,
                  as_quantum_mol(w.length, s.e.irrad))
head(s.q.irrad, 10)

##  [1] 6.391730e-12 1.509564e-11 5.366385e-11
##  [4] 1.677626e-10 3.807181e-10 9.141345e-10
##  [7] 1.960893e-09 3.171207e-09 6.601607e-09
## [10] 9.902505e-09
```

Once again, easiest is to use spectral objects. The default is to add `s.q.irrad` to the source spectrum, unless it is already present in the object in which case values are not recalculated.

```
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

my_sun.spct <- copy(sun.spct)
e2q(my_sun.spct)
```

`e2q` has a parameter `action`, with default `"add"`. Another valid argument value is `"replace"`, but it should be used with extreme care, as the returned object, is no longer a `source.spct` object and is not compatible with all operators and functions defined for `source.spct` objects.

```
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

my_sun.spct <- copy(sun.spct)
e2q(my_sun.spct, "replace")
my_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
```

3.3.14 Task: conversion from photon to energy base

`as_energy` is the inverse function of `as_quantum_mol`:

In Aphalo et al. 2012 it is written: “Example 1: red light at 600 nm has about 200 $\mathrm{kJ\,mol^{-1}}$, therefore, 1 μmol photons has 0.2 J. Example 2: UV-B radiation at 300 nm has about 400 $\mathrm{kJ\,mol^{-1}}$, therefore, 1 μmol photons has 0.4 J. Equations 3.1 and 3.2 are valid for all kinds of electromagnetic waves.” Let’s re-calculate the exact values—as the output from `as_energy` is expressed in $\mathrm{J\,mol^{-1}}$ we multiply the result by 10^{-3} to obtain $\mathrm{kJ\,mol^{-1}}$:

```
as_energy(600, 1) * 1e-3

## [1] 199.3805

as_energy(300, 1) * 1e-3

## [1] 398.7611
```

Because of vectorization we can also operate on a whole spectrum:

```
s.e.irrad <- with(sun.data, as_energy(w.length, s.q.irrad))
```

Function `q2e` is the reverse of `e2q`, it is rarely needed in user code and `source.spct` objects almost always contain `s.e.irrad`. It can also be used as a roundabout way of removing a `s.q.irrad` column, which cloud be usefull when some objects may be missing spectral energy itrradiance data.

```
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

my_sun.spct <- copy(sun.spct)
q2e(my_sun.spct, "replace")
```

Otherwise it feels more natural to use the following data.table syntax:

```
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

my_sun.spct <- copy(sun.spct)
my_sun.spct[ , s.q.irrad := NULL]

##      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 4.080616e-01
## 505:       797 4.141204e-01
## 506:       798 4.236281e-01
## 507:       799 4.185850e-01
## 508:       800 4.069055e-01
```

As we have seen above by default `q2e` and `e2q` return a modified copy of the spectrum as a new object. This is safe, but inefficient in use of memory and computing resources. We first copy the data to a new object, and delete the `s.e.irrad` variable, so that we can test the use of the functions by reference.

```
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

my_sun.spct <- copy(sun.spct)
my_sun.spct[ , s.e.irrad := NULL]

##      w.length    s.q.irrad
##    1:       293 6.391730e-12
##    2:       294 1.509564e-11
##    3:       295 5.366385e-11
##    4:       296 1.677626e-10
##    5:       297 3.807181e-10
##   ---
## 504:       796 2.715219e-06
## 505:       797 2.758995e-06
## 506:       798 2.825879e-06
## 507:       799 2.795738e-06
## 508:       800 2.721132e-06
```

When parameter `byref` is given TRUE as argument the original spectrum is modified.

```
q2e(my_sun.spct, byref=TRUE)
my_sun.spct

##      w.length    s.q.irrad    s.e.irrad
##    1:       293 6.391730e-12 2.609665e-06
##    2:       294 1.509564e-11 6.142401e-06
##    3:       295 5.366385e-11 2.176175e-05
##    4:       296 1.677626e-10 6.780119e-05
##    5:       297 3.807181e-10 1.533491e-04
##   ---
## 504:       796 2.715219e-06 4.080616e-01
## 505:       797 2.758995e-06 4.141204e-01
## 506:       798 2.825879e-06 4.236281e-01
## 507:       799 2.795738e-06 4.185850e-01
## 508:       800 2.721132e-06 4.069055e-01
```

3.3.15 Task: interpolating a spectrum

Functions `interpolate_spct` and `interpolate_spectrum` allow interpolation to different wavelength values. `interpolate_spectrum` is used internally, and accepts spectral data measured at arbitrary wavelengths. Raw data from array spectrometers is not available with a constant wavelength step. It is always best to do any interpolation as late as possible in the data analysis.

In this example we generate interpolated data for the range 280 nm to 300 nm at 1 nm steps, by default output values outside the wavelength range of the input are set to NAs unless a different argument is provided for parameter `fill`:

```
interpolate_spct(sun.spct, seq(290, 300, by=0.1))

##       w.length   s.e.irrad    s.q.irrad
##    1:    290.0          NA           NA
##    2:    290.1          NA           NA
##    3:    290.2          NA           NA
##    4:    290.3          NA           NA
##    5:    290.4          NA           NA
##   ---
##   97:    299.6 0.001072550 2.687082e-09
##   98:    299.7 0.001120551 2.808113e-09
##   99:    299.8 0.001168552 2.929144e-09
##  100:    299.9 0.001216553 3.050176e-09
##  101:    300.0 0.001264554 3.171207e-09

interpolate_spct(sun.spct, seq(290, 300, by=0.1), fill=0.0)

##       w.length   s.e.irrad    s.q.irrad
##    1:    290.0 0.000000000 0.000000e+00
##    2:    290.1 0.000000000 0.000000e+00
##    3:    290.2 0.000000000 0.000000e+00
##    4:    290.3 0.000000000 0.000000e+00
##    5:    290.4 0.000000000 0.000000e+00
##   ---
##   97:    299.6 0.001072550 2.687082e-09
##   98:    299.7 0.001120551 2.808113e-09
##   99:    299.8 0.001168552 2.929144e-09
##  100:    299.9 0.001216553 3.050176e-09
##  101:    300.0 0.001264554 3.171207e-09
```

`interpolate_spct` takes any `__.spct` object, and returns an object of the same type as its input. It can be used to interpolate source spectra as well as transmittance, reflectance, response, and even generic spectra.

`interpolate_spectrum` takes numeric vectors as arguments, but is otherwise functionally equivalent.

```
with(sun.dt,
     interpolate_spectrum(w.length, s.e.irrad, 290:300))

##   [1]           NA           NA           NA
##   [4] 2.609665e-06 6.142401e-06 2.176175e-05
##   [7] 6.780119e-05 1.533491e-04 3.669677e-04
##  [10] 7.845430e-04 1.264554e-03

with(sun.dt,
     interpolate_spectrum(w.length, s.e.irrad, 290:300, fill=0.0))

##   [1] 0.000000e+00 0.000000e+00 0.000000e+00
##   [4] 2.609665e-06 6.142401e-06 2.176175e-05
##   [7] 6.780119e-05 1.533491e-04 3.669677e-04
##  [10] 7.845430e-04 1.264554e-03
```

These functions, in their current implementation, always return interpolated values, even when the density of wavelengths in the output is less than that in the input. A future version of the package will include a `smooth_spectrum` function, and possibly a `remap_w.length` function that will automatically choose between interpolation and smoothing/averaging as needed.

3.4 Wavebands

3.4.1 How are wavebands stored?

Wavebands are derived from R lists. All valid R operations for lists can be also used with `waveband` objects. However, there are `waveband`-specific specializations of generic R methods.

3.4.2 How can the user create waveband objects

Wavebands are created by means of function `waveband` or function `new_waveband` which have in addition to the initial parameter(s) giving the wavelength range, the same additional arguments, also with the same default values.

The simplest `waveband` creation call is one supplying as argument just any R object for which the `range` function returns the wavelength limits of the desired band in nanometres. Such a call yields an un-weighted `waveband` definition, describing a range of wavelengths.

Any numeric vector of at least two elements, any spectral object or any existing `waveband` object is valid input.

```
waveband(c(300, 400))

## range.300.400
## low (nm) 300
## high (nm) 400
## weighted none
```

As you can see above, a name and label are created automatically for the new `waveband`. The user can also supply these as arguments, but must be careful not to duplicate existing names[4].

```
waveband(c(300, 400), wb.name="a.name")

## a.name
## low (nm) 300
## high (nm) 400
## weighted none
```

```
waveband(c(300, 400), wb.name="a.name", wb.label="A nice name")
```

[4]It is preferable that `wb.name` complies with the requirements for R object names and file names, while labels have fewer restrictions as they are meant to be used only for output text labels.

```
## a.name
## low (nm) 300
## high (nm) 400
## weighted none
```

An alterantive function, taking two numbers, giving the boundaries of the waveband is also available.

```
new_waveband(300, 400)

## range.300.400
## low (nm) 300
## high (nm) 400
## weighted none
```

```
new_waveband(300, 400, wb.name="a.name")

## a.name
## low (nm) 300
## high (nm) 400
## weighted none
```

```
new_waveband(300, 400, wb.name="a.name", wb.label="A nice name")

## a.name
## low (nm) 300
## high (nm) 400
## weighted none
```

See chapter 4 on page 39, in particular sections 4.4, 4.3, and 4.5 for further examples, and a more in-depth discussion of the creation and use of *un-weighted* `waveband` objects.

For both functions, even if we supply a *weighting function* (SWF), a lot of flexibility remains. One can supply either a function that takes energy irradiance as input or a function that takes photon irradiance as input. Unless both are supplied, the missing function will be automatically created. There are also arguments related to normalization, both of the output, and of the SWF supplied as argument. In the examples above, 'hinges' are created automatically for the range extremes. When using SWF with discontinuous derivatives, best results are obtained by explicitly supplying the hinges to be used as an argument to `new_waveband` call. An example follows for the definition of a waveband for the CIE98 SWF—the function `CIE.e.fun` is defined in package photobiologyWavebands but any R function taking a numeric vector of wavelengths as input and returning a numeric vector of the same length containing weights can be used.

```
waveband(c(250, 400),
         weight="SWF", SWF.e.fun=CIE.e.fun, SWF.norm=298,
         norm=298, hinges=c(249.99, 250, 298, 328, 399.99, 400),
         wb.name="CIE98.298", wb.label="CIE98")
```

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

```
new_waveband(w.low=250, w.high=400,
             weight="SWF", SWF.e.fun=CIE.e.fun, SWF.norm=298,
             norm=298, hinges=c(249.99, 250, 298, 328, 399.99, 400),
             wb.name="CIE98.298", wb.label="CIE98")

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

See chapter 5 on page 57, in particular sections **??**, **??**, and **??** for further examples, and a more in-depth discussion of the creation and use of *weighted* `waveband` objects.

3.4.3 What ‘summary’ functions are available for wavebands?

Special methods for `wavebands` of `print` giving more compact output than the default `print` method for lists. In addition, `range, min, max` when applied to wavebands return values corresponding to wavelengths, other generic functions defined in the suite give additional summaries of wavebands `spread`, `midpoint`, `color`, `labels`.

3.4.4 Operators and functions

Several functions described in chapters 4, 5, and **??** use `waveband` objects as arguments. Those functions provide selective summaries of spectra.

```
e_irrad(sun.data, UVB())

##   UVB.ISO
## 0.5881141
## attr(,"time.unit")
## [1] "second"
```

Multiplying a source spectrum by an un-weighted waveband, is equivalent to trimming with `fill` set to NA.

```
sun.spct * UVA()

##      w.length s.e.irrad
##   1:      293         0
##   2:      294         0
##   3:      295         0
##   4:      296         0
##   5:      297         0
##  ---
## 504:      796         0
## 505:       797          0
## 506:       798          0
## 507:       799          0
## 508:       800          0
```

Multiplying a source spectrum by a weighted waveband convolutes the spectrum with weights, yielding effective spectral irradiance.

```
sun.spct * CIE()

##      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
```

3.5 Internal-use functions

The generic function `check` can be used on any type of `.spct` object, and depending on its types checks that the required components are present. If they are missing they are added. If it is possible to calculate the missing values from other optional components, they are calculated, otherwise they are filled with NA. It is used internally during the creation of spectral objects.

The function `check_spectrum` may need to be called by the user if he/she disables automatic sanity checking to increase calculation speed. The family of functions for calculating multipliers are used internally by the package.

The function `insert_hinges` is used internally to insert individual interpolated values to the spectra when needed to reduce errors in calculations.

The function `integrate_irradiance` is used internally for integrating spectra, and accepts spectral data measured at arbitrary wavelengths. Raw data from array spectrometers is not available with a constant wavelength step. It is always best to do any interpolation as late as possible, or never. This function makes it possible to work with spectral data on the original pixel wavelengths.

```
try(detach(package:photobiologyFilters))
try(detach(package:photobiologyLEDs))
try(detach(package:photobiologyWavebands))
try(detach(package:photobiology))
```

CHAPTER 4

Unweighted irradiance

Abstract

In this chapter we explain how to calculate unweighted energy and photon irradiances from spectral irradiance.

4.1 Packages used in this chapter

For executing the examples listed in this chapter you need first to load the following packages from the library:

```
library(photobiology)
library(photobiologyWavebands)
```

4.2 Introduction

Functions `e_irrad` and `q_irrad` return energy irradiance and photon (or quantum) irradiance, and both take as argument a `source.spct` object containing either spectral (energy) irradiance or spectral photon irradiance data. An additional parameter accepting a `waveband` object, or a list of `waveband` objects, can be used to set the range(s) of wavelengths and spectral weighting function(s) to use for integration(s). Two additional functions, `energy_irradiance` and `photon_irradiance`, are defined for equivalent calculations on spectral irradiance data stored as numeric vectors.

We start by describing how to use and define `waveband` objects, for which we need to use function `e_irrad` in some examples before a detailed explanation of its use (see section 4.6) on page 46 for details).

4.3 Task: use simple predefined wavebands

Please, consult the packages' documentation for a list of predefined functions for creating wavebands also called `waveband` *constructors*. Here we will present just a few examples of their use. We usually associate wavebands with colours, however, in many cases there are different definitions in use. For this reason, the functions provided accept an argument that can be used to select the definition to use. In general, the default, is to use the ISO standard whenever it is applicable. The case of the various definitions in use for the UV-B waveband are described on page 41

We can use a predefined function to create a new `waveband` object, which as any other R object can be assigned to a variable:

```
uvb <- UVB()
uvb

## UVB.ISO
## low (nm) 280
## high (nm) 315
## weighted none
```

As seen above, there is a specialized `print` method for `wavebands`. `waveband` methods returning wavelength values in nm are `min`, `max`, `range`, `midpoint`, and `spread`. Method `labels` returns the name and label stored in the waveband, and method `color` returns a color definition calculated from the range of wavelengths.

```
red <- Red()
red

## Red.ISO
## low (nm) 610
## high (nm) 760
## weighted none

min(red)

## [1] 610

max(red)

## [1] 760

range(red)

## [1] 610 760

midpoint(red)

## [1] 685

spread(red)

## [1] 150
```

```
labels(red)

## $label
## [1] "Red"
##
## $name
## [1] "Red.ISO"

color(red)

## $CMF
##   Red.CMF
## "#900000"
##
## $CC
##    Red.CC
## "#FF0000"
```

The argument `standard` can be used to choose a given alternative definition[1]:

```
UVB()

## UVB.ISO
## low (nm) 280
## high (nm) 315
## weighted none

UVB("ISO")

## UVB.ISO
## low (nm) 280
## high (nm) 315
## weighted none

UVB("CIE")

## UVB.CIE
## low (nm) 280
## high (nm) 315
## weighted none

UVB("medical")

## UVB.medical
## low (nm) 290
## high (nm) 320
## weighted none

UVB("none")

## UVB.none
## low (nm) 280
## high (nm) 320
## weighted none
```

[1]When available, the definition in the ISO standard is the default.

Here we demonstrate the importance of complying with standards, and how much photon irradiance can depend on the definition used in the calculation.

```
e_irrad(sun.spct, UVB("ISO"))

##   UVB.ISO
## 0.5881141
## attr(,"time.unit")
## [1] "second"

e_irrad(sun.spct, UVB("none"))

## UVB.none
## 1.250093
## attr(,"time.unit")
## [1] "second"

e_irrad(sun.spct, UVB("ISO")) / e_irrad(sun.spct, UVB("none"))

##   UVB.ISO
## 0.4704563
## attr(,"time.unit")
## [1] "second"
```

4.4 Task: define simple wavebands

Here we briefly introduce `waveband` and `new_waveband`, and only in chapter **??** we describe their use in full detail, including the use of spectral weighting functions (SWFs). The examples in the present section only describe `wavebands` that define a wavelength range.

A `waveband` can be created based on any R object for which function `range` is defined, and returns numbers interpretable as wavelengths expressed in nanometres:

```
waveband(c(400,700))

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

waveband(400:700)

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

waveband(sun.spct)

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

wb_total <- waveband(sun.spct, wb.name="total")
```

```
e_irrad(sun.spct, wb_total)

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

A `waveband` can also be created based on extreme wavelengths expressed in nm.

```
wb1 <- new_waveband(500,600)
wb1

## range.500.600
## low (nm) 500
## high (nm) 600
## weighted none

e_irrad(sun.spct, wb1)

## range.500.600
##      68.53291
## attr(,"time.unit")
## [1] "second"

wb2 <- new_waveband(500,600, wb.name="my.colour")
wb2

## my.colour
## low (nm) 500
## high (nm) 600
## weighted none

e_irrad(sun.spct, wb2)

## my.colour
##  68.53291
## attr(,"time.unit")
## [1] "second"
```

4.5 Task: define lists of simple wavebands

Lists of wavebands can be created by grouping `waveband` objects using the R-defined constructor `list`,

```
UV.list <- list(UVC(), UVB(), UVA())
UV.list

## [[1]]
## UVC.ISO
## low (nm) 100
## high (nm) 280
## weighted none
##
## [[2]]
## UVB.ISO
## low (nm) 280
## high (nm) 315
## weighted none
##
## [[3]]
## UVA.ISO
## low (nm) 315
## high (nm) 400
## weighted none
```

in which case wavebands can be non-contiguous and/or overlapping.

In addition function `split_bands` can be used to create a list of contiguous wavebands by supplying a numeric vector of wavelength boundaries in nanometres,

```
split_bands(c(400,500,600))

## $wb1
## range.400.500
## low (nm) 400
## high (nm) 500
## weighted none
##
## $wb2
## range.500.600
## low (nm) 500
## high (nm) 600
## weighted none
```

or with longer but more meaningful names,

```
split_bands(c(400,500,600), short.names=FALSE)

## $range.400.500
## range.400.500
## low (nm) 400
## high (nm) 500
## weighted none
##
## $range.500.600
## range.500.600
## low (nm) 500
## high (nm) 600
## weighted none
```

It is also possible to also provide the limits of the region to be covered by the list of wavebands and the number of (equally spaced) wavebands desired:

```
split_bands(c(400,600), length.out=2)

## $wb1
## range.400.500
## low (nm) 400
## high (nm) 500
## weighted none
##
## $wb2
## range.500.600
## low (nm) 500
## high (nm) 600
## weighted none
```

in all cases coderange is used to find the list boundaries, so we can also split the region defined by an existing `waveband` object into smaller wavebands,

```
split_bands(PAR(), length.out=3)

## $wb1
## range.400.500
## low (nm) 400
## high (nm) 500
## weighted none
##
## $wb2
## range.500.600
## low (nm) 500
## high (nm) 600
## weighted none
##
## $wb3
## range.600.700
## low (nm) 600
## high (nm) 700
## weighted none
```

or split a whole spectrum[2] into equally sized regions,

```
split_bands(sun.spct, length.out=3)

## $wb1
## range.293.462
## low (nm) 293
## high (nm) 462
## weighted none
##
## $wb2
## range.462.631
## low (nm) 462
## high (nm) 631
## weighted none
##
## $wb3
## range.631.800
## low (nm) 631
## high (nm) 800
## weighted none
```

It is also possible to supply a list of wavelength ranges[3], and, when present, names are copied from the input list to the output list:

[2]This is not restricted to `source.spct` objects as all other classes of `____.spct` objects also have `range` methods defined.

[3]When using a list argument, even overlapping and non-contiguous wavelength ranges are valid input

```
split_bands(list(c(400,500), c(600,700)))

## $wb1
## range.400.500
## low (nm) 400
## high (nm) 500
## weighted none
##
## $wb2
## range.600.700
## low (nm) 600
## high (nm) 700
## weighted none

split_bands(list(blue=c(400,500), PAR=c(400,700)))

## $blue
## range.400.500
## low (nm) 400
## high (nm) 500
## weighted none
##
## $PAR
## range.400.700
## low (nm) 400
## high (nm) 700
## weighted none
```

Package photobiologyWavebands also predefines some useful constructors of lists of wavebands, currently `VIS_bands`, `UV_bands` and `Plant_bands`.

```
UV_bands()

## [[1]]
## UVC.ISO
## low (nm) 100
## high (nm) 280
## weighted none
##
## [[2]]
## UVB.ISO
## low (nm) 280
## high (nm) 315
## weighted none
##
## [[3]]
## UVA.ISO
## low (nm) 315
## high (nm) 400
## weighted none
```

4.6 Task: (energy) irradiance from spectral irradiance

The task to be completed is to calculate the (energy) irradiance (E) in $\mathrm{W\,m^{-2}}$ from spectral (energy) irradiance ($E(\lambda)$) in $\mathrm{W\,m^{-2}\,nm^{-1}}$ and the corresponding wavelengths (λ) in nm.

$$E_{\lambda_1<\lambda<\lambda_2} = \int_{\lambda_1}^{\lambda_2} E(\lambda)\ \mathrm{d}\ \lambda \tag{4.1}$$

Let’s assume that we want to calculate photosynthetically active radiation (PAR) energy irradiance, for which the most accepted limits are $\lambda_1 = 400$nm and $\lambda_1 = 700$nm. In this example we will use example data for sunlight to calculate $E_{400\,\mathrm{nm}<\lambda<700\,\mathrm{nm}}$. The function used for this task when working with spectral objects is `e_irrad` returning energy irradiance. The "names" of the returned valued is set according to the waveband used, and `sun.spct` is a `source.spct` object.

```
e_irrad(sun.spct, waveband(c(400,700)))

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

or using the PAR waveband constructor, defined in package photobiologyWavebands as a convenience function,

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

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

or if no waveband is supplied as argument, then irradiance is computed for the whole range of wavelengths in the spectral data, and the ‘name’ attribute is generated accordingly.

```
e_irrad(sun.spct)

## range.293.800
##      269.1249
## attr(,"time.unit")
## [1] "second"
```

If a waveband extends outside of the wavelength range of the spectral data, spectral irradiance for unavailable wavelengths is assumed to be zero:

```
e_irrad(sun.spct, waveband(c(100,400)))

## range.100.400
##      28.32466
## attr(,"time.unit")
## [1] "second"

e_irrad(sun.spct, waveband(c(100,250)))

## range.100.250
##             0
## attr(,"time.unit")
## [1] "second"
```

Both `e_irrad` and `q_irrad` accept, in addition to a waveband as second argument, a list of wavebands. In this case, the returned value is a numeric vector of the same length as the list.

```
e_irrad(sun.spct, list(UVB(), UVA()))

##    UVB.ISO    UVA.ISO
##  0.5881141 27.7365487
## attr(,"time.unit")
## [1] "second"
```

Storing emission spectral data in `source.spct` objects is recommended, as it allows better protection against mistakes, and allows automatic detection of input data base of expression and units. However, it may be sometimes more convenient or efficient to keep spectral data in individual numeric vectors, or data frames. In such cases function `energy_irradiance`, which accepts the spectral data as vectors can be used at the cost of less concise code and weaker error tests. In this case, the user must indicate whether spectral data is on energy or photon based units through parameter `unit.in`, which defaults to `"energy"`.

For example when using function `PAR()`, the code above becomes:

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

##      PAR
## 196.7004

with(sun.data,
     energy_irradiance(w.length, s.e.irrad, PAR(), unit.in="energy"))

##      PAR
## 196.7004
```

where `sun.data` is a data frame. However, the data can also be stored in separate numeric vectors of equal length.

The `sun.data` data frame also contains spectral photon irradiance values:

```
names(sun.data)

## [1] "w.length"  "s.e.irrad" "s.q.irrad"
```

which allows us to use:

```
with(sun.data,
     energy_irradiance(w.length, s.q.irrad, PAR(), unit.in="photon"))

##      PAR
## 196.7004
```

The other examples above can be re-written with similar syntax.

4.7 Task: photon irradiance from spectral irradiance

The task to be completed is to calculate the photon irradiance (Q) in $\mathrm{mol\,m^{-2}\,s^{-1}}$ from spectral (energy) irradiance ($E(\lambda)$) in $\mathrm{W\,m^{-2}\,nm^{-1}}$ and the corresponding wavelengths (λ) in nm.

Combining equations 4.1 and 3.2 we obtain:

$$Q_{\lambda_1<\lambda<\lambda_2} = \int_{\lambda_1}^{\lambda_2} E(\lambda)\ \frac{h' \cdot c}{\lambda} \mathrm{d}\,\lambda \tag{4.2}$$

Let's assume that we want to calculate photosynthetically active radiation (PAR) photon irradiance (frequently called PPFD or photosynthetic photon flux density), for which the most accepted limits are $\lambda_1 = 400$nm and $\lambda_1 = 700$nm. In this example we will use example data for sunlight to calculate $E_{400\,\mathrm{nm}<\lambda<700\,\mathrm{nm}}$. The function used for this task when working with spectral objects is `q_irrad`, returning photon irradiance in $\mathrm{mol\,m^{-2}\,s^{-1}}$. The "names" of the returned valued is set according to the waveband used, and `sun.spct` is a `source.spct` object.

```
q_irrad(sun.spct, waveband(c(400,700)))

## range.400.700
##  0.0008937598
## attr(,"time.unit")
## [1] "second"
```

to obtain the photon irradiance expressed in $\mathrm{\mu mol\,m^{-2}\,s^{-1}}$ we multiply the returned value by 1×10^6:

```
q_irrad(sun.spct, waveband(c(400,700))) * 1e6

## range.400.700
##      893.7598
## attr(,"time.unit")
## [1] "second"
```

or using the PAR waveband constructor, defined in package photobiologyWavebands as a convenience function,

```
q_irrad(sun.spct, PAR()) * 1e6

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

Examples given in section 4.6 can all be converted by replacing `e_irrad` function calls with `q_irrad` function calls.

Storing emission spectral data in `source.spct` objects is recommended (see section 4.6). However, it may be sometimes more convenient or efficient to keep spectral data in individual numeric vectors, or data frames. In such cases function `photon_irradiance`, which accepts the spectral data as vectors can be used at the cost of less concise code and weaker error tests. In this case, the user must indicate whether spectral data is on energy or photon based units through parameter `unit.in`, which defaults to `"energy"`.

For example when using function `PAR()`, the code above becomes:

```
with(sun.data,
     photon_irradiance(w.length, s.e.irrad, PAR()), unit.in="energy")  * 1e6
```

```
##      PAR
## 893.7598

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

##      PAR
## 893.7598
```

where `sun.data` is a data frame. However, the data can also be stored in separate numeric vectors of equal length.

4.8 Task: irradiances for more than one waveband

As discussed above, it is possible to calculate simultaneously the irradiances for several wavebands with a single function call by supplying a `list` of `wavebands` as argument:

```
q_irrad(sun.spct, list(Red(), Green(), Blue())) * 1e6

##   Red.ISO Green.ISO  Blue.ISO
##  452.1700  220.1562  148.9735
## attr(,"time.unit")
## [1] "second"

Q.RGB <- q_irrad(sun.spct, list(Red(), Green(), Blue())) * 1e6
signif(Q.RGB, 3)

##   Red.ISO Green.ISO  Blue.ISO
##       452       220       149
## attr(,"time.unit")
## [1] "second"

Q.RGB[1]

## Red.ISO
##  452.17

Q.RGB["Green.ISO"]

## Green.ISO
##  220.1562
```

as the value returned is in $\mathrm{mol\,m^{-2}\,s^{-1}}$ we multiply it by 1×10^6 to obtain $\mathrm{\mu mol\,m^{-2}\,s^{-1}}$.

A named list can be used to override the names used for the output:

```
q_irrad(sun.spct, list(R=Red(), G=Green(), B=Blue())) * 1e6

##         R         G         B
## 452.1700 220.1562 148.9735
## attr(,"time.unit")
## [1] "second"
```

Even when using a single waveband:

```
q_irrad(sun.spct, list('ultraviolet-B'=UVB())) * 1e6

## ultraviolet-B
##      1.526862
## attr(,"time.unit")
## [1] "second"
```

The examples above, can be easily rewritten using functions `e_irrad`, `energy_irradiance` or `photon_irradiance`.

For example, the second example above becomes:

```
e_irrad(sun.spct, list(R=Red(), G=Green(), B=Blue()))

##        R        G        B
## 79.61176 49.30478 37.57760
## attr(,"time.unit")
## [1] "second"
```

or

```
with(sun.data,
     energy_irradiance(w.length, s.e.irrad,
                       list(R=Red(), G=Green(), B=Blue())))

##        R        G        B
## 79.61176 49.30478 37.57760
```

4.9 Task: photon ratios

In photobiology sometimes we are interested in calculation the photon ratio between two wavebands. It makes more sense to calculate such ratios if both numerator and denominator wavebands have the same ‘width’ or if the numerator waveband is fully nested in the denominator waveband. However, frequently used ratios like the UV-B to PAR photon ratio do not comply with this. For this reason, our functions do not enforce any such restrictions.

For example a ratio frequently used in plant photobiology is the red to far-red photon ratio (R:FR photon ratio or ζ). If we follow the wavelength ranges in the definition given by **Morgan1981a** using photon irradiance[4]:

$$\zeta = \frac{Q_{655\mathrm{nm}<\lambda<665\mathrm{nm}}}{Q_{725\mathrm{nm}<\lambda<735\mathrm{nm}}} \tag{4.3}$$

To calculate this for our example sunlight spectrum we can use the following code:

```
q_ratio(sun.spct, Red("Smith"), Far_red("Smith"))

## Red.Smith:FarRed.Smith(q:q)
##                    1.251099
```

[4]In the original text photon fluence rate is used but it not clear whether photon irradiance was meant instead.

Function `q_ratio` also accepts lists of wavebands, for both denominator and numerator arguments, and recycling takes place when needed. Calculation of the contribution of different colors to visible light, using ISO-standard definitions.

```
q_ratio(sun.spct, UVB(), list(UV(), VIS()))

##  UVB.ISO:UV.ISO(q:q) UVB.ISO:VIS.ISO(q:q)
##          0.017862550          0.001404725
```

```
q_ratio(sun.spct,
         list(Red(), Green(), Blue()), VIS())

##   Red.ISO:VIS.ISO(q:q) Green.ISO:VIS.ISO(q:q)
##              0.4159998              0.2025454
##  Blue.ISO:VIS.ISO(q:q)
##              0.1370567
```

or using a predefined list of wavebands:

```
q_ratio(sun.spct, VIS_bands(), VIS())

## Purple.ISO:VIS.ISO(q:q)   Blue.ISO:VIS.ISO(q:q)
##              0.15004110              0.13705675
##  Green.ISO:VIS.ISO(q:q) Yellow.ISO:VIS.ISO(q:q)
##              0.20254538              0.06110784
## Orange.ISO:VIS.ISO(q:q)    Red.ISO:VIS.ISO(q:q)
##              0.05533039              0.41599981
```

Using spectral data stored in numeric vectors:

```
with(sun.data,
     photon_ratio(w.length, s.e.irrad,  Red("Smith"), Far_red("Smith")))

## [1] 1.251099
```

or using the predefined convenience function `R_FR_ratio`:

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

## [1] 1.251099
```

4.10 Task: energy ratios

An energy ratio, equivalent to ζ can be calculated as follows:

```
e_ratio(sun.spct, Red("Smith"), Far_red("Smith"))

## Red.Smith:FarRed.Smith(e:e)
##                    1.384353
```

other examples in section 4.9 above, can be easily edited to use `e_ratio` instead of `q_ratio`.

Using spectral data stored in vectors:

```
with(sun.data,
     energy_ratio(w.length, s.e.irrad,
                  Red("Smith"), Far_red("Smith")))

## [1] 1.384353
```

For this infrequently used ratio, no pre-defined function is provided.

4.11 Task: calculate average number of photons per unit energy

When comparing photo-chemical and photo-biological responses under different light sources it is of interest to calculate the photons per energy in $\mathrm{mol\,J^{-1}}$. In this case only one waveband definition is used to calculate the quotient:

$$\bar{q}' = \frac{Q_{\lambda_1<\lambda<\lambda_2}}{E_{\lambda_1<\lambda<\lambda_2}} \tag{4.4}$$

From this equation it follows that the value of the ratio will depend on the shape of the emission spectrum of the radiation source. For example, for PAR the R code is:

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

##      q:e(PAR)
## 4.543762e-06
```

for obtaining the same quotient in $\mathrm{\mu mol\,J^{-1}}$ we just need to multiply by 1×10^6,

```
qe_ratio(sun.spct, PAR()) * 1e6

## q:e(PAR)
## 4.543762
```

The seldom needed inverse ratio in $\mathrm{J\,mol^{-1}}$ can be calculated with function `eq_ratio`.

Both functions accept lists of wavebands, so several ratios can be calculated with a single function call:

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

## q:e(Purple.ISO)   q:e(Blue.ISO)  q:e(Green.ISO)
##    3.429575e-06    3.964423e-06    4.465210e-06
## q:e(Yellow.ISO) q:e(Orange.ISO)    q:e(Red.ISO)
##    4.847365e-06    5.016828e-06    5.679688e-06
```

The same ratios can be calculated for data stored in numeric vectors using function `photons_energy_ratio`:

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

## [1] 4.543762e-06
```

For obtaining the same quotient in $\mu\text{mol}\,\text{J}^{-1}$ from spectral data in $\text{W}\,\text{m}^{-2}\,\text{nm}^{-1}$ we just need to multiply by 1×10^6:

```
with(sun.data,
     photons_energy_ratio(w.length, s.e.irrad, PAR())) * 1e6

## [1] 4.543762
```

4.12 Task: calculate the contribution of different regions of a spectrum to energy irradiance

It can be of interest to split the total (energy) irradiance into adjacent regions delimited by arbitrary wavelengths. When working with `source.spct` objects, the best way to achieve this is to combine the use of the functions `e_irrad` and `split_bands` already described above, for example,

```
e_irrad(sun.spct, split_bands(c(400, 500, 600, 700)))

##      wb1      wb2      wb3
## 69.63243 68.53291 58.53508
## attr(,"time.unit")
## [1] "second"
```

or

```
e_irrad(sun.spct, split_bands(PAR(), length.out=3))

##      wb1      wb2      wb3
## 69.63243 68.53291 58.53508
## attr(,"time.unit")
## [1] "second"
```

or

```
my_bands <- split_bands(PAR(), length.out=3)
e_irrad(sun.spct, my_bands)

##      wb1      wb2      wb3
## 69.63243 68.53291 58.53508
## attr(,"time.unit")
## [1] "second"
```

For the example immediately above, we can calculate relative values as

```
e_irrad(sun.spct, my_bands) / e_irrad(sun.spct, PAR())

##       wb1       wb2       wb3
## 0.3540024 0.3484126 0.2975849
## attr(,"time.unit")
## [1] "second"
```

or more efficiently as

```
irradiances <- e_irrad(sun.spct, my_bands)
irradiances / sum(irradiances)

##       wb1       wb2       wb3
## 0.3540024 0.3484126 0.2975849
## attr(,"time.unit")
## [1] "second"
```

The examples above use short names, the default, but longer names are also available,

```
e_irrad(sun.spct, split_bands(c(400, 500, 600, 700), short.names=FALSE))

## range.400.500 range.500.600 range.600.700
##      69.63243      68.53291      58.53508
## attr(,"time.unit")
## [1] "second"

e_irrad(sun.spct, split_bands(PAR(), short.names=FALSE, length.out=3))

## range.400.500 range.500.600 range.600.700
##      69.63243      68.53291      58.53508
## attr(,"time.unit")
## [1] "second"
```

With spectral data stored in numeric vectors, we can use function `energy_irradiance` together with function `split_bands` or we can use the convenience function `split_energy_irradiance` to obtain to energy of each of the regions delimited by the values in nm supplied in a numeric vector:

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

## range.400.500 range.500.600 range.600.700
##      69.63243      68.53291      58.53508
```

It possible to obtain the 'split' as a vector of fractions adding up to one,

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

## range.400.500 range.500.600 range.600.700
##     0.3540024     0.3484126     0.2975849
```

or as percentages:

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

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

If the 'limits' cover only a region of the spectral data, relative and percent values will be calculated with that region as a reference.

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

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

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

## range.400.500 range.500.600
##        50.3979       49.6021
```

A vector of two wavelengths is valid input, although not very useful for percentages:

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

## range.400.700
##           100
```

In contrast, for `scale="absolute"`, the default, it can be used as a quick way of calculating an irradiance for a range of wavelengths without having to define a `waveband`:

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

## range.400.700
##      196.7004
```

```
try(detach(package:photobiologyWavebands))
try(detach(package:photobiology))
```

CHAPTER 5

Weighted and effective irradiance

Abstract

In this chapter we explain how to calculate weighted energy and photon irradiances from spectral irradiance.

5.1 Packages used in this chapter

For executing the examples listed in this chapter you need first to load the following packages from the library:

```r
library(photobiology)
library(photobiologyWavebands)
```

5.2 Introduction

Weighted irradiance is usually reported in weighted energy units, but it is also possible to use weighted photon based units. In practice the R code to use is exactly the same as for unweighted irradiances, as all the information needed for applying weights is stored in the `waveband` object. An additional factor comes into play and it is the *normalization wavelength*, which is accepted as an argument by the predefined waveband creation functions that describe biological spectral weighting functions (BSWFs). The focus of this chapter is on the differences between calculations for weighted irradiances compared to those for un-weighted irradiances described in chapter 4. In particular it is important that you read sections **??**, 4.7, on the calculation of irradiances from spectral irradiances and sections 4.3, and 4.4 before reading the present chapter.

Most SWFs are defined using measured action spectra or spectra derived by combining different measured action spectra. As these spectra have been measured under different conditions, what is of interest is the shape of the curve as a function of wavelength, but not the absolute values. Because of this, SWFs are normalized to an action of one at an arbitrary wavelength. In many cases there is no consensus about the wavelength to use. Normalization is simple, it consists in dividing all action values along the curve by the action value at the selected normalization wavelengths.

Another complication is that it is not always clear if a given SWF definition is based on energy or photon units for the fluence rate or irradiances. In photobiology using photon units for expressing action spectra is the norm, but SWFs based on them have rather frequently been used as weights for spectral energy irradiance. The current package makes this difference explicit, and uses the correct weights depending on the spectral data, as long as the `waveband` objects have been correctly defined. In the case of the definitions in package photobiologyWavebands, we have used, whenever possible the correct interpretation when described in the literature, or the common practice when information has been unavailable.

5.3 Task: specifying the normalization wavelength

Several constructors for SWF-based `waveband` objects are supplied. Most of them have parameters, in most cases with default arguments, so that different common uses and misuses in the literature can be reproduced. For example, function `GEN.G()` is predefined in package photobiologyWavebands as a convenience function for Green's formulation of Caldwell's generalized plant action spectrum (GPAS) **Green198x**

```
e_irrad(sun.spct, GEN.G())

## GEN.G.300
## 0.1033597
## attr(,"time.unit")
## [1] "second"
```

The code above uses the default normalization wavelength of 300 nm, which is almost universally used nowadays, but not the value used in the original publication (**Caldwell1973**). Any arbitrary wavelength (nm), within the range of the waveband is accepted as `norm` argument:

```
range(GEN.G())

## [1] 250.0 313.3

e_irrad(sun.spct, GEN.G(280))

##  GEN.G.280
## 0.02402434
## attr(,"time.unit")
## [1] "second"
```

5.4 Task: use of weighted wavebands

Please, consult the documentation of package photobiologyWavebands for a list of predefined constructor functions for weighted wavebands. Here we will present just a few examples of their use. We usually think of weighted irradiances as being defined only by the weighting function, however, as mentioned above, in many cases different normalization wavelengths are in use, and the result of calculations depends very strongly on which wavelength is used for normalization. In a few cases different mathematical formulations are available for the 'same' SWF, and the differences among them can be also important. In such cases separate functions are provided for each formulation (e.g. GEN.N and GEN.T for Green's and Thimijan's formulations of Caldwell's GPAS).

```
GEN.G()

## GEN.G.300
## low (nm) 250
## high (nm) 313
## weighted SWF
## normalized at 300 nm

GEN.T()

## GEN.T.300
## low (nm) 250
## high (nm) 390
## weighted SWF
## normalized at 300 nm
```

We can use one of the predefined functions to create a new waveband object, which as any other R object can be assigned to a variable:

```
cie <- CIE()
cie

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

As described in section 4.3, there are several methods for querying and printing waveband objects. The same functions described for un-weighted waveband objects can be used with any waveband object, including those based on SWFs.

5.5 Task: define wavebands that use weighting functions

In sections ?? and 3.4 we briefly introduced functions waveband and new_waveband, and here we describe their use in full detail. Most users are unlikely to frequently need to define new waveband objects as common SWFs are already defined in package photobiologyWavebands.

Although the constructors are flexible, and can automatically handle both definitions based on action or response spectra in photon or energy units, some care is needed when performance is important.

When defining a new weighted `waveband`, we need to supply to the constructor more information than in the case on un-weighted wavebands. We start with a simple 'toy' example:

```
toy.wb <- waveband(c(400,700), weight="SWF",
                       SWF.e.fun=function(wl){(wl / 550)^2},
                       norm=550, SWF.norm=550,
                       wb.name="TOY")
toy.wb

## TOY
## low (nm) 400
## high (nm) 700
## weighted SWF
## normalized at 550 nm
```

where the first argument is the range of wavelengths included, `weight="SWF"` indicates that spectral weighting will be used, `SWF.e.fun=function(wl)wl * 2 / 550` supplies an 'anonymous' spectral weighting function based on energy units, `norm=550` indicates the default normalization wavelength to use in calculations, `SWF.norm=550` indicates the normalization wavelength of the output of the SWF, and `wb.name="TOY"` gives a name for the waveband.

In the example above the constructor generates automatically the SWF to use with spectral photon irradiance from the function supplied for spectral energy irradiance. The reverse is true if only an SWF for spectral photon irradiance is supplied. If both functions are supplied, they are used, but no test for their consistency is applied.

5.6 Task: calculate effective energy irradiance

We can use the `waveband` object defined above in calculations:

```
e_irrad(sun.spct, toy.wb)

##      TOY
## 196.6993
## attr(,"time.unit")
## [1] "second"
```

Just in the same way as we can use those created with the specific constructors, including using anonymous objects created on the fly:

```
e_irrad(sun.spct, CIE())

##  CIE98.298
## 0.08177754
## attr(,"time.unit")
## [1] "second"
```

or lists of wavebands, such as

```
e_irrad(sun.spct, list(GEN.G(), GEN.T()))

## GEN.G.300 GEN.T.300
## 0.1033597 0.1473573
## attr(,"time.unit")
## [1] "second"
```

or

```
e_irrad(sun.spct, list(GEN.G(280), GEN.G(300)))

##  GEN.G.280  GEN.G.300
## 0.02402434 0.10335965
## attr(,"time.unit")
## [1] "second"
```

Nothing prevents the user from defining his or her own `waveband` object constructors for new SWFs, and making this easy was an important goal in the design of the packages.

5.7 Task: calculate effective photon irradiance

All what is needed is to use function `q_irrad` instead of `e_irrad`. However, one should think carefully if such a calculation is what is needed, as in some research fields it is rarely used, even when from the theoretical point of view would be in most cases preferable.

```
q_irrad(sun.spct, GEN.G())

##   GEN.G.300
## 2.59202e-07
## attr(,"time.unit")
## [1] "second"
```

5.8 Task: calculate daily effective energy exposure

To calculate daily exposure values, we need to apply the same code as used above, but using spectral daily exposure instead of spectral irradiance as starting point:

```
e_irrad(sun.daily.spct, GEN.G())

## GEN.G.300
##  2803.238
## attr(,"time.unit")
## [1] "day"
```

the output from the code above is in units of $\mathrm{J\,m^{-2}\,d^{-1}}$, the code below returns the same result in the more common uints of $\mathrm{kJ\,m^{-2}\,d^{-1}}$:

```
e_irrad(sun.daily.spct, GEN.G()) * 1e-3

## GEN.G.300
##  2.803238
## attr(,"time.unit")
## [1] "day"
```

by comparing these result with those for effective irradiances above, it can be seen that the `time.unit` attribute of the spectral data is copied to the result, allowing us to distinguish irradiance values (`time.unit="second"`) from daily exposure values (`time.unit="day"`).

```
try(detach(package:photobiologyWavebands))
try(detach(package:photobiology))
```

CHAPTER 6

Transmission and reflection

Abstract

In this chapter we explain how to do calculations related to the description of absortion and reflection of UV and VIS radiation.

6.1 Packages used in this chapter

For executing the examples listed in this chapter you need first to load the following packages from the library:

```
library(photobiology)
library(photobiologyWavebands)
library(photobiologyFilters)
library(photobiologyLEDs)
```

6.2 Introduction

6.3 Task: absorbance and transmittance

Transmittance is defined as:

$$\tau(\lambda) = \frac{I}{I_0} = \frac{E(\lambda)}{E_0(\lambda)} = \frac{Q(\lambda)}{Q_0(\lambda)} \tag{6.1}$$

Given this simple relation $\tau(\lambda)$ can be calculated as a division between two "source.spct" objects. This gives the correct answer, but as an object of class "source.scpt".

```
tau <- spc_above / spc_below
```

Absorptance is just $1 - \tau(\lambda)$, but should be distinguished from absorbance ($A(\lambda)$) which is measured on a logarithmic scale:

$$A(\lambda) = -\log_{10}\frac{I}{I_0} \tag{6.2}$$

In chemistry 10 is always used as the base of the logarithm, but in other contexts sometimes e is used as base.

Given the simple equation, $A(\lambda)$ can be also easily calculated using the operators for spectra. This gives the correct answer, but in an object of class "source.scpt".

The conversion between $\tau(\lambda)$ and $A(\lambda)$ is:

$$A(\lambda) = -\log_{10}\tau(\lambda) \tag{6.3}$$

which in S language is:

```
my_T2A <- function(x) {-log10(x)}
```

The conversion between $A(\lambda)$ and $\tau(\lambda)$ is:

$$\tau(\lambda) = 10^{-A(\lambda)} \tag{6.4}$$

which in S language is:

```
my_A2T <- function(x) {10^-x}
```

Instead of these functions, the package defines generic functions and specialized functions, that can be used on vectors and on `filter.spc` objects. Then functions defined above could be directly applied to vectors but doing this on a column in a `filter.spc` is more cumbersome. As the spectra objects are data.tables, one can add a new column, say with transmittances to a copy of the filter data as follows.

```
my_gg400.spct <- copy(gg400.spct)
my_gg400.spct[ , A := T2A(Tfr)]

##      w.length   Tfr A
##   1:      200 1e-05 5
##   2:      210 1e-05 5
##   3:      220 1e-05 5
##   4:      230 1e-05 5
##   5:      240 1e-05 5
##  ---
## 176:     4950 1e-05 5
## 177:     5000 1e-05 5
## 178:     5050 1e-05 5
## 179:     5100 1e-05 5
## 180:     5150 1e-05 5

my_gg400.spct
```

```
##      w.length   Tfr A
##   1:      200 1e-05 5
##   2:      210 1e-05 5
##   3:      220 1e-05 5
##   4:      230 1e-05 5
##   5:      240 1e-05 5
##  ---
## 176:     4950 1e-05 5
## 177:     5000 1e-05 5
## 178:     5050 1e-05 5
## 179:     5100 1e-05 5
## 180:     5150 1e-05 5
```

6.4 Task: spectral absorbance from spectral transmittance

Using `filter.spct` objects, the calculations become very simple.

```
my_gg400.spct <- copy(gg400.spct)
T2A(my_gg400.spct)
a.gg400.spct <- T2A(my_gg400.spct, action="replace")
```

6.5 Task: spectral transmittance from spectral absorbance

```
A2T(a.gg400.spct)
A2T(a.gg400.spct, action="replace")
```

6.6 Task: reflected or transmitted spectrum from spectral reflectance and spectral irradiance

When we multiply a `source.spct` by a `filter.spct` or by a `reflector.spct` we obtain as a result a new `source.spct`.

```
class(sun.spct)

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

class(gg400.spct)

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

```
my_sun.spct <- copy(sun.spct)
my_gg400.spct <- copy(gg400.spct)
filtered_sun.spct <- sun.spct * gg400.spct
class(filtered_sun.spct)

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

```
head(filtered_sun.spct)

##    w.length    s.e.irrad
## 1:      293 2.609665e-11
## 2:      294 6.142401e-11
## 3:      295 2.176175e-10
## 4:      296 6.780119e-10
## 5:      297 1.533491e-09
## 6:      298 3.669677e-09
```

The result of the calculation can be directly used as an argument, for example, when calulating irradiance.

```
q_irrad_spct(sun.spct, UV()) * 1e6

##  UV.ISO
## 85.4784
## attr(,"time.unit")
## [1] "second"

q_irrad_spct(my_sun.spct, UV()) * 1e6

##  UV.ISO
## 85.4784
## attr(,"time.unit")
## [1] "second"

q_irrad_spct(filtered_sun.spct, UV()) * 1e6

##   UV.ISO
## 3.153016
## attr(,"time.unit")
## [1] "second"

q_irrad_spct(sun.spct * gg400.spct, UV()) * 1e6

##   UV.ISO
## 3.153016
## attr(,"time.unit")
## [1] "second"

q_irrad_spct(my_sun.spct * my_gg400.spct, UV()) * 1e6

##   UV.ISO
## 3.153016
## attr(,"time.unit")
## [1] "second"
```

```
q_irrad_spct(my_sun.spct * my_gg400.spct) * 1e6

## range.293.800
##      1135.601
## attr(,"time.unit")
## [1] "second"

q_irrad_spct(my_sun.spct * my_gg400.spct,
             new_waveband(min(sun.spct), max(sun.spct))) * 1e6
```

```
## range.293.800
##      1134.281
## attr(,"time.unit")
## [1] "second"
```

Remember, thet if we want to predict the output of a light source composed of different lamps or LEDs we can add the individual spectral irradiance, but using data measured from the target positions of each individaul light source. If we want then to add the effect of a filter we must multiply by the filter transmittance.

In the current version of package `photobiology` the operator is “chosen” based on the first operand. For this reason, when including a numeric operand, it should always be the second operand of binary operators for spectra.

```
# not working
my_luminaire <-
  (0.5 * Norlux_B.spct + Norlux_R.spct) *  PLX0A000_XT.spct
my_luminaire

## NULL

# works fine
my_luminaire <-
  (Norlux_B.spct * 0.5 + Norlux_R.spct) *  PLX0A000_XT.spct
my_luminaire

##       w.length s.e.irrad
##    1:   200.00         0
##    2:   200.47         0
##    3:   200.95         0
##    4:   201.00         0
##    5:   201.42         0
##   ---
## 2355:   936.05         0
## 2356:   936.48         0
## 2357:   936.91         0
## 2358:   937.00         0
## 2359:   937.34         0

q_ratio_spct(my_luminaire,
             list(Red(), Blue(), Green()), PAR())

##   Red.ISO:PAR(q:q)  Blue.ISO:PAR(q:q)
##        0.816195602        0.146121825
## Green.ISO:PAR(q:q)
##        0.003908976

q_irrad_spct(my_luminaire,
             list(PAR(), Red(), Blue(), Green())) *  1e6
```

```
##           PAR      Red.ISO     Blue.ISO
## 1.591314e-02 1.298824e-02 2.325257e-03
##    Green.ISO
## 6.220409e-05
## attr(,"time.unit")
## [1] "second"
```

6.7 Task: total spectral transmittance from internal spectral transmittance and spectral reflectance

6.8 Task: combined spectral transmittance of two or more filters

6.8.1 Ignoring reflectance

6.8.2 Considering reflectance

6.9 Task: light scattering media (natural waters, plant and animal tissues)

```
try(detach(package:photobiologyFilters))
try(detach(package:photobiologyLEDs))
try(detach(package:photobiologyWavebands))
try(detach(package:photobiology))
```

CHAPTER 7

Colour

Abstract

In this chapter we explain how to use colours according to visual sensitivity. For example calculating red-green-blue (RGB) values for humans.

7.1 Packages used in this chapter

For executing the examples listed in this chapter you need first to load the following packages from the library:

```
library(photobiology)
```

7.2 Introduction

The calculation of equivalent colours and colour spaces is based on the number of photoreceptors and their spectral sensitivities. For humans it is normally accepted that there are three photoreceptors in the eyes, with maximum sensitivities in the red, green, and blue regions of the spectrum.

When calculating colours we can take either only the colour or both colour and apparent luminance. In our functions, in the first case one needs to provide as input 'chromaticity coordinates' (CC) and in the second case 'colour matching functions' (CMF). The suite includes data for humans, but the current implementation of the functions should be able to handle also calculations for other organisms with tri-chromic vision.

The 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 wavelengths, based on the assumption of a flat energy irradiance across this 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. The range of wavelengths used in the calculations is that in the chromaticity data.

One use of these functions is to generate realistic colour for 'key' on plots of spectral data. Other uses are also possible, like simulating how different, different objects would look to a certain organism.

> This package is very 'young' so may be to some extent buggy, and/or have rough edges. We plan to add at least visual data for honey bees.

7.3 Task: calculating an RGB colour from a single wavelength

Function `w_length2rgb` must be used in this case. If a vector of wavelengths is supplied as argument, then a vector of `colors`, of the same length, is returned. Here are some examples of calculation of R color definitions 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"
```

7.4 Task: calculating an RGB colour for a range of wavelengths

Function `w_length_range2rgb` must be used in this case. This function expects as input a vector of two number, as returned by the function `range`.

If a longer vector is supplied as argument, its range is used, with a warning. If a vector of lengths one is given as argument, then the same output as from function `w_length2rgb` is returned. This function assumes a flat energy spectral irradiance curve within the range. Some examples: 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.data$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"
```

7.5 Task: calculating an RGB colour for spectrum

Function `s_e_irrad2rgb` in contrast to those described above, when calculating the color takes into account the spectral irradiance.

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.spct))

## [1] "#544F4B"

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

## [1] "#59534F"

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

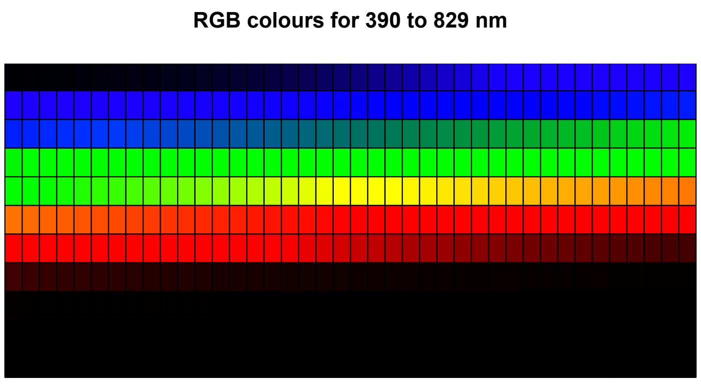
```
## [1] "#B63C37"

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

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

Except for the first example, we specificity the visual sensitivity data to use.

7.6 A sample of colours

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)])
}
```



```r
try(detach(package:photobiology))
```

CHAPTER 8

Plotting spectra and colours

Abstract

In this chapter we explain how to plot spectra and colours, using packages ggplot2, ggtern, and the functions in our package photobiologygg. Both ggtern for ternary plots and photobiologygg for annotating spectra build new functionality on top of the ggplot2 package. We also use several functions and data from package photobiology in the examples.

8.1 Packages used in this chapter

For executing the examples listed in this chapter you need first to load the following packages from the library:

```
library(ggplot2)
library(scales)
library(ggtern)

##
## Attaching package:  'ggtern'
##
## The following objects are masked from 'package:ggplot2':
##
##     %+%, %+replace%, aes, calc_element,
##     geom_density2d, geom_segment,
##     geom_smooth, ggplot_build,
##     ggplot_gtable, ggsave, opts,
##     stat_density2d, stat_smooth, theme,
##     theme_bw, theme_classic, theme_get,
##     theme_gray, theme_grey, theme_minimal,
##     theme_set, theme_update

library(gridExtra)
```

```
## Loading required package:  grid
```

```r
library(photobiology)
library(photobiologyFilters)
library(photobiologyWavebands)
library(photobiologygg)
```

8.2 Introduction to plotting spectra

We show in this chapter examples of how spectral data can be plotted. All the examples are done with package `ggplot2`, sometimes using in addition other packages. `ggplot2` provides the most recent, but stable, type of plotting functionality in R, and is what we use here for most examples. Both `base` graphic functions, part of R itself and ‘trellis’ graphics provided by package `lattice` are other popular alternatives. The new package `ggvis` uses similar grammar as `ggplot2` but drastically improves on functionality for interactive plots. Several of the functions used in this chapter are extensions to package `ggplot2`[1]

How to depict a spectrum in a figure has to be thought in relation to what aspect of the information we want to highlight. A line plot of a spectrum with peaks and/or valleys labelled highlights the shape of the spectrum, while a spectrum plotted with the area below the curve filled highlights the total energy irradiance (or photon irradiance) for a given region of the spectrum. Adding a bar with the colours corresponding to the different wavelengths, facilitates the reading of the plot for people not familiar with the interpretation on wavelengths expressed in nanometres. Labeling regions of the spectrum with waveband names also facilitates the understanding of plotted spectral data. A basic line plot of spectral data can be easily done with `ggplot2` or any of the other plotting functions in R. In this chapter we focus on how to add to basic line and dot plots all the ‘fancy decorations’ that can so much facilitate their reading and interpretation.

Towards the end of the chapter we give examples of plotting of RGB (red-green-blue) colours for human vision on a ternary plot, and show how to do a ternary plot for GBU (green-blue-ultraviolet) flower colours for honeybee vision using as reference the reflectance of a background.

If you are not familiar with `ggplot2` and `ggtern` plotting, please read Appendix **??** on page **??** before continuing reading the present chapter.

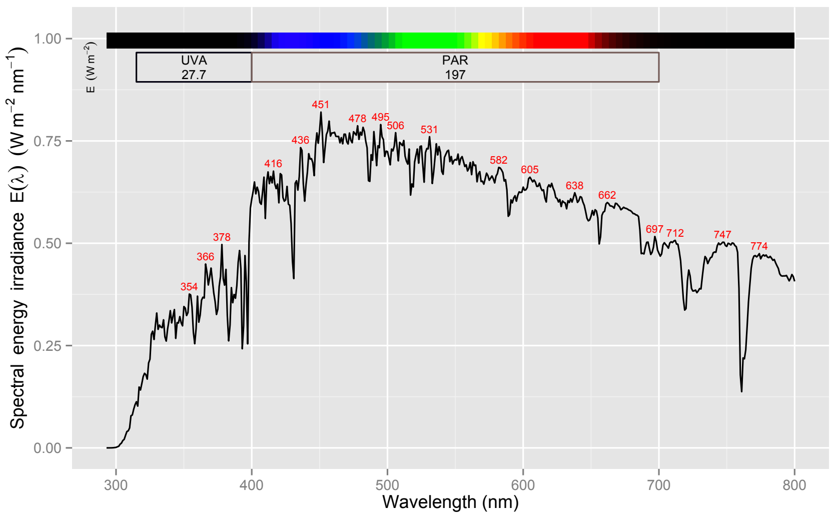
8.3 Task: simple plotting of spectra

Pakage `photobiologygg` defines specializations of the generic `plot` function of R. These functions are available for spectral objects. They return a

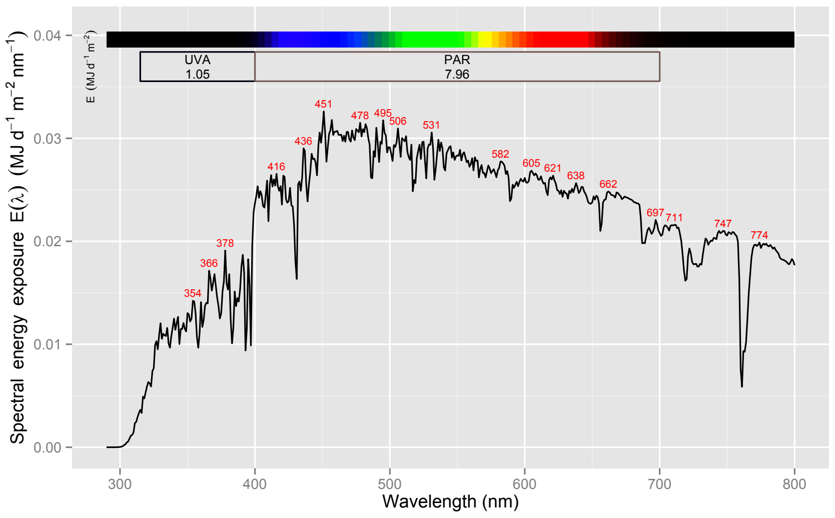
[1] `ggplot2` is feature-frozen, in other words the user interface defined by the functions and their arguments will not change in future versions. Consequently it is a good basis for adding application-specific functionality through separate packages. `ggplot2` uses the *grammar of graphics* for describing the plots. This grammar, because it is consistent, tends to be easier to understand, and makes it easier to design new functionality that uses extensions based on the same ‘language grammar’ as used by the original package.

`ggplot` object, to which additional layers can be added if desired. An example of it simplest use follows. As the spectral objects have spectral irradiance expressed in known energy or photon units, and an attribute indicating the time unit, the axis labels are produced automatically. The two plots that follow show spectral irradiance, and spectral daily exposure, respectively.

```
plot(sun.spct)
```

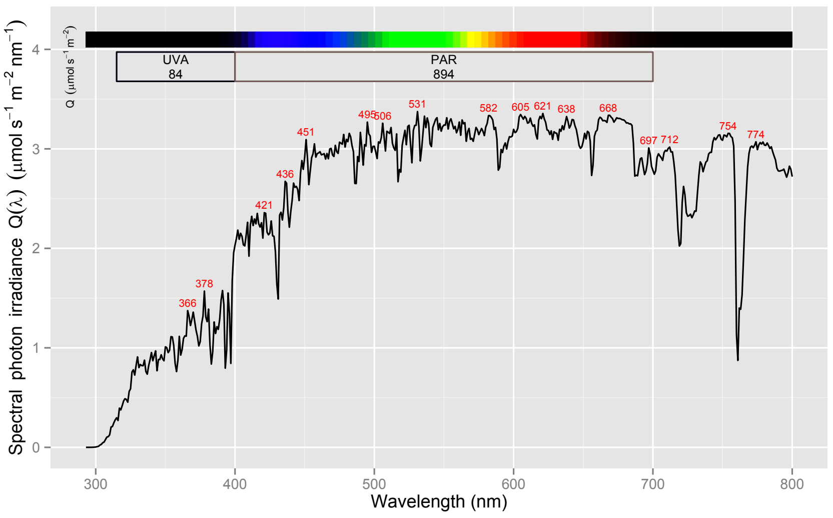


```
plot(sun.daily.spct)
```



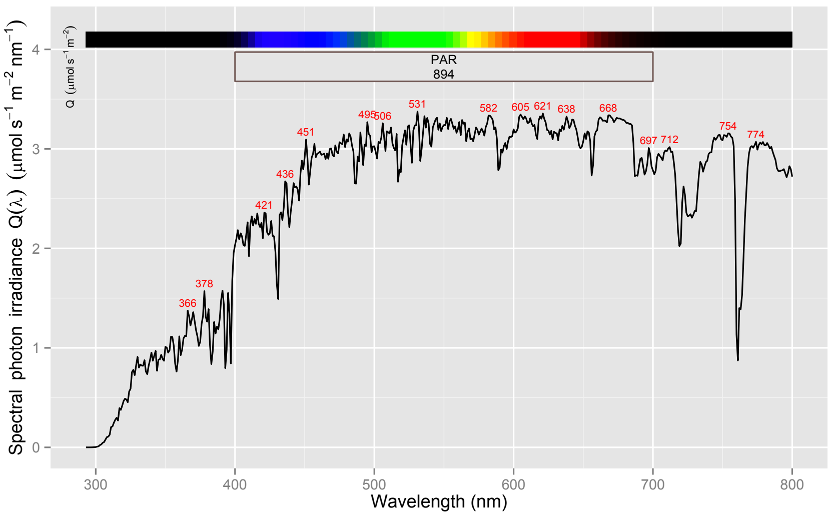
The parameter `unit` can be set to `"photon"` to obtain a plot depicting spectral photon irradiance. This works irrespective of whether the `source.spct` object contains the spectral data in photon or energy units.

```
plot(sun.spct, unit="photon")
```

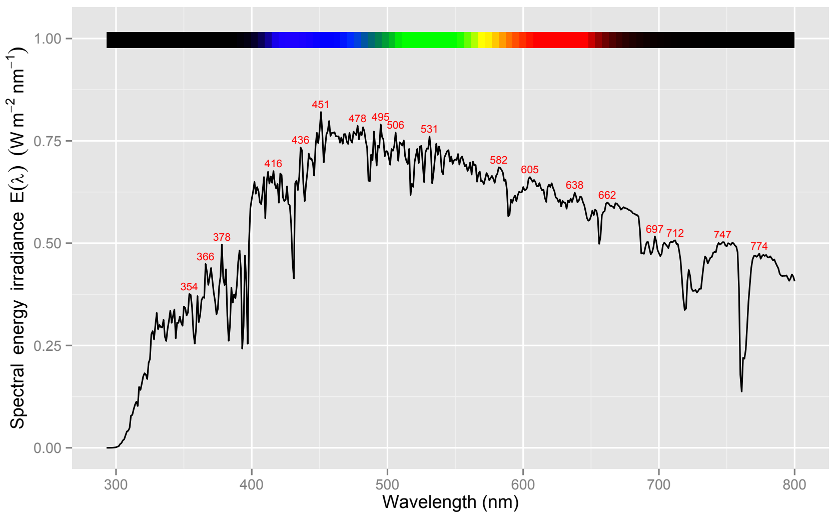


A list of wave bands, or a single wave band, to be used for annotation can be supplied through the `bands` parameter. A NULL waveband results in no waveband labels, while the next example shows how to obtain the total irradiance.

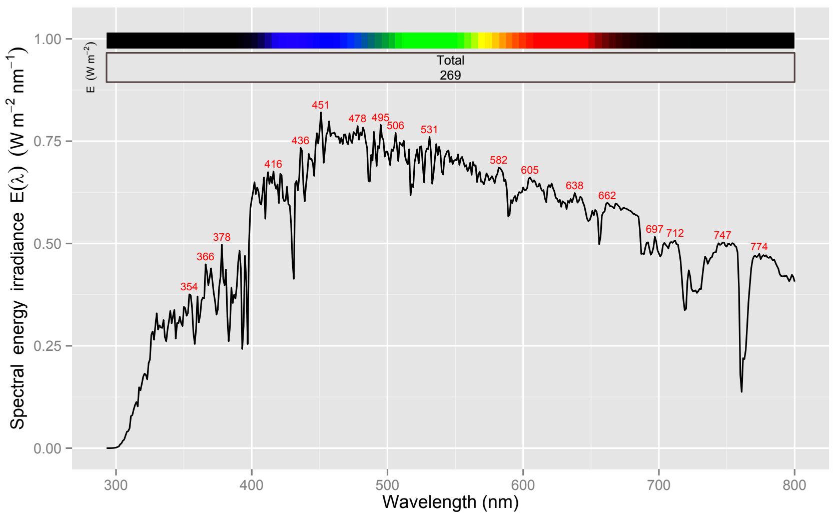
```
plot(sun.spct, bands=PAR(), unit="photon")
```



```
plot(sun.spct, bands=NULL)
```

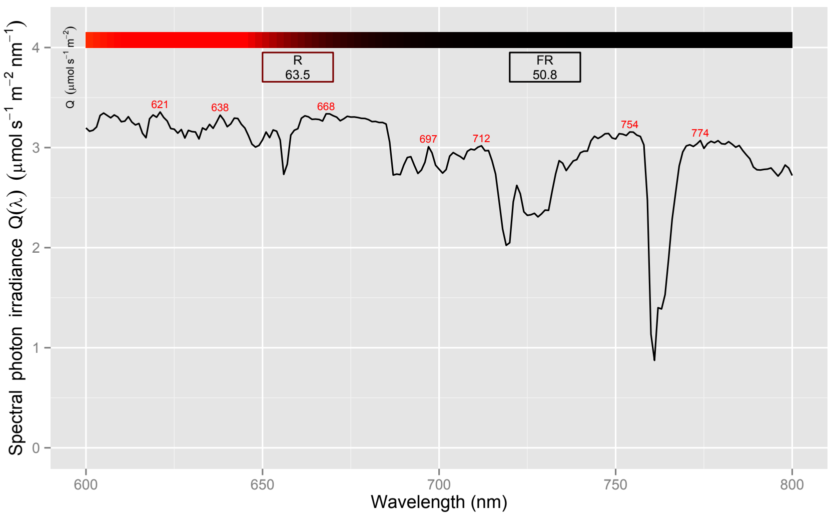


```
plot(sun.spct, bands=waveband(sun.spct))
```



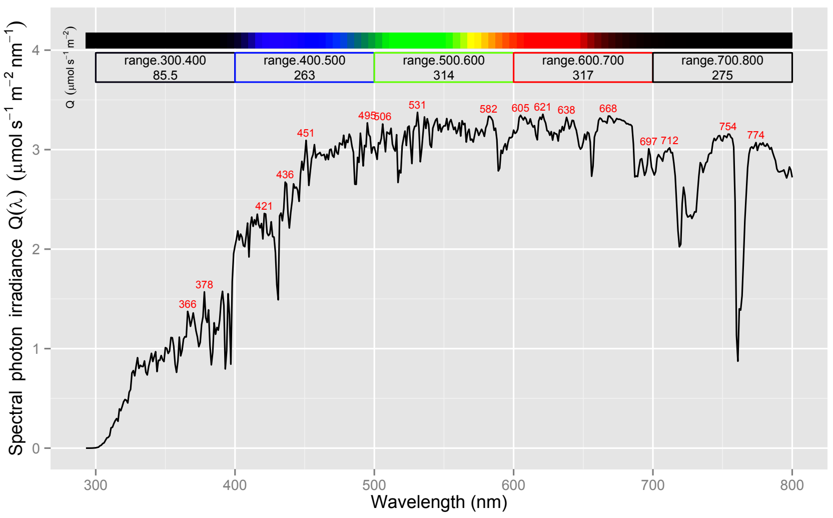
Of course the arguments to these parameters can be supplied in different combinations, and combined with other functions as need. This last example shows how to plot using photon-based units, selecting only a specific region of the spectrum, annotated with the red and far-red photon irradiances, using Prof. Harry Smith's definitions for these two wavebands.

```
plot(trim_spct(sun.spct, waveband(c(600,800))),
     bands=list(Red("Smith"), Far_red("Smith")), unit="photon")
```

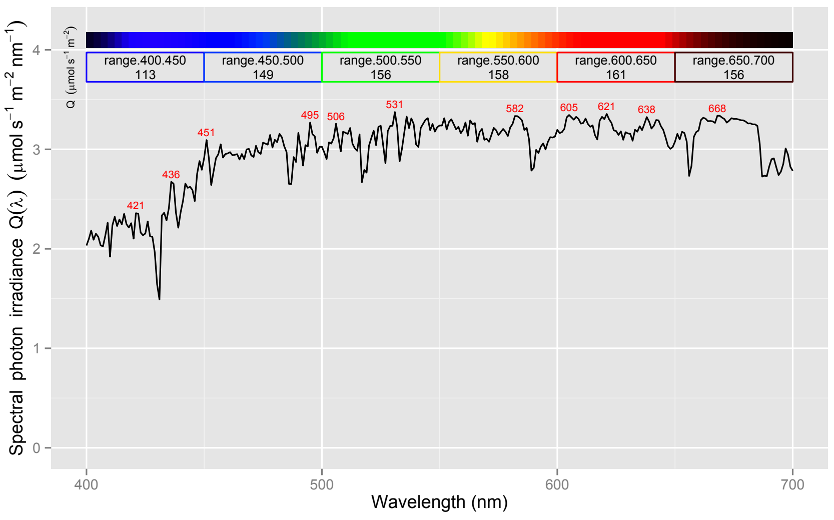


Two final examples show how to annotate a spectrum plot by equal sized wavebands.

```r
plot(sun.spct,
     bands=split_bands(c(300,800), length.out=5), unit="photon")
```



```r
plot(trim_spct(sun.spct, PAR()),
     bands=split_bands(PAR(), length.out=6), unit="photon")
```



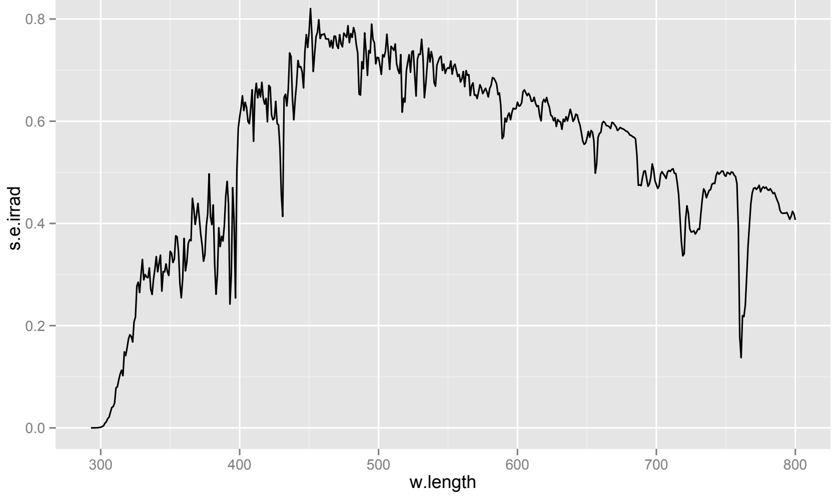
As the current implementation uses annotations rather than a `ggplot` ‘statistic’, waveband irradiance annotations ignore global aesthetics and facets. If used for simultaneous plotting of several spectra (stored in a single R object), then parameter `bands` should given NULL as argument.

8.4 Task: plotting spectra with `ggplot2`

We create a simple line plot, assign it to a variable called `fig_sun.e0` and then on the next line `print` it[2]. We obtain a plot with the axis labeled with the names of the variables, which is enough to check the data, but not good enough for publication.

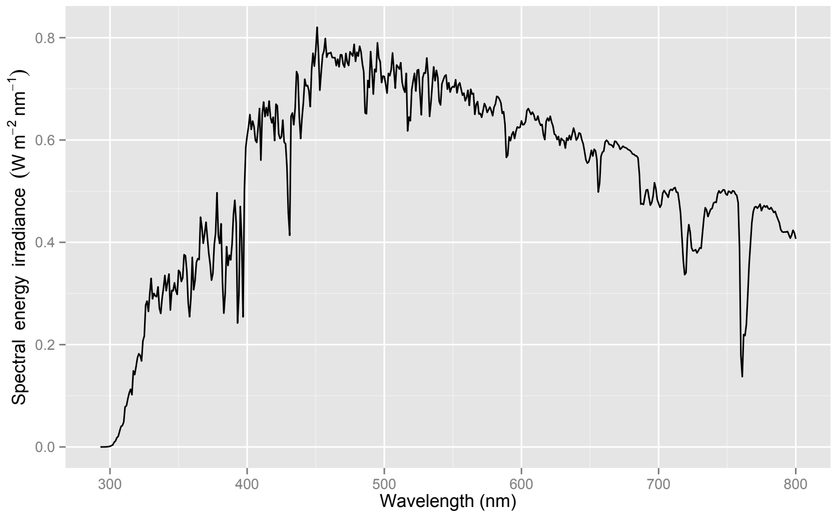
```
fig_sun.e0 <-
  ggplot(data=sun.spct, aes(x=w.length, y=s.e.irrad)) +
  geom_line()

fig_sun.e0
```

[2]we could have used `print(fig_sun.e0)` explicitly, but this is needed only in scripts because printing takes places automatically when working at the R console.



Next we add `labs` to obtain nicer axis labels, instead of assigning the result to a variable for reuse, we print it on-the-fly. As we need superscripts for the y-label we have to use `expression` instead of a character string as we use for the x-label. The syntax of expressions is complex, so please look at `help(plotmath)` and appendix **??** for more details.

```
fig_sun.e0  +
  labs(
    y = expression(Spectral~~energy~~irradiance~~(W~m^{-2}~nm^{-1})),
    x = "Wavelength (nm)")
```



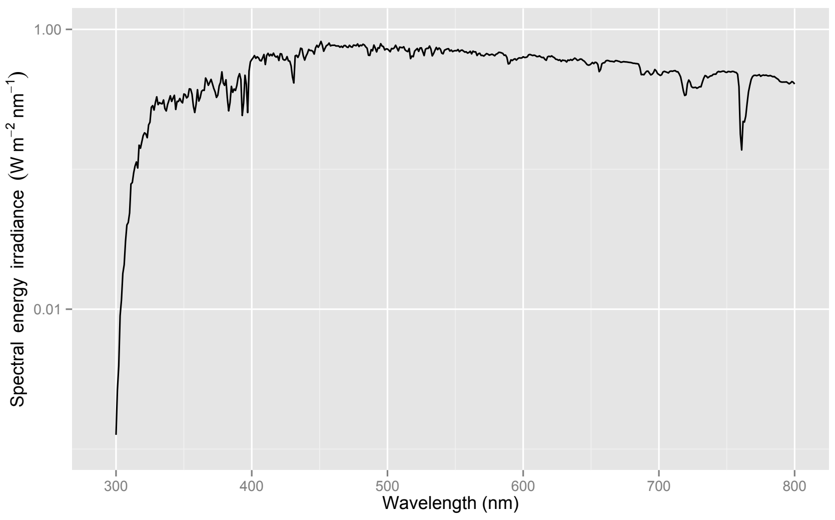
As we are going to re-use the same axis-labels in later plots, it is handy to save their definitions to variables. These definitions will be used in many of this chapter's plots. We also add `atop` to two of the expressions to making shorter versions by setting the spectral irradiance units on a second line in the axis labels.

```
ylab_watt <-
  expression(Spectral~~energy~~irradiance~~(W~m^{-2}~nm^{-1}))
ylab_watt_atop <-
  expression(atop(Spectral~~energy~~irradiance,
                  (W~m^{-2}~nm^{-1})))
ylab_umol <-
  expression(Spectral~~photon~~irradiance~~(mu*mol~m^{-2}~s^{-1}~nm^{-1}))
ylab_umol_atop <-
  expression(atop(Spectral~~photon~~irradiance,
                  (mu*mol~m^{-2}~s^{-1}~nm^{-1})))
xlab_nm <- "Wavelength (nm)"
```

8.5 Task: using a log scale

Here without need to recreate the figure, we add a logarithmic scale for the y-axis and print on the fly the result, and two of the just saved axis-labels. In this case we override the automatic limits of the scale. We do not give further examples of this, but could be also used with later examples, just by adjusting the values used as scale limits.

```
fig_sun.e0 +
  scale_y_log10(limits=c(1e-3, 1e0)) +
  labs(x = xlab_nm, y = ylab_watt)

## Warning:  Removed 7 rows containing missing values
(geom_path).
```



The code above generates some harmless warnings, which are due some y values not being valid input for `log10`, the function used for the re-scaling, or because they fall outside the scale limits.

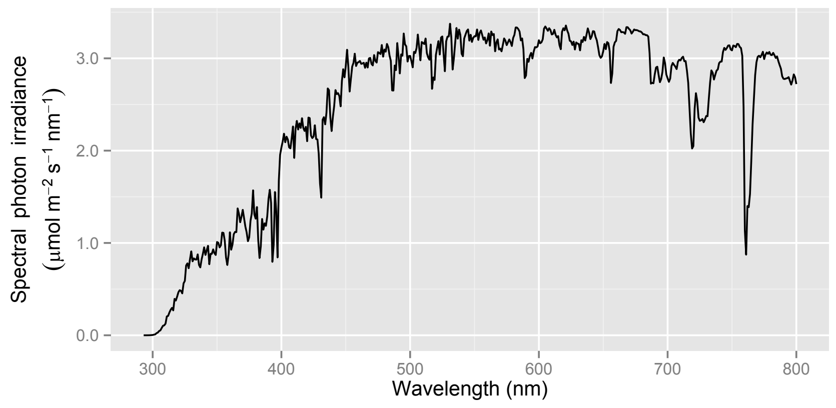
8.6 Task: compare energy and photon spectral units

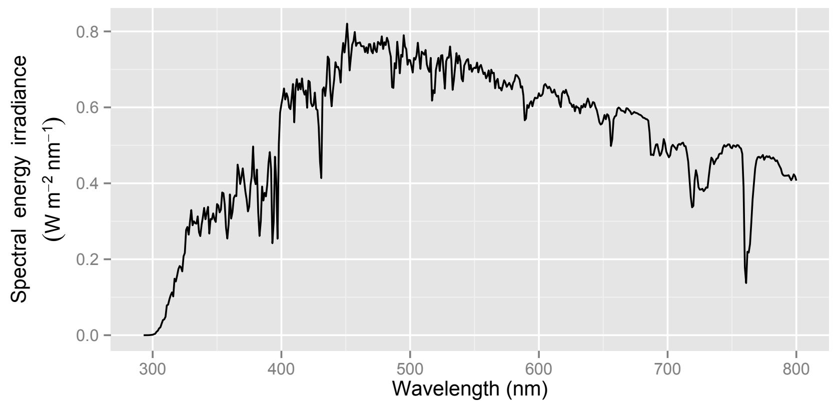
We use once more the axis-labels saved above, but this time use the two-line label for the y-axis. To make sure that the width of the plotting area of both plots is the same, we need to have tick labels of the same width and format in both plots. For this we define a formatting function `num_one_dec` and then use it in the scale definition.

```
num_one_dec <- function(x, ...) {
  format(x, nsmall=1, trim=FALSE, width=4, ...)
  }

fig_sun.q <-
  ggplot(data=sun.spct, aes(x=w.length, y=s.q.irrad * 1e6)) +
  geom_line() +
  scale_y_continuous(labels = num_one_dec) +
  labs(x = xlab_nm)

fig_sun.e1 <-
  ggplot(data=sun.spct, aes(x=w.length, y=s.e.irrad)) +
  geom_line() +
  scale_y_continuous(labels = num_one_dec) +
  labs(x = xlab_nm)
```

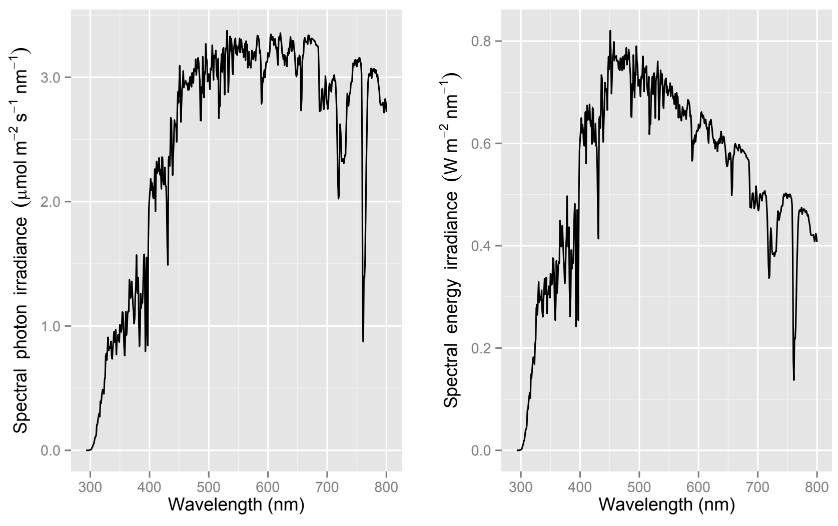
We can use function `grid.arrange` to make a single plot from two separate ggplots, and put them side by or on top of each other. We use different y-axis labels in the two cases to make better use of the available space.

```
grid.arrange(fig_sun.q  + labs(y = ylab_umol_atop),
             fig_sun.e1 + labs(y = ylab_watt_atop),
             nrow=2)
```





```
grid.arrange(fig_sun.q  + labs(y = ylab_umol),
             fig_sun.e1 + labs(y = ylab_watt),
             nrow=1)
```



8.7 Task: finding peaks and valleys in spectra

We first show the use of function `get_peaks` that returns the wavelengths at which peaks are located. The parameter `span` determines the number of values used to find a local maximum (the higher the value used, the fewer maxima are detected), and the parameter `ignore_threshold` the fraction of the total span along the irradiance that is taken into account (a value of 0.75, requests only peaks in the upper 25% of the y-range to be returned; a value of -0.75 works similarly but for the lower half of the y-range)[3]. It is good to mention that `head` returns the first six rows of its argument, and we use it here just to reduce the length of the output, if you run these examples yourself, you can remove `head` from the code. In the output, x corresponds to wavelength, and y to spectral irradiance, while `label` is a character string with the wavelength, possibly formatted.

```
head(with(sun.spct,
          get_peaks(w.length, s.e.irrad, span=31)))

##     x         y label
## 1 378 0.4969714   378
## 2 416 0.6761818   416
## 3 451 0.8204633   451
## 4 478 0.7869773   478
## 5 495 0.7899872   495
## 6 531 0.7603297   531

head(with(sun.spct,
          get_peaks(w.length, s.e.irrad, span=31,
                    ignore_threshold=0.75)))

##     x         y label
## 1 416 0.6761818   416
## 2 451 0.8204633   451
## 3 478 0.7869773   478
## 4 495 0.7899872   495
## 5 531 0.7603297   531
## 6 582 0.6853736   582
```

The parameter span, indicates the size in number of observations (e.g. number of discrete wavelength values) included in the window used to find local maxima (peaks) or minima (valleys). By providing different values for this argument we can 'adjust' how *fine* or *coarse* is the structure described by the peaks returned by the function. The window is always defined using an odd number of observations, if an even number is provided as argument, it is increased by one, with a warning.

```
head(with(sun.spct,
          get_peaks(w.length, s.e.irrad, span=21)))

##     x         y label
```

[3]In the current example setting `ignore_threshold` equal to 0.75 given that the range of the spectral irradiance data goes from 0.00 μmol m^{-2} s^{-1} nm^{-1} to 0.82 μmol m^{-2} s^{-1} nm^{-1}, causes any peaks having a spectral irradiance of less than 0.62 μmol m^{-2} s^{-1} nm^{-1} to be ignored.

```
## 1 354 0.3758625   354
## 2 366 0.4491898   366
## 3 378 0.4969714   378
## 4 416 0.6761818   416
## 5 436 0.7336607   436
## 6 451 0.8204633   451

head(with(sun.spct,
          get_peaks(w.length, s.e.irrad, span=51)))

##     x         y label
## 1 451 0.8204633   451
## 2 495 0.7899872   495
## 3 747 0.5025733   747
```

The equivalent function for finding valleys is `get_valleys` taking the same parameters as `get_peaks` but returning the wavelengths at which the valleys are located.

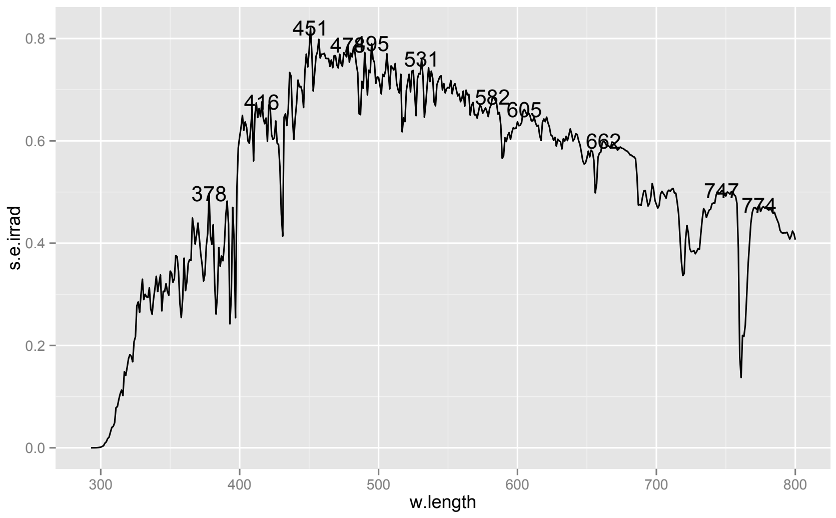
```
head(with(sun.spct,
          get_valleys(w.length, s.e.irrad, span=51)))

##     x         y label
## 1 358 0.2544907   358
## 2 393 0.2422023   393
## 3 431 0.4136900   431
## 4 487 0.6511654   487
## 5 517 0.6176652   517
## 6 589 0.5658760   589

head(with(sun.spct,
          get_valleys(w.length, s.e.irrad, span=51,
                      ignore_threshold=0.5)))

##     x         y label
## 1 431 0.4136900   431
## 2 487 0.6511654   487
## 3 517 0.6176652   517
## 4 589 0.5658760   589
## 5 656 0.4982959   656
```

In the next section, we plot spectra and annotate them with peaks and valleys. If you find the meaning of the parameters `span` and `ignore_threshold` difficult to grasp from the explanation given above, please, study the code and plots in section 8.8.

8.8 Task: annotating peaks and valleys in spectra

Here we show an example of the use the new `ggplot` 'statistics' `stat_peaks` from our package `photobiologygg`. It uses the same parameter names and take the same arguments as the `get_peaks` function described in section 8.7. We reuse once more `fig_sun.e` saved in section 8.4.

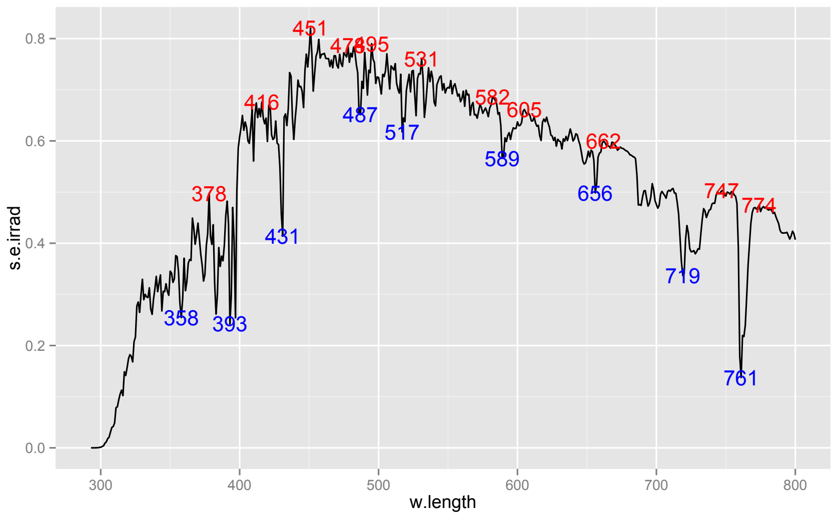
```
fig_sun.e0 + stat_peaks(span=31)
```



Now we play with ggplot2 to show different ways of plotting the peaks and valleys. It behaves as a ggplot2 stat_xxxx function accepting a geom argument and all the aesthetics valid for the chosen geom. By default geom_text is used.

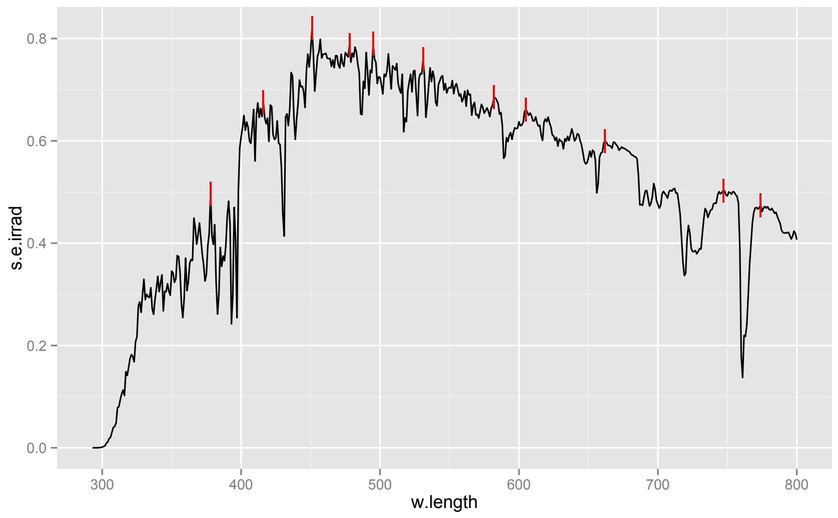
We can change aesthetics, for example the colour:

```
fig_sun.e0 + stat_peaks(colour="red", span=31) +
             stat_valleys(colour="blue", span=51)
```



We can also use a different geom, in this case geom_point, however, be aware that the geom parameter takes as argument a character string giving the name of the geom, in this case "point". We change a few additional aesthetics of the points: we set shape to a character, and set its size to 6.

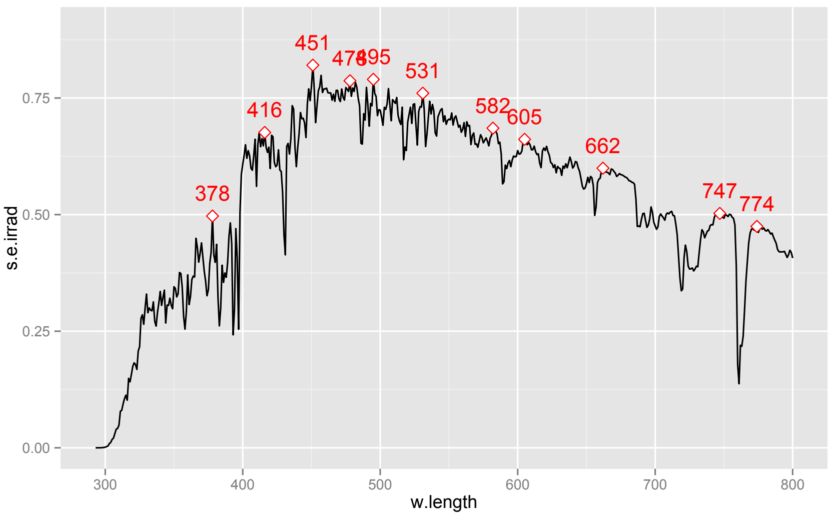
```
fig_sun.e0 +
  stat_peaks(colour="red", geom="point",
             shape="|", size=6, span=31)
```



We can add the same `stat` two or more times to a ggplot, in this example, each time with a different `geom`. First we add points to mark the peaks, and afterwards add labels showing the wavelengths at which they are located using geom `"text"`. For the `shape`, or type of symbol, we use one that supports ‘fill', and set the `fill` to `"white"` but keep the border of the symbol `"red"` by setting `colour`, we also change the `size`. With the labels we use `vjust` to ‘justify' the text moving the labels vertically, so that they do not overlap the line depicting the spectrum[4] In addition we expand the y-axis scale so that all labels fall within the plotting area.

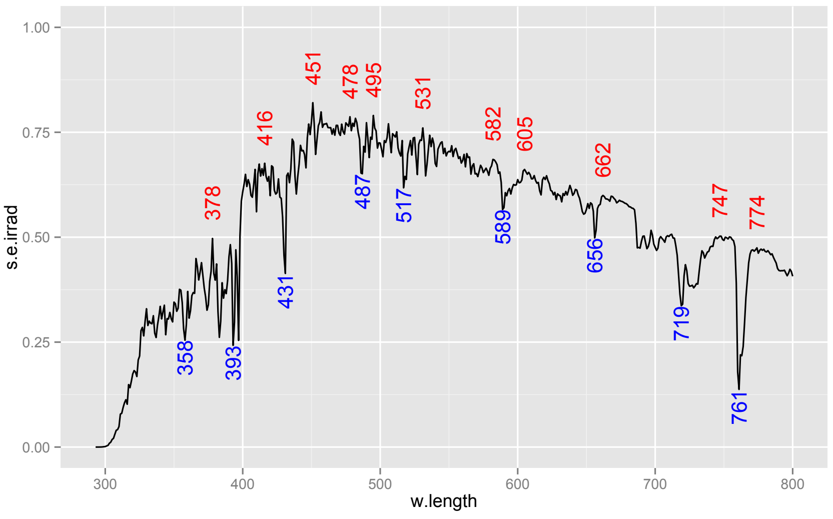
```
fig_sun.e0 +
  stat_peaks(colour="red", geom="point", shape=23,
             fill="white", size=3, span=31) +
  stat_peaks(colour="red", vjust=-1, span=31) +
  expand_limits(y=0.9)
```

[4]The default position of labels is to have them centred on the coordinates of the peak or valley. Unless we rotate the label, `vjust` can be used to shift the label along the y-axis, however, justification is a property of the text, not the plot, so the vertical direction is referenced to the position of the text of the label. A value of 0.5 indicates centering, a negative value ‘up' and a positive value ‘down'. For example a value of -1 puts the x, y coordinates of the peak or valley at the lower edge of the ‘bounding box' of the text. For `hjust` values of -1 and 1 right and left justify the label with respect to the x, y coordinates supplied. Values other than -1, 0.5, and 1, are valid input, but are rather tricky to use for `hjust` as the displacement is computed relative to the width of the bounding box of the label, the displacement being different for the same numerical value depending on the length of the label text.



Finally an example with rotated labels, using different colours for peaks and valleys. Be aware that the 'justification' direction, as discussed in the footnote, is referenced to the position of the text, and for this reason to move the rotated labels upwards we need to use `hjust` as the desired displacement is horizontal with respect to the orientation of the text of the label. As we put peak labels above the spectrum and valleys bellow it, we need to use `hjust` values of opposite sign, but the exact values used were simply adjusted by trial and error until the figure looked as desired.

```
fig_sun.e0 +
  stat_peaks(angle=90, hjust=-0.5, colour="red", span=31) +
  stat_valleys(angle=90, hjust=1, color="blue", span=51) +
  expand_limits(y=1.0)
```



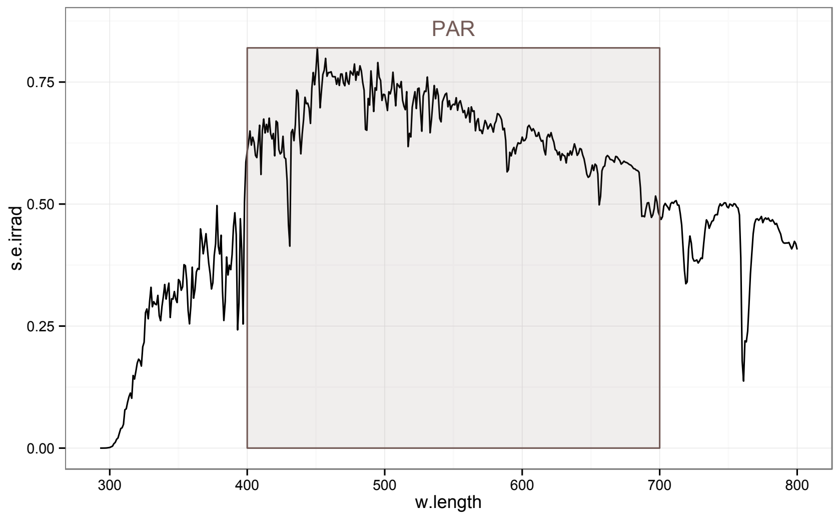
See section **??** in chapter **??** for an example these stats together with facets.

8.9 Task: annotating wavebands

The function `annotate_waveband` can be used to highlight a waveband in a plot of spectral data. Its first argument should be a `waveband` object, and the second argument a `geom` as a character string. The positions on the x-axis are calculated automatically by default, but they can be overridden by explicit arguments. The vertical positions have no default, except for `ymin` which is equal to zero by default. The colour has a default value calculated from waveband definition, in addition x is by default set to the midpoint of the waveband along the wavelength limits. The default value of the labels is the 'name' of the waveband as returned by `labels.waveband`.

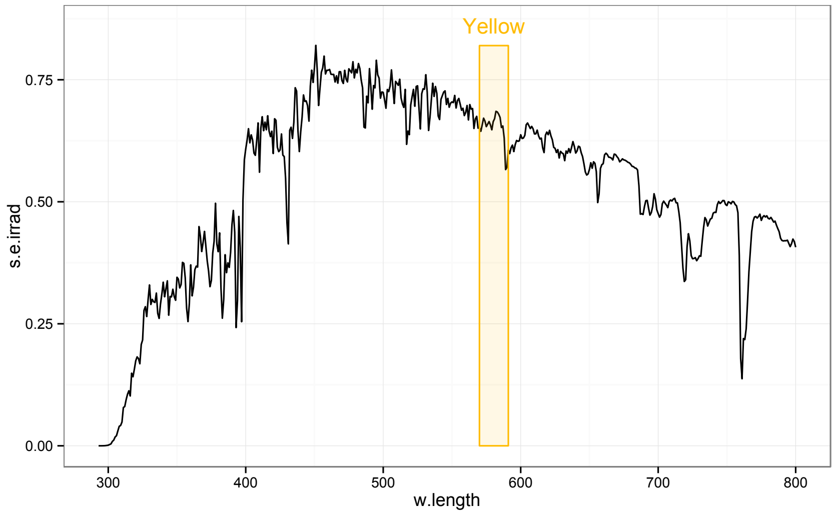
Here is an example for PAR using defaults, and with arguments supplied only for parameters with no defaults. The example does the annotation using two different 'geoms', `"rect"` for marking the region, and `"text"` for the labels.

```
figvl <- fig_sun.e0 + annotate_waveband(PAR(), "rect", ymax=0.82) +
                     annotate_waveband(PAR(), "text", y=0.86)

figvl + theme_bw()
```



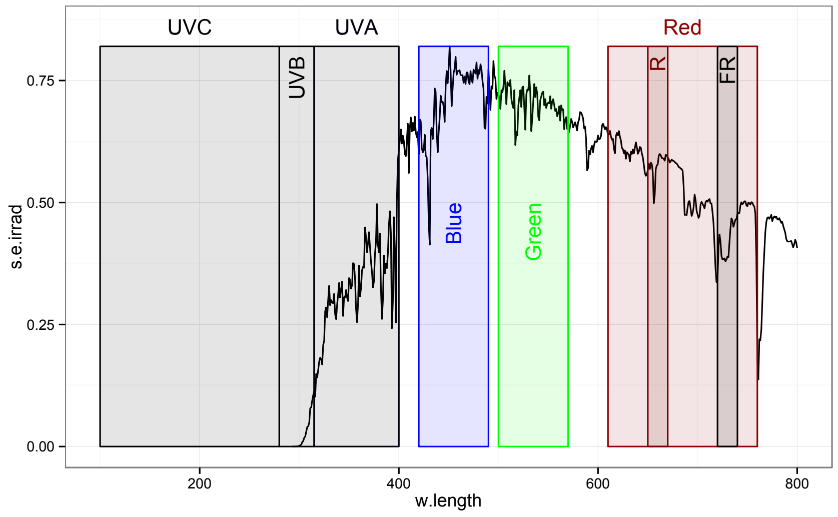
This example annotates a narrow waveband.

```
figvl <- fig_sun.e0 + annotate_waveband(Yellow(), "rect", ymax=0.82) +
                     annotate_waveband(Yellow(), "text", y=0.86)

figvl + theme_bw()
```



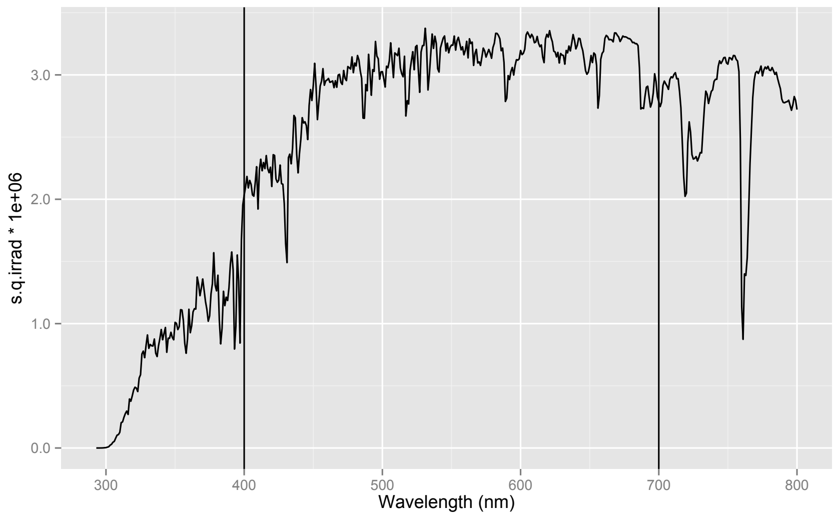
Now an example that is more complex, and demonstrates the flexibility of plots produced with ggplot2. We add annotations for eight different wavebands, some of them overlapping. For each one we use two 'geoms' and some labels are rotated and justified. We can also see in this example that the annotations look nicer on a white background, which can be obtained with theme_bw. A much simpler, but less flexible approach for adding annotations for several wavebands is described on page 118.

```
figv2 <- fig_sun.e0 +
  annotate_waveband(UVC(), "rect",
                    ymax=0.82) +
  annotate_waveband(UVC(), "text",
                    y=0.86) +
  annotate_waveband(UVB(), "rect",
                    ymax=0.82) +
  annotate_waveband(UVB(), "text",
                    y=0.80, angle=90, hjust=1) +
  annotate_waveband(UVA(), "rect",
                    ymax=0.82) +
  annotate_waveband(UVA(), "text",
                    y=0.86) +
  annotate_waveband(Blue("Sellaro"), "rect",
                    ymax=0.82) +
  annotate_waveband(Blue("Sellaro"), "text",
                    y=0.5, angle=90, hjust=1) +
  annotate_waveband(Green("Sellaro"), "rect",
                    ymax=0.82) +
  annotate_waveband(Green("Sellaro"), "text",
                    y=0.50, angle=90, hjust=1) +
  annotate_waveband(Red(), "rect",
                    ymax=0.82) +
  annotate_waveband(Red(), "text",
                    y=0.86) +
  annotate_waveband(Red("Smith"), "rect",
                    ymax=0.82) +
  annotate_waveband(Red("Smith"), "text",
                    y=0.80, angle=90, hjust=1) +
  annotate_waveband(Far_red("Smith"), "rect",
                    ymax=0.82) +
  annotate_waveband(Far_red("Smith"), "text",
                    y=0.80, angle=90, hjust=1)

figv2 + theme_bw()
```



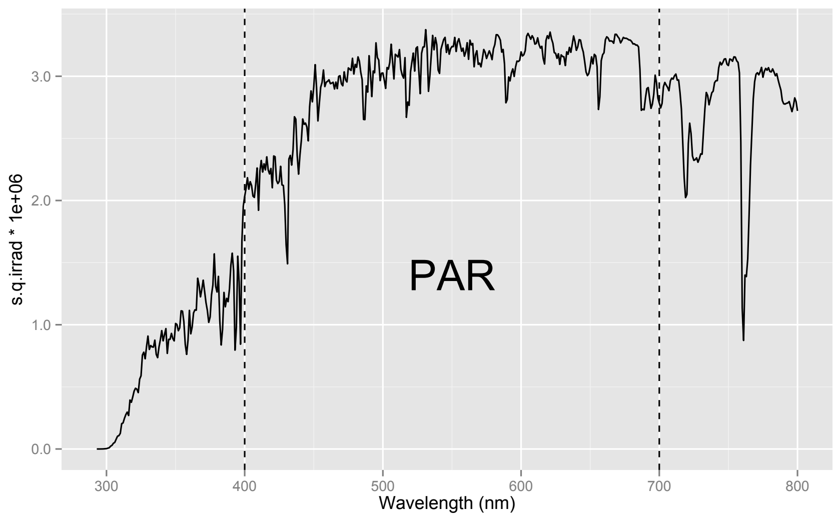
A simple example using `geom_vline`:

```
figvl3 <- fig_sun.q +
  geom_vline(xintercept=range(PAR()))

figvl3
```



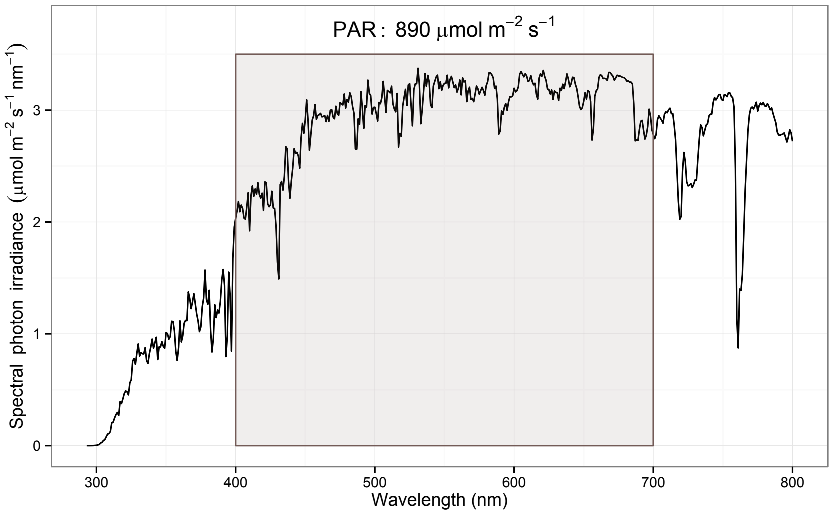
And one where we change some of the aesthetics, and add a label:

```
figvl4 <- fig_sun.q +
  geom_vline(xintercept=range(PAR()), linetype="dashed") +
  annotate_waveband(PAR(), "text", y=1.4, size=10, colour="black")

figvl4
```



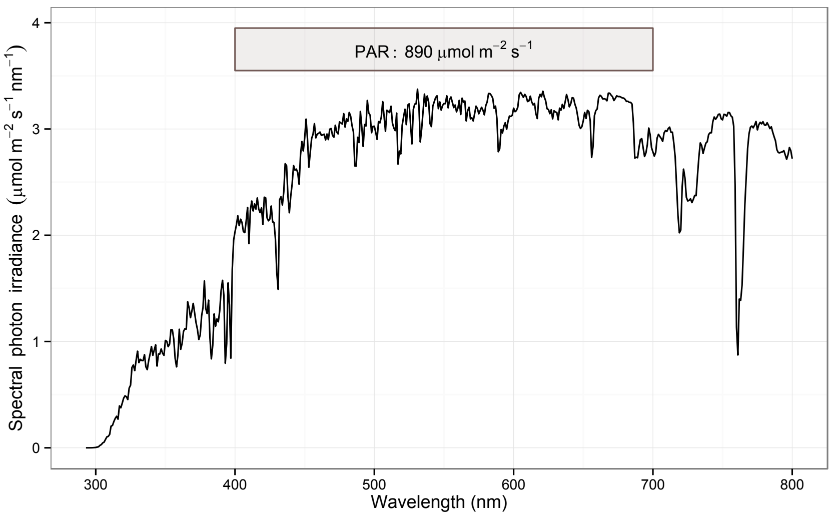
Now including calculated values in the label, first with a simple example with only PAR. Because of using expressions to obtain superscripts we need to add `parse=TRUE` to the call. In addition as we are expressing the integral in photon based units, we also change the type of units used for plotting the spectral irradiance (multiplying by $1 \cdot 10^6$ to because of the unit multiplier used).

```
fig_sun <- ggplot(data=sun.spct,
                  aes(x=w.length, y=s.q.irrad * 1e6)) +
  geom_line() +
  labs(y = ylab_umol,
       x = "Wavelength (nm)")

par <- q_irrad(sun.spct, PAR()) * 1e6

fig_sun2 <- fig_sun +
  annotate_waveband(PAR(), "rect", ymax=3.5) +
  annotate_waveband(PAR(), "text",
                    label=paste("PAR:~", signif(par,digits=2),
                                "*~mu*mol~m^{-2}~s^{-1}", sep=""),
                    y=3.75, colour="black", parse=TRUE)

fig_sun2 + theme_bw()
```



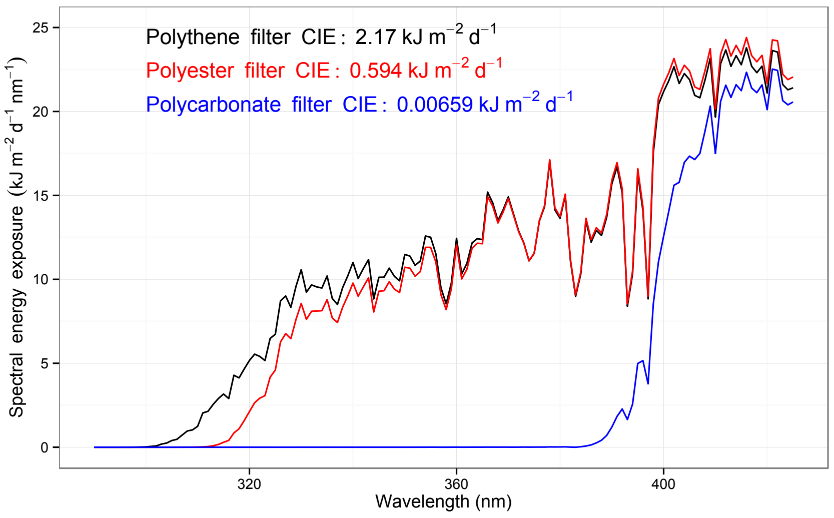
A variation of the previous figure shows how to use smaller rectangles for annotation, which yields plots where the spectrum itself is easier to see than when the rectangle overlaps the spectrum. We achieve this by supplying as argument both `ymax` and `ymin`, and slightly reducing the size of the text with `size = 4`.

```
fig_sun <- ggplot(data=sun.spct,
                  aes(x=w.length, y=s.q.irrad * 1e6)) +
  geom_line() +
  labs(y = ylab_umol,
       x = "Wavelength (nm)")

par <- q_irrad(sun.spct, PAR()) * 1e6

fig_sun2 <- fig_sun +
  annotate_waveband(PAR(), "rect", ymax=3.95, ymin=3.55) +
  annotate_waveband(PAR(), "text", size=4,
                    label=paste("PAR:~", signif(par,digits=2),
                                "*~mu*mol~m^{-2}~s^{-1}", sep=""),
                    y=3.75, colour="black", parse=TRUE)

fig_sun2 + theme_bw()
```



This type of annotations can be also easily done for effective exposures or doses, but in this example as we position the annotations manually, we can use ggplot2's 'normal' `annotate` function. We use `xlim` to restrict the plotted region of the spectrum to the range of wavelengths of interest.

```
fig_dsun <-
  ggplot(data=sun.daily.spct * polythene.new.spct,
         aes(x=w.length, y=s.e.irrad * 1e-3)) + geom_line() +
  geom_line(data=sun.daily.spct * polyester.new.spct,
            colour="red") +
  geom_line(data=sun.daily.spct * PC.spct,
            colour="blue") +
  labs(y =
   expression(Spectral~~energy~~exposure~~(kJ~m^{-2}~d^{-1}~nm^{-1})),
       x = "Wavelength (nm)") + xlim(290, 425) + ylim(0, 25)

cie.pe <-
  e_irrad(sun.daily.spct * polythene.new.spct, CIE()) * 1e-3
cie.ps <-
  e_irrad(sun.daily.spct * polyester.new.spct, CIE()) * 1e-3
cie.pc <-
  e_irrad(sun.daily.spct * PC.spct, CIE()) * 1e-3
y.pos <- 22.5

fig_dsun2 <- fig_dsun +
  annotate("text",
           label=paste("Polythene~~filter~~CIE:~",
                       signif(cie.pe, digits=3),
                       "*~kJ~m^{-2}~d^{-1}", sep=""),
           y=y.pos+2, x=300, hjust=0, colour="black",
           parse=TRUE) +
  annotate("text", label=paste("Polyester~~filter~~CIE:~",
                               signif(cie.ps, digits=3),
                               "*~kJ~m^{-2}~d^{-1}", sep=""),
           y=y.pos, x=300,  hjust=0,  colour="red",
           parse=TRUE) +
  annotate("text", label=paste("Polycarbonate~~filter~~CIE:~",
                               signif(cie.pc, digits=3),
                                   "*~kJ~m^{-2}~d^{-1}", sep=""),
           y=y.pos-2, x=300, hjust=0,  colour="blue",
           parse=TRUE)

fig_dsun2 + theme_bw()
```



8.10 Task: using colour as data in plots

The examples in this section use a single spectrum, `sun.spct`, but all functions used are methods for `generiic.spct` objects, so are equally applicable to the plotting of other spectra like transmittance, reflectance or response ones.

When we want to colour-label individual spectral values, for example, by plotting the individual data points with the colour corresponding to their wavelengths, or fill the area below a plotted spectral curve with colours, we need to first `tag` the spectral data set using a waveband definition or a list of waveband definitions. If we just want to add a guide or labels to the plot, we can create new data instead of tagging the spectral data to be plotted. In section 8.10.2 we show code based on tagging spectral data, and in section 8.10.3 the case of using different data for plotting the guide or key is described.

8.10.1 Scale definitions

First we define some new scales for use for plotting with `ggplot` when plotting wavelength derived colours. In the future something equivalent may be included in package `photobiologygg` as predefined scales. We define two very similar scales, one for colour, and one for fill aesthetics.

```
scale_colour_tgspct <-
  function(...,
           tg.spct,
           labels = NULL,
           guide = NULL,
           na.value=NA) {
    spct.tags <- attr(tg.spct, "spct.tags", exact=TRUE)
    if (is.null(guide)){
      if (spct.tags$wb.num > 12) {
        guide = "none"
      } else {
        guide = guide_legend(title=NULL)
      }
    }
    values <- as.character(spct.tags$wb.colors)
    if (is.null(labels)) {
      labels <- spct.tags$wb.names
    }
    ggplot2:::manual_scale("colour",
                           values = values,
                           labels = labels,
                           guide = guide,
                           na.value = na.value,
                           ...)
}
```

```
scale_fill_tgspct <-
  function(...,
           tg.spct,
           labels = NULL,
           guide = NULL,
           na.value=NA) {
    spct.tags <- attr(tg.spct, "spct.tags", exact=TRUE)
    if (is.null(guide)){
      if (spct.tags$wb.num > 12) {
        guide = "none"
      } else {
        guide = guide_legend(title=NULL)
      }
    }
    values <- as.character(spct.tags$wb.colors)
    if (is.null(labels)) {
      labels <- spct.tags$wb.names
    }

    ggplot2:::manual_scale("fill",
                           values = values,
                           labels = labels,
                           guide = guide,
                           na.value = na.value,
                           ...)
}
```

8.10.2 Plots using colour for the spectral data

We start by describing how to tag a spectrum, and then show how to use tagged spectra for plotting data. Tagging consist in adding wavelength-derived colour data and waveband-related data to a spectral object. We start with a very simple example.

```
cp.sun.spct <- copy(sun.spct)
tag(cp.sun.spct)
```

As no waveband information was supplied as input, only wavelength-dependent colour information is added to the spectrum plus a factor `wb.f` with only NA level.

If we instead provide a waveband as input then both wavelength-dependent colour and waveband information are added to the spectral data object.

```
uvb.sun.spct <- copy(sun.spct)
tag(uvb.sun.spct, UVB())
levels(uvb.sun.spct[["wb.f"]])

## [1] "UVB"
```

The output contains the same variables (columns) but now the factor `wb.f` has a level based on the name of the waveband, and a value of NA outside it.

We can alter the name used for the `wb.f` factor levels by using a named list as argument.

```
tag(uvb.sun.spct, list('ultraviolet-B' = UVB()))
levels(uvb.sun.spct[["wb.f"]])

## [1] "ultraviolet-B"
```

This example also shows, that re-tagging a spectrum replaces the old tagging data with the new one.

If we use a list of wavebands then the tagging is based on all of them, but be aware that the wavelength ranges of the wavebands overlap, the result is undefined.

```
plant.sun.spct <- copy(sun.spct)
tag(plant.sun.spct, Plant_bands())
levels(plant.sun.spct[["wb.f"]])

## [1] "UVB"   "UVA"   "Blue"  "Green" "R"
## [6] "FR"
```

Tagging also adds some additional data as an attribute to the spectrum. This data can be retrieved with the base R function `attr`.

```
attr(cp.sun.spct, "spct.tag")

## $time.unit
## [1] "second"
##
## $wb.key.name
## [1] "Bands"
##
## $wl.color
## [1] TRUE
##
## $wb.color
## [1] TRUE
```

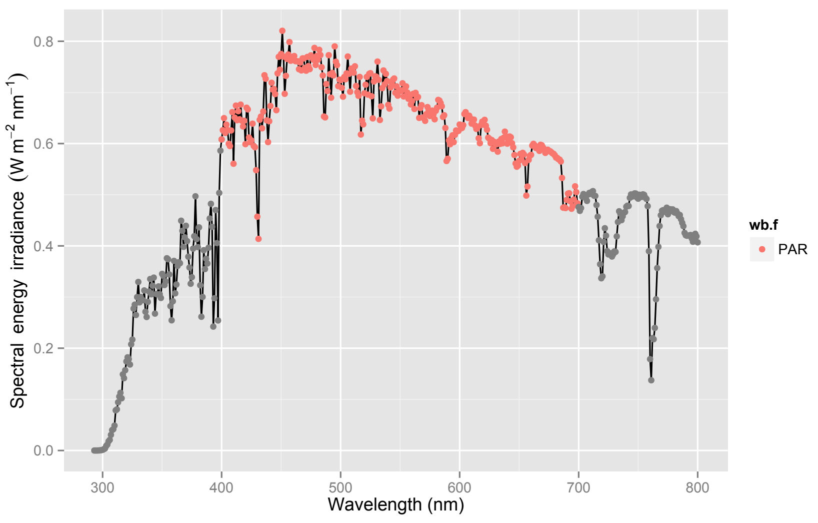
```
##
## $wb.num
## [1] 0
##
## $wb.colors
## [1] NA
##
## $wb.names
## [1] NA
##
## $wb.list
## NULL

attr(uvb.sun.spct, "spct.tag")

## $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]]
## [1] "#000000"
##
##
## $wb.names
## [1] "ultraviolet-B"
##
## $wb.list
## $wb.list$`ultraviolet-B`
## UVB.ISO
## low (nm) 280
## high (nm) 315
## weighted none
```

We now tag a spectrum for use in our first plot example.

```
par.sun.spct <- copy(sun.spct)
tag(par.sun.spct, PAR())
```

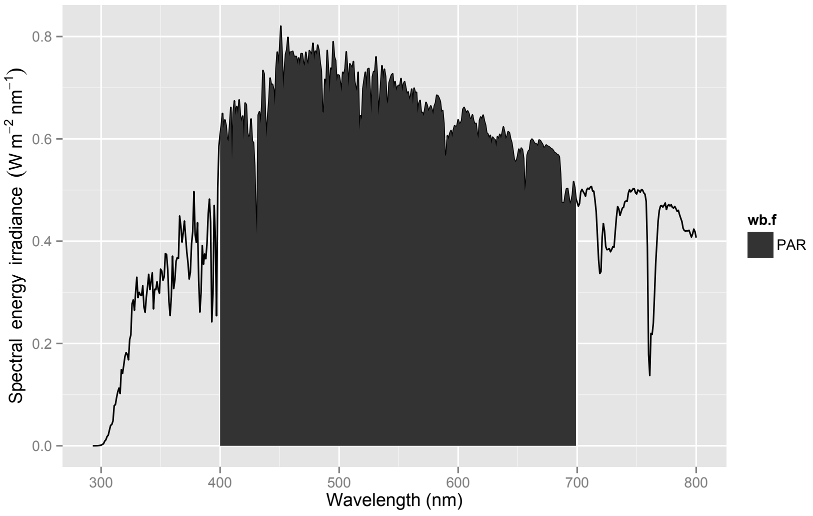
Here we simply use the `wb.f` factor that was added as part of the tagging, with the default colour scale of `ggplot2`, which results in a palette unrelated to the real colour of the different wavelengths.

```
fig_sun.t00 <-
  ggplot(data=par.sun.spct,
         aes(x=w.length, y=s.e.irrad)) +
  geom_line() +
  geom_point(aes(color=wb.f)) +
  labs(
    y = ylab_watt,
    x = "Wavelength (nm)")

fig_sun.t00
```



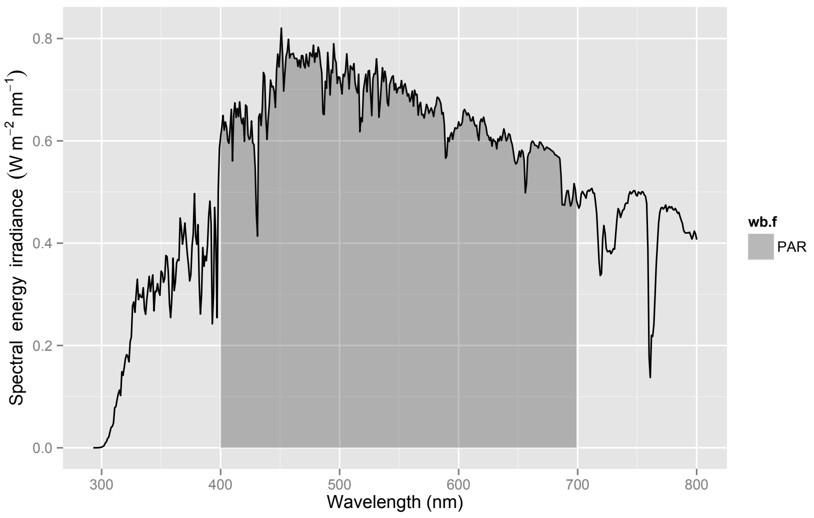
We can also use other geoms like `geom_area` in the next chunk, together with, as an example, a grey fill scale from `ggplot2`.

```
fig_sun.t01 <-
  ggplot(data=par.sun.spct,
         aes(x=w.length, y=s.e.irrad)) +
  geom_line() +
  geom_area(color=NA, aes(fill=wb.f)) +
  scale_fill_grey(na.value=NA) +
  labs(
    y = ylab_watt,
    x = "Wavelength (nm)")

fig_sun.t01
```



The default fill looks too dark and bold, so we change the transparency of the fill by setting `fill = 0.3`. The grid in the background becomes slightly visible also in the filled region, facilitating 'reading' of the plot and avoiding a to stark contrast between regions, which tends to be disturbing. In later plots we frequently use `alpha` to improve how plots look, but we exemplify the effect of changing this aesthetic only here.

```
fig_sun.t01 <-
  ggplot(data=par.sun.spct,
         aes(x=w.length, y=s.e.irrad)) +
  geom_line() +
  geom_area(color=NA, alpha=0.3, aes(fill=wb.f)) +
  scale_fill_grey(na.value=NA) +
  labs(
    y = ylab_watt,
    x = "Wavelength (nm)")

fig_sun.t01
```



As part of the tagging colour information was also added to the spectral data object[5]. We tag each observation in the solar spectrum with human vision colours as defined by ISO.

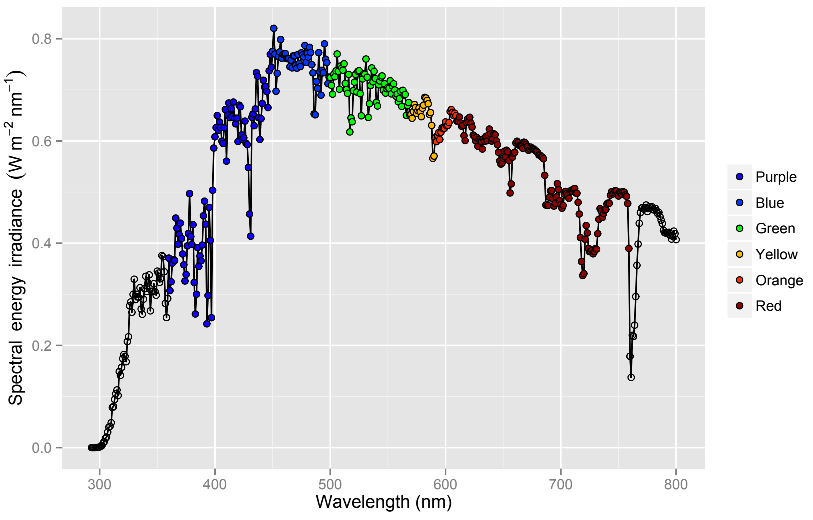
```
tg.sun.spct <- copy(sun.spct)
tag(tg.sun.spct, VIS_bands())
```

See section 8.10.1 on page 97 for the definition of the colour and fill scales used for tagged spectra. These definitions are needed for most of the plots in the remaining of the present and next sections. These scales retrieve information about the wavebands both from the data itself and from the attribute described above.

Here we plot using colours by waveband—using the colour definitions by ISO—, with symbols filled with colours. The colour data outside the wavebands is set to NA so those points are not filled. One can play with the `size` of points until ones get the result wanted. The default 'shape' used by `ggplot2` do not accept a `fill` aesthetic, while shape '21' gives circles that can be 'filled'.

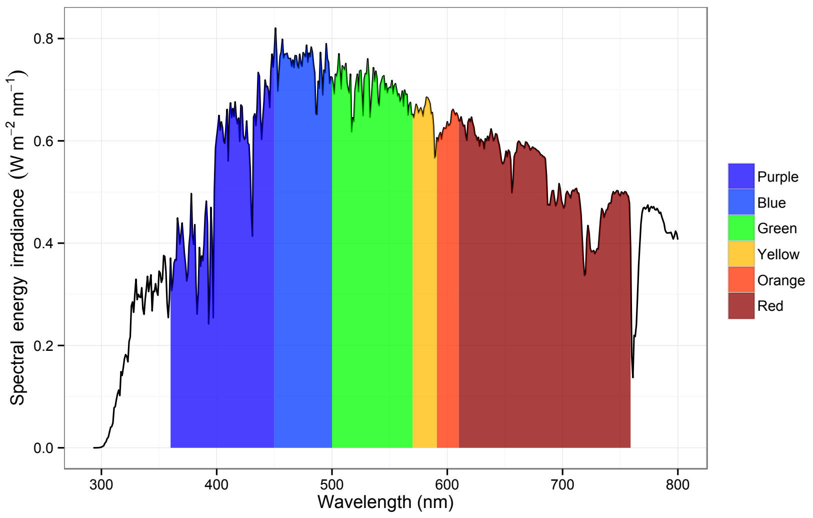
```
fig_sun.t02 <-
  ggplot(data=tg.sun.spct,
         aes(x=w.length, y=s.e.irrad)) +
  geom_line() +
  scale_fill_tgspct(tg.spct=tg.sun.spct) +
  geom_point(aes(fill=wb.f), shape=21)  +
  labs(
    y = ylab_watt,
    x = "Wavelength (nm)")

fig_sun.t02
```

[5]We may want to increase the number of 'observations' in the spectrum by interpolation if there are too few observations for a smooth colour gradient.



Using `geom_area` we can fill the area under the curve according to the colour of different wavebands, we set the fill only for this geom, so that the NAs do not affect other plotting. To get a single black curve for the spectrum we use `geom_line`. This approach works as long as wavebands do not share the same value for the color, which means that it is not suitable either when more than one band is outside the visible range, or when using many narrow wavebands.

```
fig_sun.t03 <-
  ggplot(tg.sun.spct,
         aes(x=w.length, y=s.e.irrad)) +
  scale_fill_tgspct(tg.spct=tg.sun.spct) +
  geom_line() +
  geom_area(aes(fill=wb.f), alpha=0.75) +
  labs(
    y = ylab_watt,
    x = "Wavelength (nm)")

fig_sun.t03 + theme_bw()
```

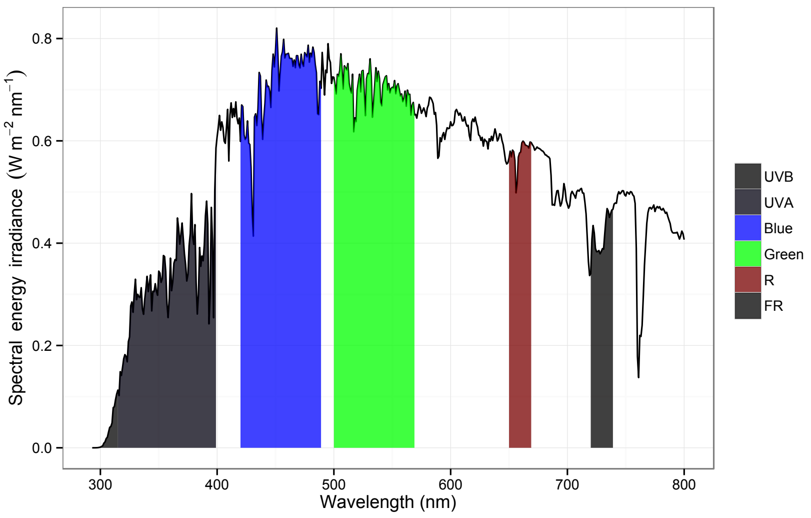


In the next example we tag the solar spectrum with colours using the definitions of plant sensory 'colours'.

```
pl.sun.spct <- copy(sun.spct)
tag(pl.sun.spct, Plant_bands())
```

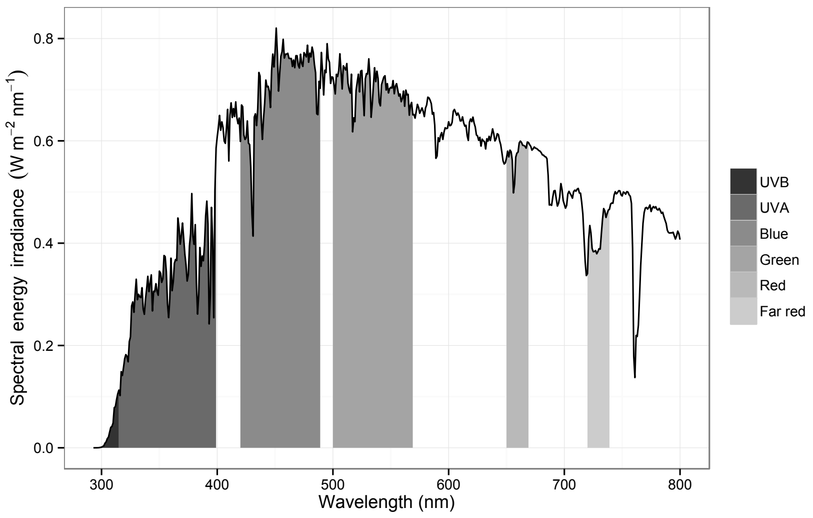
Here we plot the wavebands corresponding to plant sensory 'colours', using the spectrum we tagged in the previous code chunk.

```
fig_sun.pl0 <-
  ggplot(pl.sun.spct,
         aes(x=w.length, y=s.e.irrad)) +
  scale_fill_tgspct(tg.spct=pl.sun.spct) +
  geom_line() +
  geom_area(aes(fill=wb.f), alpha=0.75) +
  labs(
    y = ylab_watt,
    x = "Wavelength (nm)")

fig_sun.pl0 + theme_bw()
```



We can also use the factor `wb.f` which has value NA outside the wavebands, changing the colour used for NA to NA which renders it invisible. We can change the labels used for the wavebands in two different way, when plotting by supplying a labels argument to the scale used, or when tagging the spectrum. The second approach is simpler when producing several different plots from the same spectral object, or when wanting to have consistent labels and names used also in derived results such as irradiance.

```
fig_sun.pl1 <-
  ggplot(pl.sun.spct,
         aes(x=w.length, y=s.e.irrad)) +
  geom_area(aes(fill=wb.f)) +
  scale_fill_grey(na.value=NA, name="",
                  labels=c("UVB", "UVA", "Blue",
                           "Green", "Red", "Far red")) +
  geom_line() +
  labs(
    y = ylab_watt,
    x = "Wavelength (nm)")

fig_sun.pl1 + theme_bw()
```



When using a factor we can play with the scale definitions and represent the wavebands in any way we may want. For example we can use `split_bands` to split a waveband or spectrum into many adjacent narrow bands and get an almost continuous gradient, but we need to get around the problem of repeated colours by using the factor and redefining the scale.

When an spectrum has very few observations we can 'fake' a longer spectrum by interpolation as a way of getting a more even fill. The example below is not run, in later examples we just use the example spectral data as is.

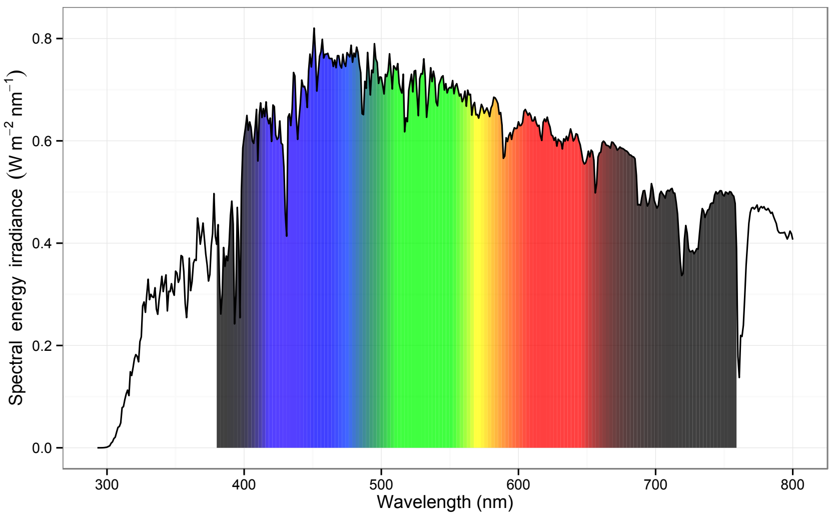
```
interpolate_spct(sun.spct, length.out=800)
```

We tag the VIS region of the spectrum with 150 narrow wavebands. As 'hinges' are inserted, there is no gap, and usually there is no need to increase the length of the spectrum by interpolation. If needed one could try something like. However, the longer spectrum should not be used for statistical calculations, not even plotting using `geom_smooth`.

```
splt.sun.spct <- copy(sun.spct)
tag(splt.sun.spct, split_bands(VIS(), length.out=150))
```

In the code above, we made a copy of `sun.spct` because being part of the package, it is write protected, and `tag` works by modifying its argument.

```
fig_sun.splt0 <-
  ggplot(splt.sun.spct,
         aes(x=w.length, y=s.e.irrad)) +
  scale_fill_tgspct(tg.spct=splt.sun.spct) +
  geom_area(aes(fill=wb.f), alpha=0.75) +
  geom_line() +
  labs(
    y = ylab_watt,
    x = "Wavelength (nm)")

fig_sun.splt0 + theme_bw()
```

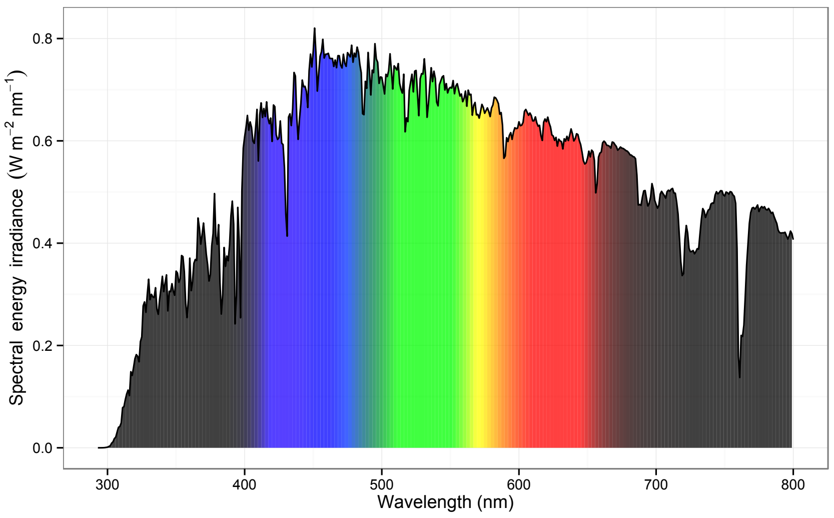


In this other example we tag the whole spectrum, dividing it into 200 wavebands.

```
splt1.sun.spct <- copy(sun.spct)
# splt1.sun.spct <- interpolate_spct(splt1.sun.spct, length.out=1000)
tag(splt1.sun.spct, split_bands(sun.spct, length.out=200))
```

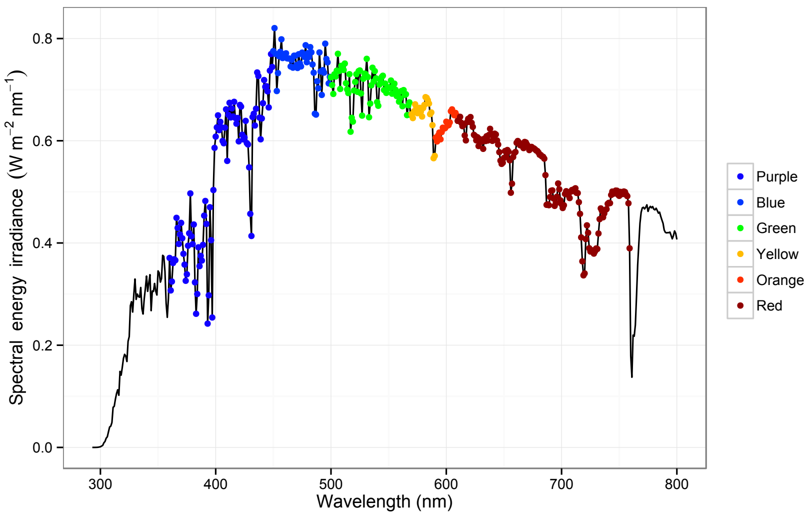
We use `geom_area` and `fill`, and colour the area under the curve. This does not work with `geom_line` because there would not be anything to fill, here we use `geom_area` instead.

```
fig_sun.splt1 <-
  ggplot(splt1.sun.spct,
         aes(x=w.length, y=s.e.irrad)) +
  scale_fill_tgspct(tg.spct=splt1.sun.spct) +
  geom_area(aes(fill=wb.f), alpha=0.75) +
  geom_line() +
  labs(
    y = ylab_watt,
    x = "Wavelength (nm)")

fig_sun.splt1 + theme_bw()
```



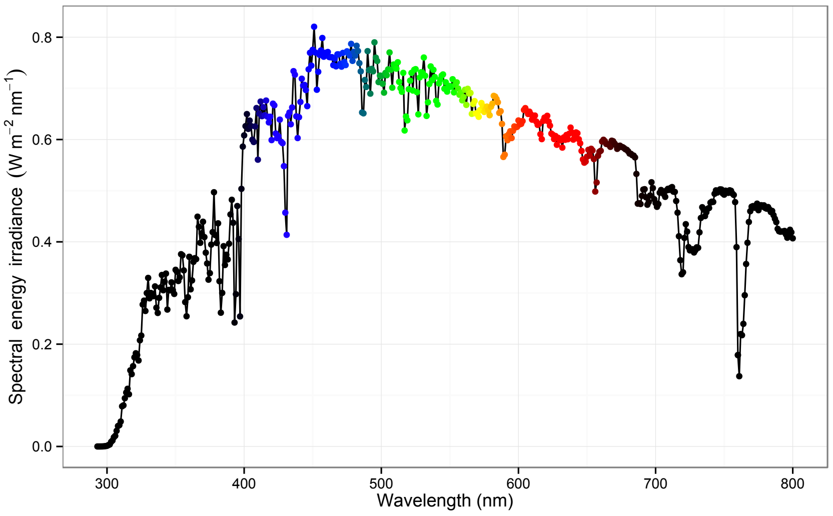
The next example uses `geom_point` and `colour` to color the data points according the waveband they are included in.

```
fig_sun.tg1 <-
  ggplot(tg.sun.spct,
         aes(x=w.length, y=s.e.irrad)) +
  scale_colour_tgspct(tg.spct=tg.sun.spct) +
  geom_line() +
  geom_point(aes(colour=wb.f)) +
  labs(
    y = ylab_watt,
    x = "Wavelength (nm)")

fig_sun.tg1 + theme_bw()
```



When plotting points, rather than an area we may, instead of using colours from wavebands, want to plot the colour calculated for each individual wavelength value, which `tag` adds to the spectrum, whether a waveband definition is supplied or not. In this case we need to use `scale_color_identity`.

```
fig_sun.tg2 <-
  ggplot(data=tg.sun.spct,
         aes(x=w.length, y=s.e.irrad)) +
  geom_line() +
  scale_color_identity() +
  geom_point(aes(color=wl.color))  +
  labs(
    y = ylab_watt,
    x = "Wavelength (nm)")

fig_sun.tg2  + theme_bw()
```



Other possibilities are for example, using one of the symbols that can be filled, and then for example for symbols with a black border and a colour matching its wavelength as a fill aesthetic. It is also possible to use `alpha` with points.

8.10.3 Plots using waveband definitions

In the previous section we showed how tagging spectral data can be used to add colour information that can be used when plotting. In contrast, in the present section we create new 'fake' spectral data starting from waveband definitions that then we plot as 'annotations'. We show different types of annotations based on plotting with different geoms. We show the use of `geom_rect`, `geom_text`, `geom_vline`, and `geom_segment`, that we consider the most useful geometries in this context.

We use three different functions from package `photobiology` to generate the data to be plotted from lists of waveband definitions. We use mainly predefined wavebands, but user defined wavebands can be used as well. We start by showing the output of these functions, starting with `wb2spct` the simplest one.

```
wb2spct(PAR())
wb2spct(Plant_bands())
```

Function `wb2tagged_spct` returns the same 'spectrum', but tagged with the same wavebands as used to create the spectral data, and you will also notice that a 'hinge' has been added, which is redundant in the case of a single waveband, but needed in the case of wavebands sharing a limit.

```
wb2tagged_spct(PAR())
wb2tagged_spct(Plant_bands())
```

The third function, `wb2rect_spct` is what we use in most examples. It generates data that make it easier to plot rectangles with `geom_rect` as we will see in later examples.

```
wb2rect_spct(PAR())
wb2rect_spct(Plant_bands())
```

In this case instead of two rows per waveband, we obtain only one row per waveband, with a `w.length` value corresponding to its midpoint but with two additional columns giving the low and high wavelength limits.

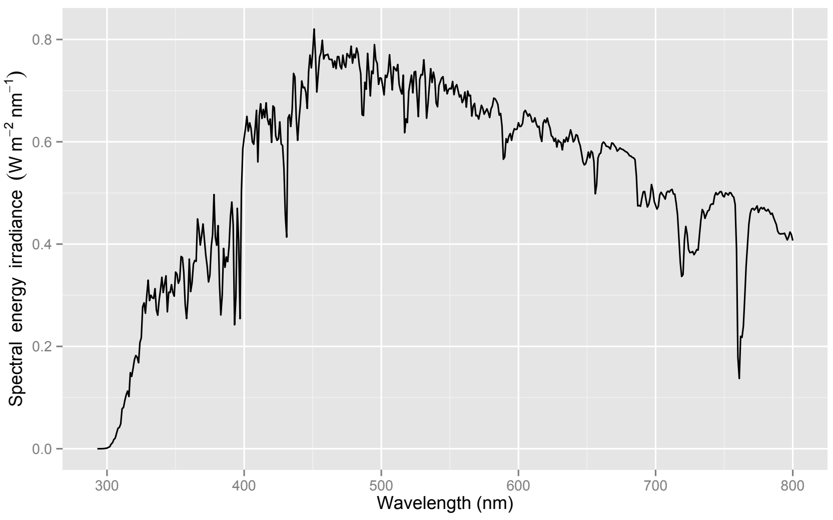
As we saw earlier for tagged spectra, additional data is stored in an attribute.

```
attr(wb2rect_spct(PAR()), "spct.tags")

## $time.unit
## [1] "none"
##
## $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
```

The first plot examples show how to add a colour bar as key. We create new data for use in what is closer to the concept of annotation that to plotting. In most of the examples below we use waveband definitions to create tagged spectral data for use in plotting the guide using `geom_rect`. We present three cases: an almost continuous colour reference guide, a reference guide for colours perceived by plants and one for ISO colour definitions. We also add labels to the bar with `geom_text` and show some examples of how to change the color of the line enclosing the rectangles and of text labels. Finally we show how to use `fill` and `alpha` to adjust how the guides look. Later on we show some examples using other geoms and also examples combining the use of tagged spectra as described in the previous section with the 'annotations' described here.

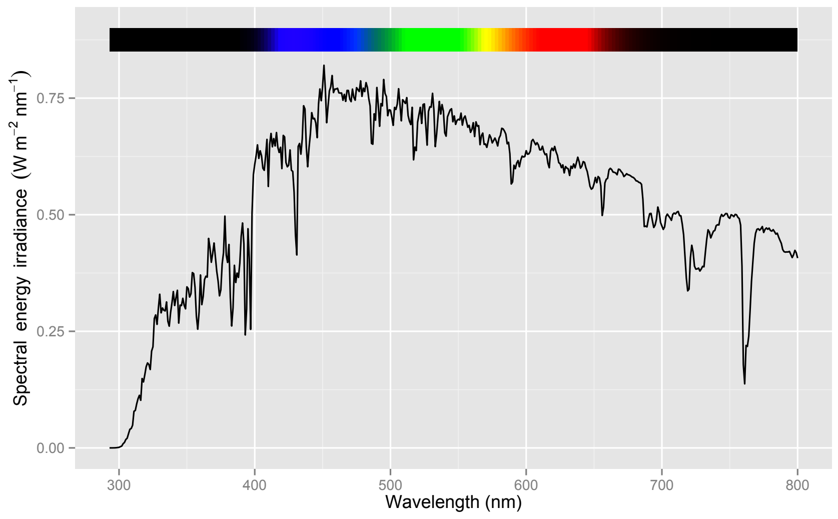
First we create a simple line plot of the solar spectrum, that we will use as a basis for most of the examples below.

```
fig_sun.z0 <-
  ggplot(data=sun.spct,
         aes(x=w.length, y=s.e.irrad)) +
  geom_line() +
  labs(
    y = ylab_watt,
    x = "Wavelength (nm)")

fig_sun.z0
```



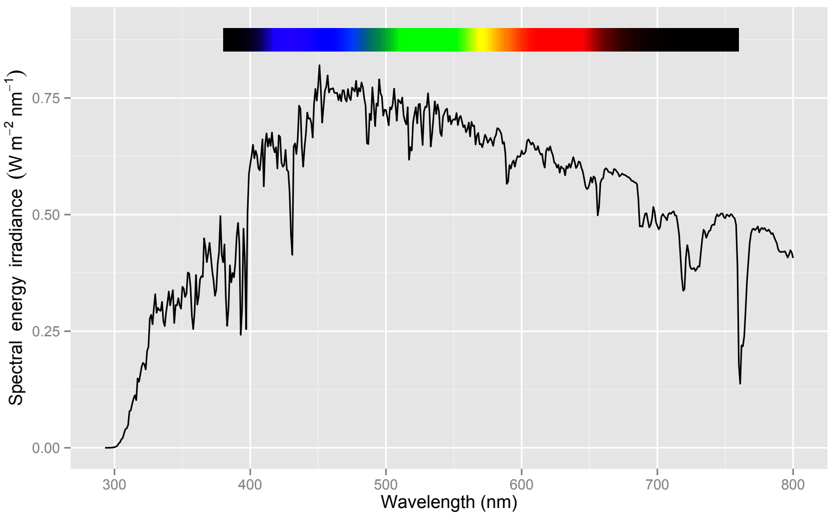
We now add to the plot created above a nearly continuous colour bar for the whole spectrum. To obtain an almost continuous colour scale we use a list of 200 wavebands. We need to specify `color = NA` to prevent the line enclosing each of the 200 rectangles from being plotted. We position the bar at the top because we think that it looks best, but by changing the values supplied to `ymax` and `ymin` move the bar vertically and also change its width.

```
wl.guide.spct <-
  wb2rect_spct(split_bands(sun.spct,
                                   length.out=200))

fig_sun.z2 <- fig_sun.z0 +
  geom_rect(data=wl.guide.spct,
            aes(xmin = wl.low, xmax = wl.high,
                ymin = y + 0.85, ymax = y + 0.9,
                y = 0, fill=wb.f),
            color = NA) +
  scale_fill_tgspct(tg.spct=wl.guide.spct)

fig_sun.z2
```



This second example differs very little from the previous one, but by using a waveband definition instead of a spectrum as argument to `split_bands`, we restrict the region covered by the colour fill to that of the waveband. In fax a vector of length two, or any object for which a `range` method is available can be used as input to this function.

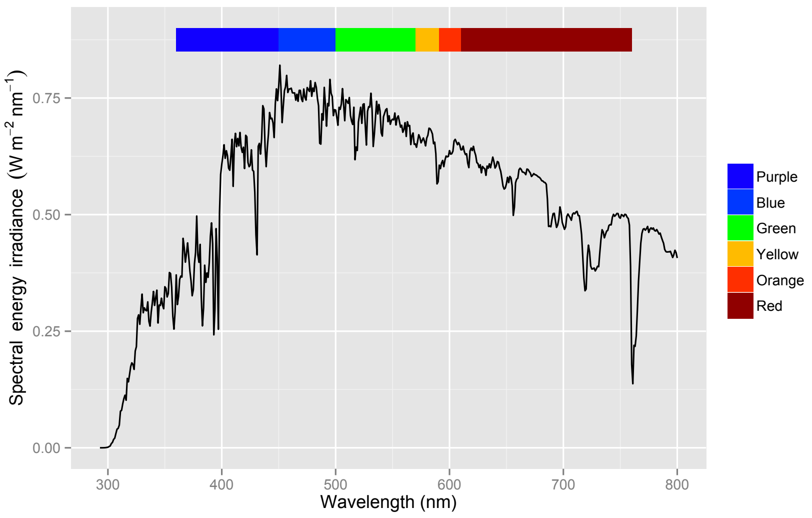
```
wl.guide.spct <- wb2rect_spct(split_bands(VIS(), length.out=200))

fig_sun.z1 <- fig_sun.z0 +
  geom_rect(data=wl.guide.spct,
            aes(xmin = wl.low, xmax = wl.high,
                ymin = y + 0.85, ymax = y + 0.9,
                y = 0, fill=wb.f),
            color = NA) +
  scale_fill_tgspct(tg.spct=wl.guide.spct)

fig_sun.z1
```



In the examples above we have used a list of 200 waveband definitions created with `split_bands`. If we instead use a shorter list of definitions, we get a plot where the wavebands are clearly distinguished. By default if the list of wavebands is short, a key or 'guide' is also added to the plot.

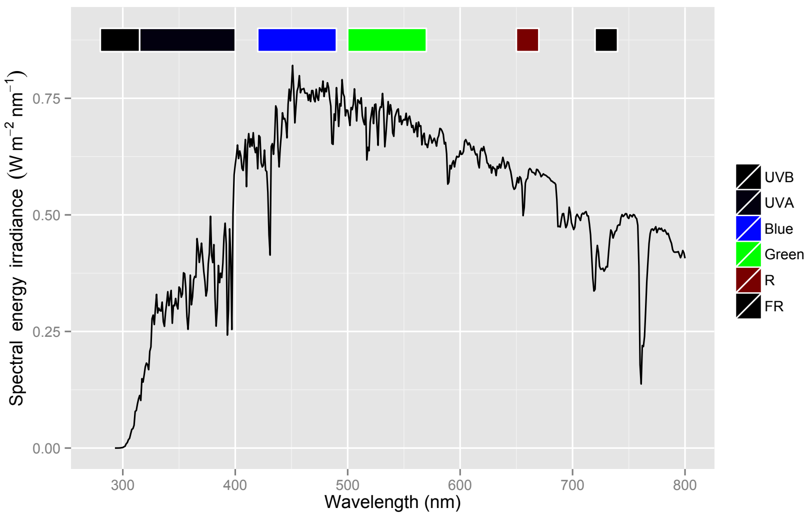
To demonstrate this we replace in the previous example, the previous tagged spectrum with one based on ISO colours. We need to do this replacement in the calls to both `geom_rect` and `scale_fill_tgspct`.

```
iso.guide.spct <- wb2rect_spct(VIS_bands())

fig_sun.z3 <- fig_sun.z0 +
  geom_rect(data=iso.guide.spct,
            aes(xmin = wl.low, xmax = wl.high,
                ymin = y + 0.85, ymax = y + 0.9,
                y = 0, fill=wb.f),
            color = NA) +
  scale_fill_tgspct(tg.spct=iso.guide.spct)

fig_sun.z3
```



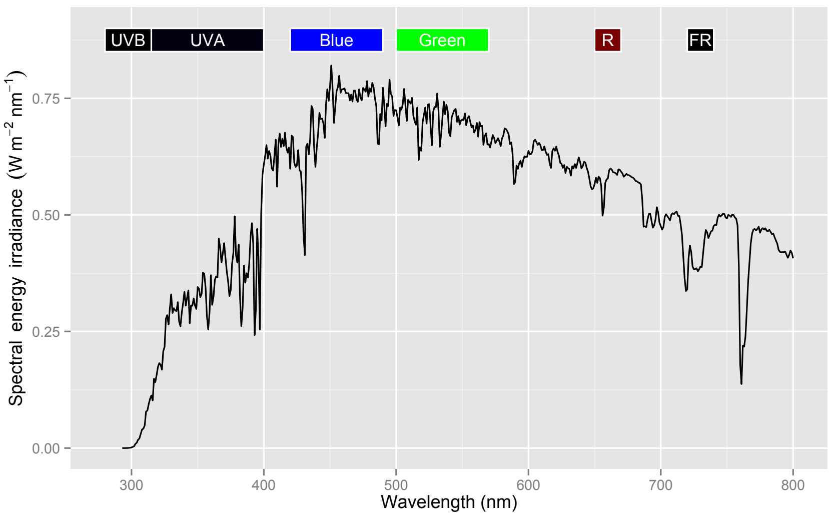
We use as an example plant's sensory colours, to show the case when the wavebands in the list are not contiguous.

```
plant.guide.spct <- wb2rect_spct(Plant_bands())

fig_sun.z4 <- fig_sun.z0 +
  geom_rect(data=plant.guide.spct,
            aes(xmin = wl.low, xmax = wl.high,
                ymin = y + 0.85, ymax = y + 0.9,
                y = 0, fill=wb.f),
            color = "white") +
  scale_fill_tgspct(tg.spct=plant.guide.spct)

fig_sun.z4
```



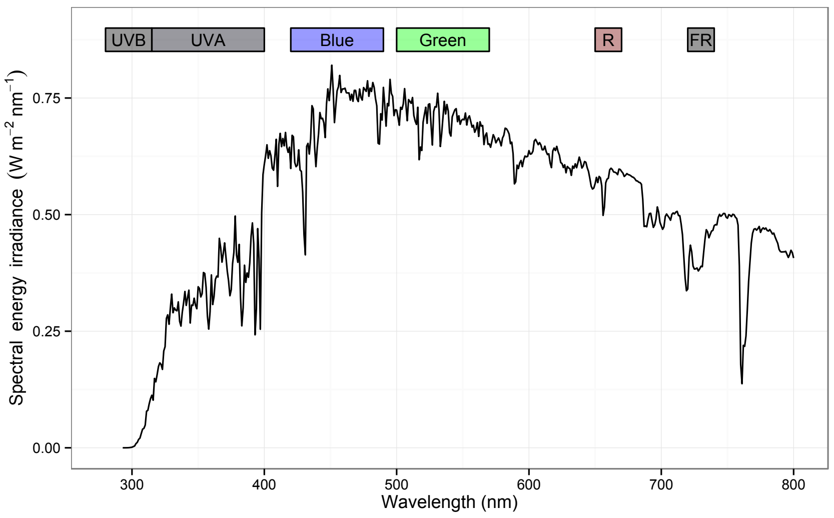
We add text labels on top of the guide, and make the rectangle borders and text white to make the separation between the different 'invisible' wavebands clear. As we are adding labels, the 'guide' or key becomes redundant and we remove it by adding `guide="none"` to the fill scale.

```
plant.guide.spct <- wb2rect_spct(Plant_bands())

fig_sun.z5 <- fig_sun.z0 +
  geom_rect(data=plant.guide.spct,
            aes(xmin = wl.low, xmax = wl.high,
                ymin = y + 0.85, ymax = y + 0.9,
                y = 0, fill=wb.f),
            color = "white") +
  geom_text(data=plant.guide.spct,
            aes(y = y + 0.875, label = as.character(wb.f)),
            color = "white", size=4) +
  scale_fill_tgspct(tg.spct=plant.guide.spct, guide="none")

fig_sun.z5
```



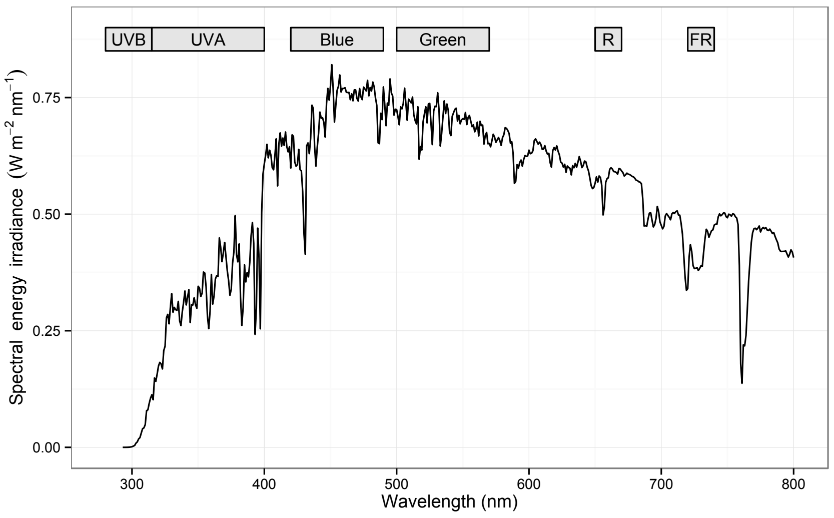
Here we add `alpha` or transparency to make the colours paler, and use black text and lines.

```
plant.guide.spct <- wb2rect_spct(Plant_bands())

fig_sun.z6 <- fig_sun.z0 +
  geom_rect(data=plant.guide.spct,
       aes(xmin = wl.low, xmax = wl.high,
           ymin = y + 0.85, ymax = y + 0.9,
           y = 0, fill=wb.f),
       color = "black", alpha=0.4) +
  geom_text(data=plant.guide.spct,
       aes(y = y + 0.875, label = as.character(wb.f)),
       color = "black", size=4) +
  scale_fill_tgspct(tg.spct=plant.guide.spct, guide="none")

fig_sun.z6 + theme_bw()
```



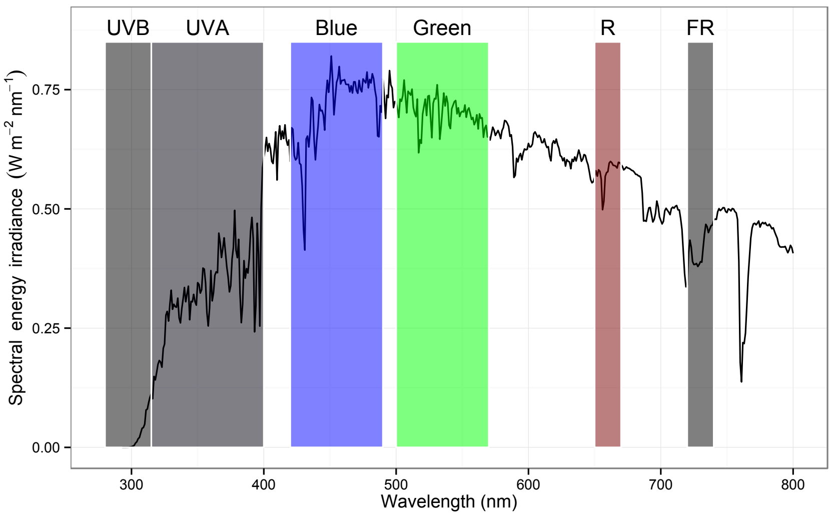
We change the guide so that all rectangles are filled with the same shade of grey by moving `fill` out of `aes` and setting it to a constant.

```
plant.guide.spct <- wb2rect_spct(Plant_bands())

fig_sun.z7 <- fig_sun.z0 +
  geom_rect(data=plant.guide.spct,
            aes(xmin = wl.low, xmax = wl.high,
                ymin = y + 0.85, ymax = y + 0.9,
                y = 0),
            color = "black", fill="grey90") +
  geom_text(data=plant.guide.spct,
            aes(y = y + 0.875, label = as.character(wb.f)),
            color = "black", size=4)

fig_sun.z7 + theme_bw()
```



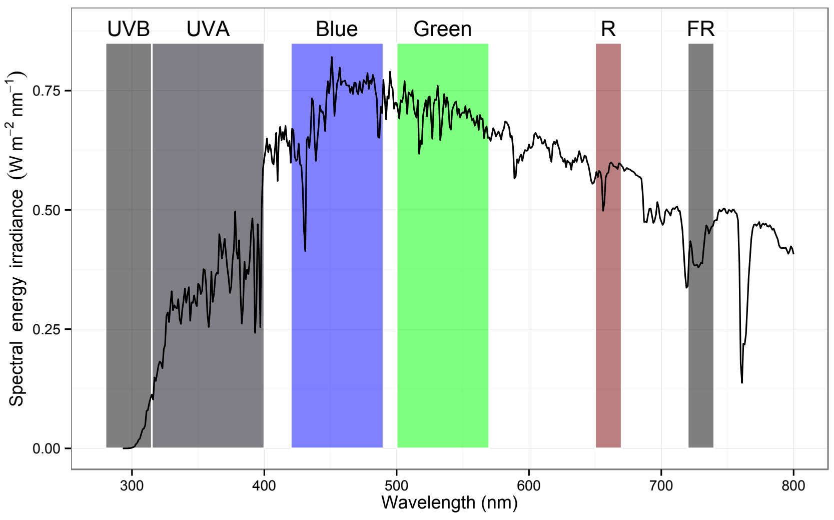
We can obtain annotations similar to those in ?? in page ?? created with `annotate_waveband` using geoms.

```
plant.guide.spct <- wb2rect_spct(Plant_bands())

fig_sun.z8 <- fig_sun.z0 +
  geom_rect(data=plant.guide.spct,
       aes(xmin = wl.low, xmax = wl.high,
           ymin = y, ymax = y + 0.85,
           y = 0, fill=wb.f),
       color = "white", alpha=0.5) +
  geom_text(data=plant.guide.spct,
       aes(y = y + 0.88, label = as.character(wb.f)),
       color = "black") +
  scale_fill_tgspct(tg.spct=plant.guide.spct, guide="none")

fig_sun.z8 + theme_bw()
```



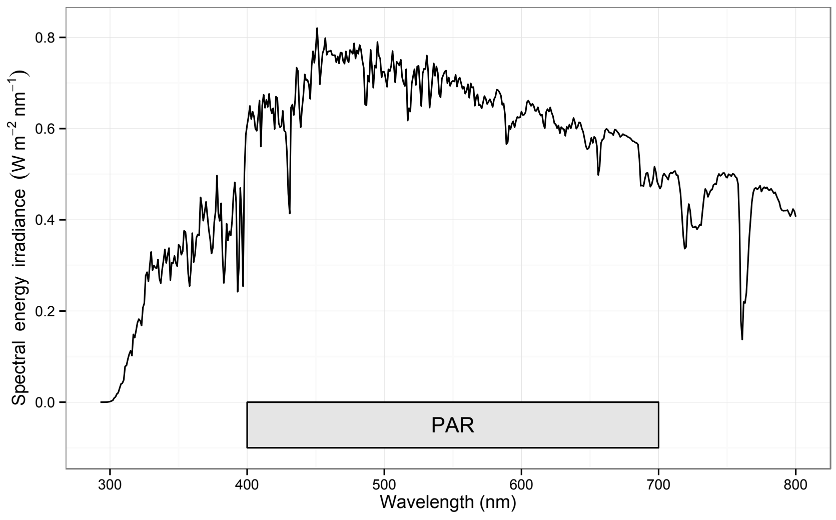
The example above can be improved by changing the order in which the geoms are added. In the plot above we can see that the rectangles are plotted on top of the line for the spectral irradiance. By changing the order we obtain a better plot.

```
plant.guide.spct <- wb2rect_spct(Plant_bands())

fig_sun.z8a <-
  ggplot(data=sun.spct,
         aes(x=w.length, y=s.e.irrad)) +
  geom_rect(data=plant.guide.spct,
       aes(xmin = wl.low, xmax = wl.high,
           ymin = y, ymax = y + 0.85,
           y = 0, fill=wb.f),
       color = "white", alpha=0.5) +
  geom_text(data=plant.guide.spct,
       aes(y = y + 0.88, label = as.character(wb.f)),
       color = "black") +
  geom_line() +
  scale_fill_tgspct(tg.spct=plant.guide.spct, guide="none") +
  labs(
    y = ylab_watt,
    x = "Wavelength (nm)")

fig_sun.z8a + theme_bw()
```



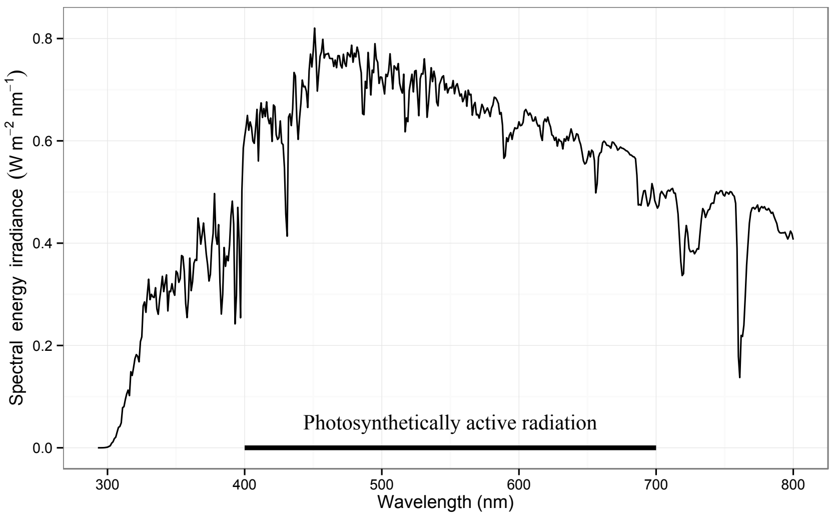
In the examples above we used predefined lists of wavebands, but one can, of course, use any list of waveband definitions, for example explicitly created with `list` and `new_waveband`, or `list` and any combination of user-defined and predefined wavebands. Even single waveband definitions are allowed.

```
par.guide.spct <- wb2rect_spct(PAR())

fig_sun.z9 <- fig_sun.z0 +
  geom_rect(data=par.guide.spct,
            aes(xmin = wl.low, xmax = wl.high,
                ymin = y - 0.1, ymax = y,
                y = 0),
            color = "black", fill="grey90") +
  geom_text(data=par.guide.spct,
            aes(y = y - 0.05, label = as.character(wb.f)),
            color = "black")

fig_sun.z9 + theme_bw()
```



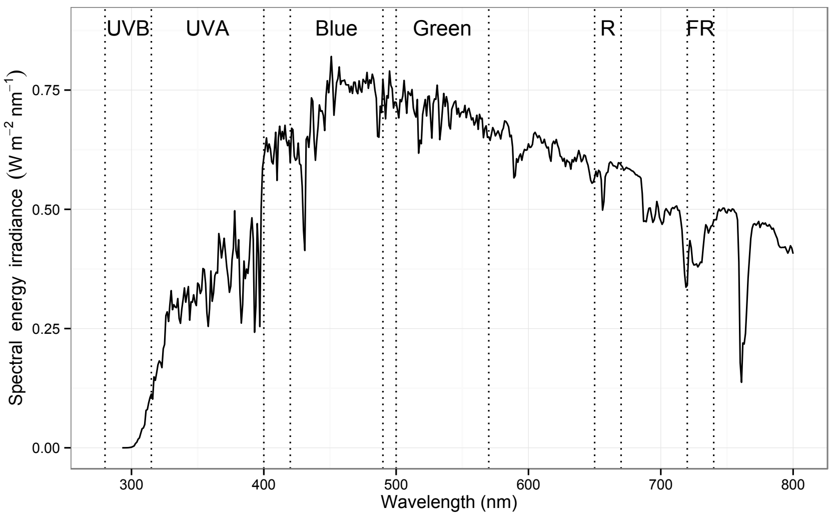
We can also use `geom_segment` to draw lines, including arrows. In this example we also set a different font `family` and label text. We can replace the label text which is by default obtained from the waveband definition by assigning a name to the waveband as member of the list. We use single quotes so that the long name containing space characters is accepted by `list`.

```
par.guide1.spct <-
  wb2rect_spct(list('Photosynthetically active radiation' = PAR()))

fig_sun.z10 <- fig_sun.z0 +
  geom_segment(data=par.guide1.spct,
        aes(x = wl.low, xend = wl.high,
            y = y, yend = y),
        size = 1.5, color = "black") +
  geom_text(data=par.guide1.spct,
        aes(y = y + 0.05, label = as.character(wb.f)),
        color = "black", family="serif")

fig_sun.z10 + theme_bw()
```



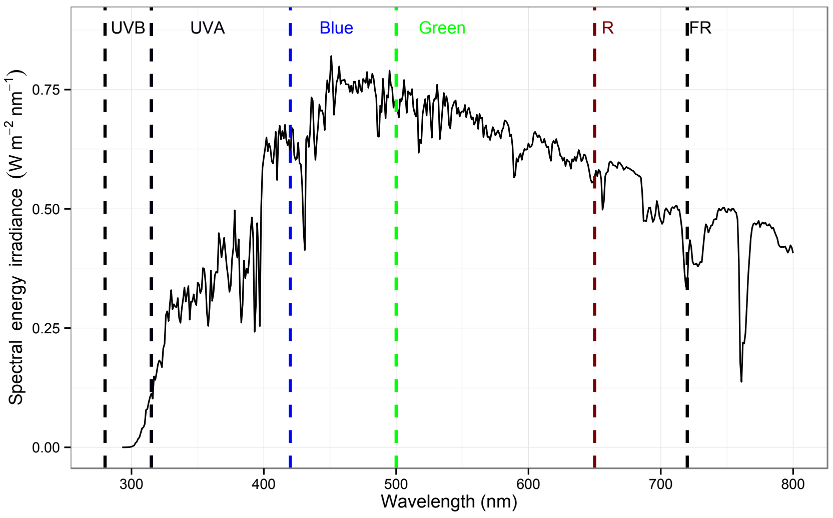
In this section we have used until now function wb2rect_spct to create 'spectral' annotation data from waveband definitions. Two other functions are available, that are needed or easier to use in some cases. One such case is when we have a list of wavebands and we would like to mark their boundaries with vertical lines. How to do this with annotate and range was show earlier in this chapter, but this can become tedious when we have several wavebands. Here we show an alternative approach.

```r
plant.boundaries.spct <- wb2spct(Plant_bands())
plant.guide.spct <- wb2rect_spct(Plant_bands())

fig_sun.z11 <- fig_sun.z0 +
  geom_vline(data=plant.boundaries.spct,
             aes(xintercept = w.length),
             linetype = "dotted") +
  geom_text(data=plant.guide.spct,
        aes(y = y + 0.88, label = as.character(wb.f)),
        color = "black")

fig_sun.z11 + theme_bw()
```



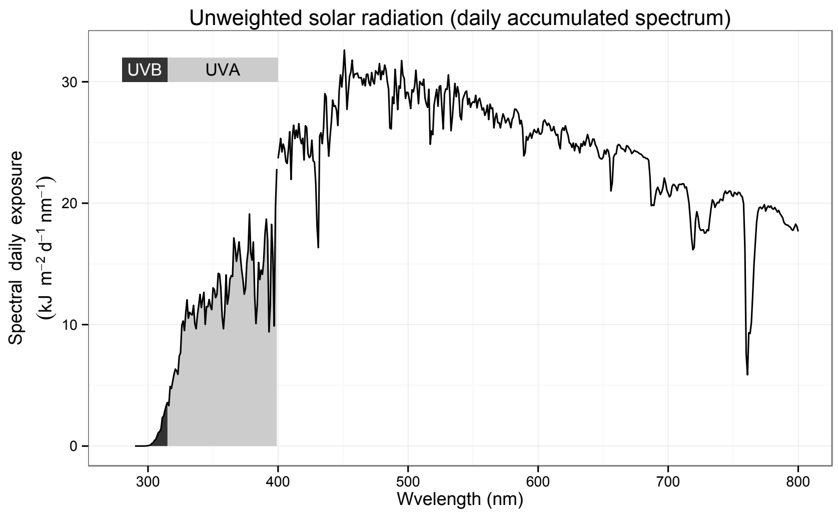
Function `wb2tagged_spct` returns the same data as `wb2spct` but 'tagged'. As shown in the next code chunk, tagging allows us to use waveband-dependent colours to the vertical lines.

```r
plant.boundaries.spct <- wb2tagged_spct(Plant_bands())
plant.guide.spct <- wb2rect_spct(Plant_bands())

fig_sun.z12 <- fig_sun.z0 +
  geom_vline(data=plant.boundaries.spct,
             aes(xintercept = w.length, color=wb.f),
             size=1, linetype="dashed") +
  geom_text(data=plant.guide.spct,
        aes(y = y + 0.88, label = as.character(wb.f), colour=wb.f),
        size=4) +
  scale_colour_tgspct(tg.spct=plant.guide.spct, guide="none")

fig_sun.z12 + theme_bw()
```



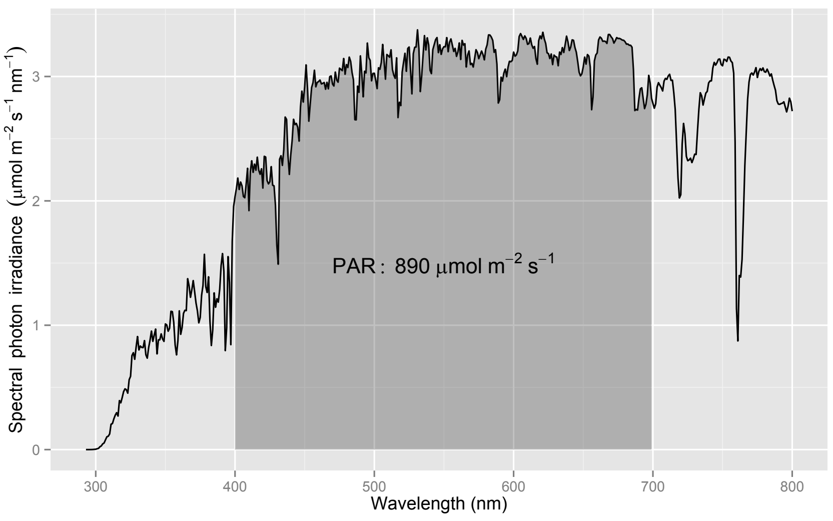
Of course it is possible to combine tagged data spectra and tagged spectra created from wavebands. The tagging is consistent, so, as demonstrated in the next figure, the same aesthetic 'link' works for both spectra. In this case the fill scale and the setting of fill to `wb.f` work accross different 'data' and yield a consistent look. This figure also shows that when assigning a constant to an aesthetic, it is possible to use a vector, which in the present example, saves us some work compared to adding a column to the data and using an identity scale. Contrary to earleir examples where we have added layers to a previously saved plot, here we show the whole code needed to build the figure.

```
my.sun.spct <- sun.daily.spct
tag(my.sun.spct, list(UVB(), UVA()))
annotation.spct <- wb2rect_spct(list(UVB(), UVA()))
fig_sun.uv1 <- ggplot(my.sun.spct,
                      aes(x=w.length,
                          y=s.e.irrad * 1e-3,
                          fill=wb.f)) +
  scale_fill_grey(na.value=NA, guide="none") +
  geom_area() + geom_line() +
  labs(x = "Wvelength (nm)",
       y =  expression(atop(Spectral~~daily~~exposure,
                           (kJ~~m^{-2}~d^{-1}~nm^{-1}))),
       fill = "",
       title =
   "Unweighted solar radiation (daily accumulated spectrum)") +
  geom_rect(data=annotation.spct,
            aes(xmin=wl.low, xmax=wl.high, ymin=30, ymax=32)) +
  geom_text(data=annotation.spct,
            aes(label=as.character(wb.f), y=31),
            color=c("white","black"), size=4) +
  theme_bw()

fig_sun.uv1
```

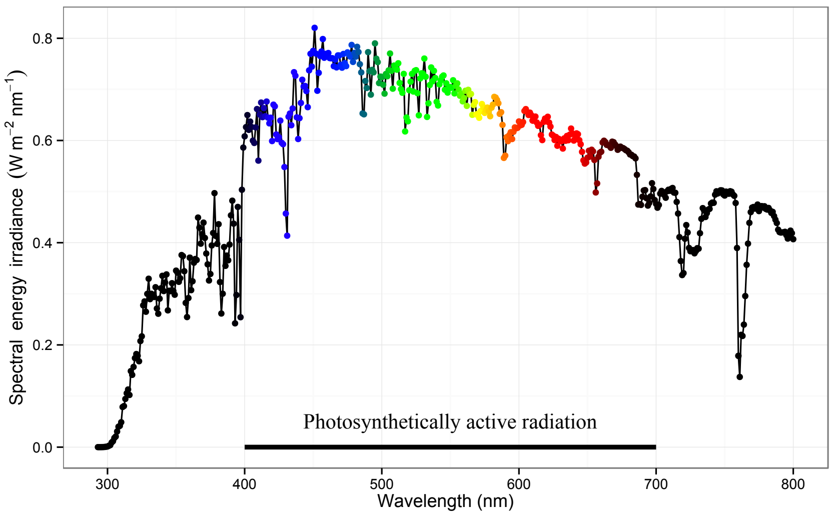


Possible variations are almost endless, so we invite the reader to continue exploring how the functions from package `photobiology` can be used together with `ggplot`, to obtain beautiful plots of spectra. As an example here we show new versions of two plots from the previous section, one using a filled area to label the PAR region, and another one using symbols with colours according to their wavelength, to which we add a guide for PAR.

```
par <- q_irrad_spct(sun.spct, PAR()) * 1e6

fig_sun.tgrect1 <-
  ggplot(data=par.sun.spct,
         aes(x=w.length, y=s.q.irrad * 1e6)) +
  geom_line() +
  geom_area(color=NA, alpha=0.3, aes(fill=wb.f))  +
  scale_fill_grey(na.value=NA, guide="none") +
  labs(
    y = ylab_umol,
    x = "Wavelength (nm)") +
  annotate_waveband(PAR(), "text",
                    label=paste("PAR:~", signif(par,digits=2),
                                " * ~mu * mol~m^{-2}~s^{-1}", sep=""),
                    y=1.5, colour="black", size=5, parse=TRUE)

fig_sun.tgrect1
```



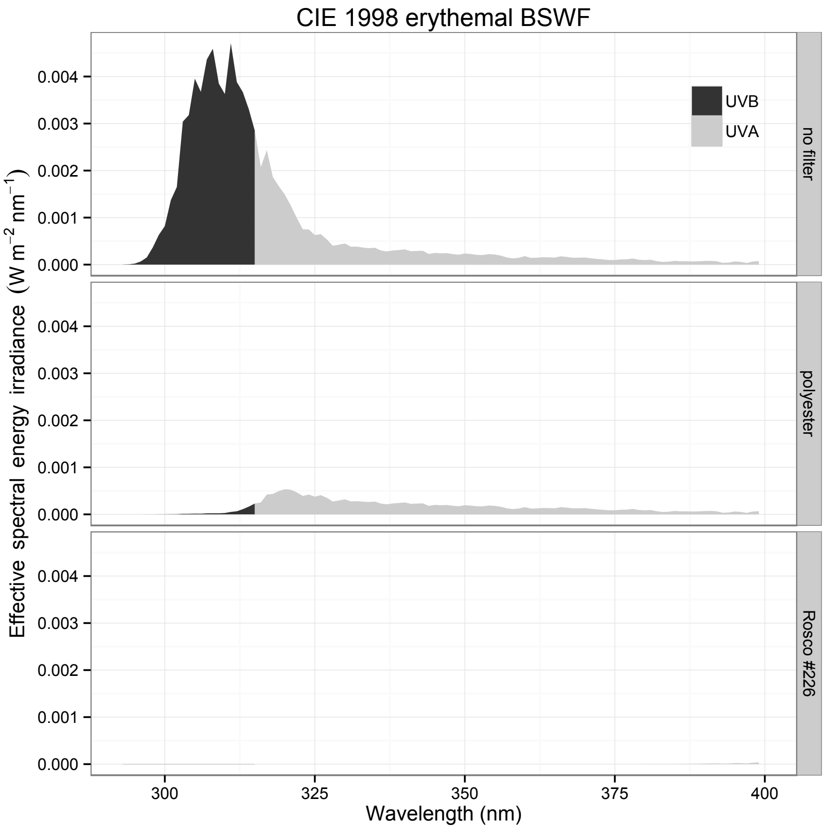
```
par.guide.spct <-
  wb2rect_spct(list('Photosynthetically active radiation' = PAR()))

fig_sun.tgrect2 <-
  ggplot(data=tg.sun.spct,
         aes(x=w.length, y=s.e.irrad)) +
  geom_line() +
  scale_color_identity() +
  geom_point(aes(color=wl.color))  +
  labs(
    y = ylab_watt,
    x = "Wavelength (nm)") +
  geom_segment(data=par.guide.spct,
       aes(x = wl.low, xend = wl.high, y = y, yend = y),
       size = 1.5, color = "black") +
  geom_text(data=par.guide.spct,
       aes(y = y + 0.05, label = as.character(wb.f)),
       color = "black", family="serif")

fig_sun.tgrect2 + theme_bw()
```



8.11 Task: plotting effective spectral irradiance

This task is here simply to show that there is nothing special about plotting spectra based on calculations, and that one can combine different functions to get the job done. We also show how to 'row bind' spectra for plotting, in this case to make it easy to use facets.

```
sun.eff.cie.nf.spct <- sun.spct * CIE()
sun.eff.cie.pe.spct <- sun.spct * polyester.new.spct * CIE()
sun.eff.cie.226.spct <- sun.spct * uv.226.new.spct * CIE()
tag(sun.eff.cie.nf.spct, UV_bands())
tag(sun.eff.cie.pe.spct, UV_bands())
tag(sun.eff.cie.226.spct, UV_bands())
sun.eff.cie.nf.spct[ , filter := 'no filter']

##       w.length    s.e.irrad  wl.color  wb.f
##    1:       293 2.609665e-06  #000000   UVB
##    2:       294 6.142401e-06  #000000   UVB
##    3:       295 2.176175e-05  #000000   UVB
##    4:       296 6.780119e-05  #000000   UVB
##    5:       297 1.533491e-04  #000000   UVB
##   ---
## 507:       796 0.000000e+00  #000000    NA
## 508:       797 0.000000e+00  #000000    NA
## 509:       798 0.000000e+00  #000000    NA
## 510:       799 0.000000e+00  #000000    NA
## 511:       800 0.000000e+00  #000000    NA
##         filter
##    1: no filter
##    2: no filter
##   3: no filter
##   4: no filter
##   5: no filter
##  ---
## 507: no filter
## 508: no filter
## 509: no filter
## 510: no filter
## 511: no filter

sun.eff.cie.pe.spct[ , filter := 'polyester']

##      w.length    s.e.irrad wl.color wb.f
##   1:      293 7.828995e-09  #000000  UVB
##   2:      294 1.842720e-08  #000000  UVB
##   3:      295 6.528525e-08  #000000  UVB
##   4:      296 2.034036e-07  #000000  UVB
##   5:      297 4.600472e-07  #000000  UVB
##  ---
## 507:      796 0.000000e+00  #000000   NA
## 508:      797 0.000000e+00  #000000   NA
## 509:      798 0.000000e+00  #000000   NA
## 510:      799 0.000000e+00  #000000   NA
## 511:      800 0.000000e+00  #000000   NA
##         filter
##   1: polyester
##   2: polyester
##   3: polyester
##   4: polyester
##   5: polyester
##  ---
## 507: polyester
## 508: polyester
## 509: polyester
## 510: polyester
## 511: polyester

sun.eff.cie.226.spct[ , filter := 'Rosco #226']

##      w.length s.e.irrad wl.color wb.f     filter
##   1:      293         0  #000000  UVB Rosco #226
##   2:      294         0  #000000  UVB Rosco #226
##   3:      295         0  #000000  UVB Rosco #226
##   4:      296         0  #000000  UVB Rosco #226
##   5:      297         0  #000000  UVB Rosco #226
##  ---
## 507:      796         0  #000000   NA Rosco #226
## 508:      797         0  #000000   NA Rosco #226
## 509:      798         0  #000000   NA Rosco #226
## 510:      799         0  #000000   NA Rosco #226
## 511:      800         0  #000000   NA Rosco #226

sun.eff.cie.spct <- rbindspct(list(sun.eff.cie.nf.spct,
                                   sun.eff.cie.pe.spct,
                                   sun.eff.cie.226.spct))
sun.eff.cie.spct[ , filter := factor(filter)]

##       w.length    s.e.irrad wl.color wb.f
##    1:      293 2.609665e-06  #000000  UVB
##    2:       293 7.828995e-09  #000000  UVB
##    3:       293 0.000000e+00  #000000  UVB
##    4:       294 6.142401e-06  #000000  UVB
##    5:       294 1.842720e-08  #000000  UVB
##   ---
## 1529:       799 0.000000e+00  #000000   NA
## 1530:       799 0.000000e+00  #000000   NA
## 1531:       800 0.000000e+00  #000000   NA
## 1532:       800 0.000000e+00  #000000   NA
## 1533:       800 0.000000e+00  #000000   NA
##            filter
##    1:  no filter
##    2:  polyester
##    3: Rosco #226
##    4:  no filter
##    5:  polyester
##   ---
## 1529:  polyester
## 1530: Rosco #226
## 1531:  no filter
## 1532:  polyester
## 1533: Rosco #226

fig_sun.cie0 <-
  ggplot(data=sun.eff.cie.spct, aes(x=w.length, y=s.e.irrad, fill=wb.f)) +
  scale_fill_grey() +
  geom_area() +
  labs(x = xlab_nm,
       y = expression(Effective~~spectral~~energy~~irradiance~~(W~m^{-2}~nm^{-1})),
       title = "CIE 1998 erythemal BSWF") +
  facet_grid(filter~.) +
  labs(fill="") +
  xlim(NA, 400) +
  theme_bw() +
  theme(legend.position=c(0.90, 0.9))

fig_sun.cie0

## Warning:  Removed 401 rows containing missing values
(position_stack).
## Warning:  Removed 401 rows containing missing values
(position_stack).
## Warning:  Removed 401 rows containing missing values
(position_stack).
```



There is one warning issued for each panel, as the use of `xlim` discards 400 observations for wavelengths longer than 400 (nm). One should be aware that these are estimated values and in practice stray light reduces the eficiency of the filters for blocking radiation, and the amount of stray light depends on many factors including the relative positions of plants, filter and sun.

A couple of details need to be remembered: the tagging has to be done before row-binding the spectra, as `tag` works only on spectra that have unique values for wavelengths and discards 'repeated' rows if they are present. We use `theme(legend.position=c(0.90, 0.9))` to change where the legend or guide is positioned. In this case, we move the legend to a place within the plotting region. As we are using also `theme_bw()` which resets the legend position to the default, the order in which they are added is significant.

8.12 Task: making a bar plot of effective irradiance

In this task we aim at creating bar plots depicting the contributions of the UVB and UVA bands to the total erythemal effective irradiance in sunlight filtered with different plastic films. First we calculate the effective energy irradiance using the waveband definition for erythemal BSWF (CIE98) separately for the estimated solar spectral irradiance under each filter type.

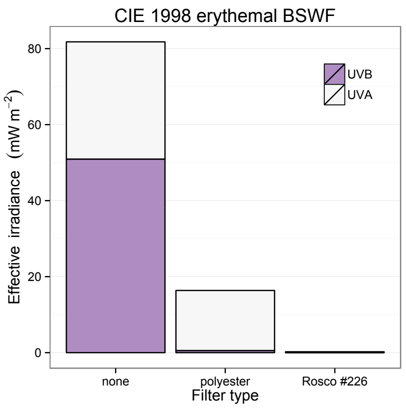
```
cie.nf.irrad <- e_irrad(sun.spct * CIE(),
                        list(UVB(), UVA()))
cie.pe.irrad <- e_irrad(sun.spct * polyester.new.spct * CIE(),
                        list(UVB(), UVA()))
cie.226.irrad <- e_irrad(sun.spct * uv.226.new.spct * CIE(),
                         list(UVB(), UVA()))
```

We assemble a data table by concatenating the irradiance and adding factors for filter type and wave bands. When defining the factors, we use `levels` to make sure that the levels are ordered as we would like to plot them.

```
cie.dt <- data.table(
  cie.irrad = c(cie.nf.irrad, cie.pe.irrad, cie.226.irrad),
  filter = factor(rep(c('none', 'polyester', 'Rosco #226'), c(2,2,2)),
                  levels=c('none', 'polyester', 'Rosco #226')),
  w.band = factor(rep(c('UVB', 'UVA'), 3),
                  levels=c('UVB', 'UVA')) )
```

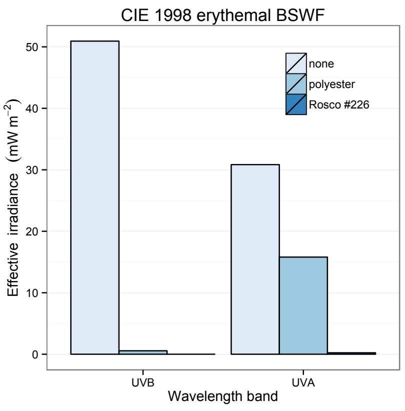
Now we plot stacked bars using `geom_bar`, however as the default `stat` of this geom is not suitable for our data, we specify `stat="identity"` to have the data plotted as is. We set a specific palette for fill, and add a black border to the bars by means of `color="black"`, we remove the grid lines corresponding to the x-axis, and also position the legend within the plotting region.

```
fig_cie_bars0 <- ggplot(data=cie.dt,
                        aes(y = cie.irrad * 1e3,
                            x = filter,
                            fill = w.band)) +
  scale_fill_brewer(palette="PRGn") +
  geom_bar(stat="identity", colour="black") +
  labs(x = "Filter type",
       y = expression(Effective~~irradiance~~~(mW~m^{-2})),
       title = "CIE 1998 erythemal BSWF",
       fill = "") +
  theme_bw(13) +
  theme(legend.position=c(0.85, 0.85)) +
  theme(panel.grid.minor.x=element_blank(),
        panel.grid.major.x=element_blank())

fig_cie_bars0
```



The figure above is good for showing the relative contribution of UVB and UVA radiation to the total effect, and the size of the total effect. On the other hand if we would like to show how much the effective irradiance in the UVB and UVA decreases under each of the filters is better to avoid stacking of the bars, plotting them side by side using `position=position_dodge()`. In addition we swap the aesthetics to which the two factors are linked.

```
fig_cie_bars1 <- ggplot(data=cie.dt,
                        aes(y = cie.irrad * 1e3,
                            x = w.band,
                            fill=filter)) +
  geom_bar(stat="identity",
           position=position_dodge(),
           color="black") +
  scale_fill_brewer() +
  labs(x = "Wavelength band",
       y = expression(Effective~~irradiance~~~(mW~m^{-2})),
       title = "CIE 1998 erythemal BSWF",
       fill = "") +
  theme_bw() +
  theme(legend.position=c(0.80, 0.85)) +
  theme(panel.grid.minor.x=element_blank(),
        panel.grid.major.x=element_blank())

fig_cie_bars1
```



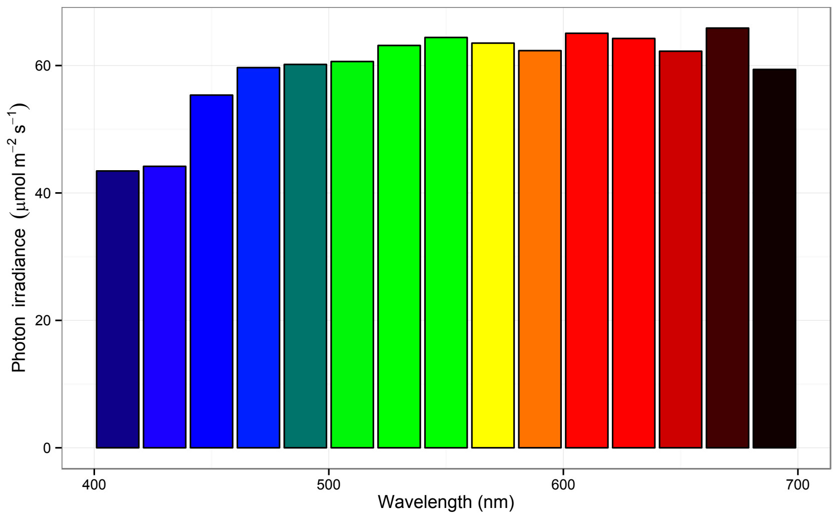
8.13 Task: plotting a spectrum using colour bars

We show now the last example, related to the ones above, but creating a bar plot with more bars. First we calculate photon irradiance for different equally spaced bands within PAR using function `split_bands`. The code is written so that by changing the first two lines you can adjust the output.

```
wl.range <- range(PAR())
num.bands <- 15
many.bands <- split_bands(wl.range, length.out=num.bands)
w.length <- numeric(num.bands)
wb.name <- wb.color <- character(num.bands)

for (i in 1:num.bands) {
  w.length[i] <- midpoint(many.bands[[i]])
  wb.color[i] <- color(many.bands[[i]], type="CMF")
  wb.name[i] <- labels(many.bands[[i]])[["name"]]
}

q.irrad.bands.sun <- q_irrad(sun.spct, many.bands)
q.irrad.sun.dt <- data.table(q.irrad = q.irrad.bands.sun,
                                   w.length = w.length,
                                   wb.color = wb.color,
                                   wb.name = wb.name)
```

Now we can plot the data as bars, filling each bar with the corresponding colour. In this case we plot the bars using a continuous variable, wavelength, for the x-axis.

```
fig_qirrad_bar <- ggplot(data=q.irrad.sun.dt,
                        aes(y = q.irrad * 1e6,
                            x = w.length,
                            fill=as.character(wb.color))) +
  geom_bar(stat="identity",
           color="black") +
  scale_fill_identity(guide="none") +
  labs(x = xlab_nm,
       y = expression(Photon~~irradiance~~(mu*mol~m^{-2}~s^{-1})),
       fill = "") +
  theme_bw()

fig_qirrad_bar
```



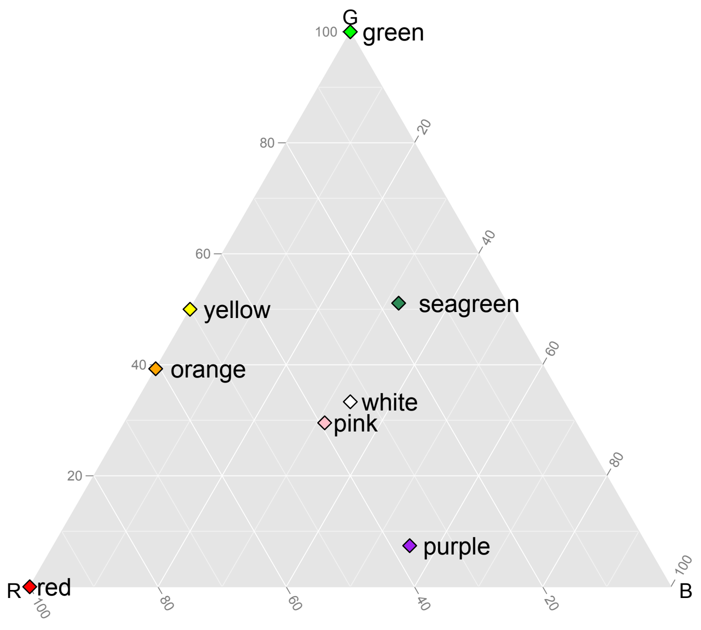
In the case of the example spectrum with equal wavelength steps, one could have directly summed the values, however, the approach shown here is valid for any type of spacing of the values along the wavelength axis, including variable one, like is the case for array spectrometers.

8.14 Task: plotting colours in Maxwell's triangle

8.14.1 Human vision: RGB

Given a color definition, we can convert it to RGB values by means of R's function `col2rgb`. We can obtain a color definition for monochromatic light from its wavelength with function `w_length2rgb` (see section ??), from a waveband with function `color` (see section ??), for a wavelength range with `w_length_range2rgb` (see section ??), and from a spectrum with function `s_e_irrad2rgb` (see section ??). The RGB values can be used to locate the position of any colour on Maxwell's triangle, given a set of chromaticity coordinates defining the triangle. In the first example we use some of R's predefined colors. We use the function `ggtern` from the package of the same name. It is based on `ggplot` and to produce a ternary diagram we need to use `ggtern` instead of `ggplot`. Geoms, aesthetics, stats and faceting function normally in most cases. Of course, being a ternary plot, the aesthetics `x`, `y`, and `z` should be all assigned to variables in the data.

```
colours <- c("red", "green", "yellow", "white",
             "orange", "purple", "seagreen", "pink")
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=23, size=3) +
  geom_text(hjust=-0.2) +
  labs(x = "R", y="G", z="B") + scale_fill_identity()
maxwell.tern
```



8.15 Honey-bee vision: GBU

In this case we start with the spectral responsiveness of the photoreceptors present in the eyes of honey bees. Bees, as humans have three photoreceptors, but instead of red, green and blue (RGB), bees see green, blue and UV-A (GBU). To plot colours seen by bees one can still use a ternary plot, but the axes represent different photoreceptors than for humans, and the colour space is shifted towards shorter wavelengths.

The calculations we will demonstrate here, in addition are geared to compare a background to a foreground object (foliage vs. flower). We have followed xxxxx **chitka?** in this example, but be aware that calculations presented in this reference do not match the equations presented. In the original published example, the calculations have been simplified by leaving out $\delta\lambda$. Although not affecting the final result for their example, intermediate results are different (wrong?). We have further generalized the calculations and equations to make the calculations also valid for spectra measured using $\delta\lambda$ that itself varies along the wavelength axis. This is the usual situation with array spectrometers, nowadays frequently used when measuring reflectance.

The assessment of the perceived ‘colour difference’ between background and foreground objects requires taking into consideration several spectra: the incident ‘light’ spectrum, the reflectance spectra of the two objects, and the sensitivity spectra of three photoreceptors in the case of trichromic vision. In addition to these data, we need to take into consideration the shape of the dose response of the photoreceptors.

```r
try(detach(package:photobiologygg))
try(detach(package:ggtern))
try(detach(package:ggplot2))
try(detach(package:gridExtra))
try(detach(package:photobiologyFilters))
try(detach(package:photobiologyWavebands))
try(detach(package:photobiology))
```

Part III

Catalogue of data sources

Part IV

Data acquisition and modelling

Bibliography

Aphalo, P. J., A. Albert, L. O. Björn, L. Ylianttila, F. L. Figueroa and P. Huovinen (2012). ‘Introduction’. In: *Beyond the Visible: A handbook of best practice in plant UV photobiology*. Ed. by P. J. Aphalo, A. Albert, L. O. Björn, A. R. McLeod, T. M. Robson and E. Rosenqvist. 1st ed. COST Action FA0906 “UV4growth”. Helsinki: University of Helsinki, Department of Biosciences, Division of Plant Biology. Chap. 1, pp. 1–33. ISBN: ISBN 978-952-10-8363-1 (PDF), 978-952-10-8362-4 (paperback). URL: `http://hdl.handle.net/10138/37558` (cit. on p. 31).

Caldwell, M. M. (1971). ‘Solar UV irradiation and the growth and development of higher plants’. In: *Photophysiology*. Ed. by A. C. Giese. Vol. 6. New York: Academic Press, pp. 131–177. ISBN: 012282606X (cit. on p. viii).

Green, A. E. S., T. Sawada and E. P. Shettle (1974). ‘The middle ultraviolet reaching the ground’. In: *Photochemistry and Photobiology* 19, pp. 251–259. DOI: `10.1111/j.1751-1097.1974.tb06508.x` (cit. on p. viii).

Sliney, D. H. (2007). ‘Radiometric quantities and units used in photobiology and photochemistry: recommendations of the Commission Internationale de L’Eclairage (International Commission on Illumination)’. In: *Photochemistry and Photobiology* 83, pp. 425–432. DOI: `10.1562/2006-11-14-RA-1081` (cit. on p. vii).

Thimijan, R. W., H. R. Carns and L. E. Campbell (1978). *Final Report (EPA-IAG-D6-0168): Radiation sources and related environmental control for biological and climatic effects UV research (BACER)*. Tech. rep. Washington, DC: Environmental Protection Agency (cit. on p. viii).

Part V

Appendixes

Appendix A
Build information

```
Sys.info()

##                       sysname
##                     "Windows"
##                       release
##                      "7 x64"
##                       version
## "build 7601, Service Pack 1"
##                      nodename
##                      "MUSTI"
##                       machine
##                     "x86-64"
##                         login
##                     "aphalo"
##                          user
##                     "aphalo"
##                effective_user
##                     "aphalo"
```

```
sessionInfo()

## R version 3.1.2 (2014-10-31)
## Platform: x86_64-w64-mingw32/x64 (64-bit)
##
## locale:
## [1] LC_COLLATE=English_United Kingdom.1252
## [2] LC_CTYPE=English_United Kingdom.1252
## [3] LC_MONETARY=English_United Kingdom.1252
## [4] LC_NUMERIC=C
## [5] LC_TIME=English_United Kingdom.1252
##
## attached base packages:
## [1] grid      methods   tools     stats
## [5] graphics  grDevices utils     datasets
## [9] base
##
## other attached packages:
## [1] scales_0.2.4     plyr_1.8.1
## [3] splus2R_1.2-0    proto_0.3-10
## [5] data.table_1.9.4 lubridate_1.3.3
## [7] stringr_0.6.2    knitr_1.8
##
## loaded via a namespace (and not attached):
##  [1] chron_2.3-45
##  [2] colorspace_1.2-4
##  [3] digest_0.6.4
##  [4] evaluate_0.5.5
##  [5] formatR_1.0
##  [6] ggmap_2.3
##  [7] ggplot2_1.0.0
##  [8] ggtern_1.0.3.2
##  [9] gridExtra_0.9.1
## [10] gtable_0.1.2
## [11] highr_0.4
## [12] labeling_0.3
## [13] lattice_0.20-29
## [14] mapproj_1.2-2
## [15] maps_2.3-9
## [16] MASS_7.3-35
## [17] memoise_0.2.1
## [18] munsell_0.4.2
## [19] photobiology_0.4.8
## [20] photobiologyFilters_0.1.13
## [21] photobiologygg_0.1.14
## [22] photobiologyLEDs_0.1.4
## [23] photobiologyWavebands_0.1.0
## [24] png_0.1-7
## [25] RColorBrewer_1.0-5
## [26] Rcpp_0.11.3
## [27] reshape2_1.4
## [28] RgoogleMaps_1.2.0.6
## [29] rjson_0.2.15
## [30] RJSONIO_1.3-0
## [31] sp_1.0-16
```