R for Photobiology

A handbook

Pedro J. Aphalo
and
Andreas Albert

Git: tag 'none', committed with hash 21da9cc on 2014-11-27 21:06:24 +0200
by Pedro J. Aphalo
PDF created 27th November 2014

Contents

Preface

This is just a very early draft of a handbook 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

CHAPTER 1

Introduction

Abstract

In this chapter we explain the physical basis of optics and photochemistry.

1.1 Radiation and molecules

CHAPTER 2

Optics

Abstract

In this chapter we explain how to .

2.1 Task:

CHAPTER 3

Photochemistry

Abstract

In this chapter we explain how to .

3.1 Task:

CHAPTER 4

Software

Abstract

In this chapter we describe the software we used to run the code examples and typeset this handbook, and how to install it.

4.1 Task:

4.2 Introduction

The software used for typesetting this handbook and developing the r4photobiology suite is free and open source. All of it is available for the most common operating systems (Unix including OS X, Linux and its variants, and Windows). It is also possible to run everything described here on a Linux server running the server version of RStudio, and access the server through a web browser.

For just running the examples in the handbook, you would need only to have R installed. That would be enough as long as you also have a text editor available. This is possible, but does not give a very smooth workflow for data analyses which are beyond the very simple. The next stage is to use a text editor which integrates to some extent with R, but still this is not ideal, specially for writing packages or long scrips. Currently the best option is to use the integrated development environment (IDE) called 'RStudio'. This is an editor, but tightly integrated with R. Its advantages are especially noticeable in the case of errors and 'debugging'. We also use a LaTeX for typesetting. Is what we used for the first handbook (Aphalo et al. 2012), and what we routinely for reporting data analyses, and that PJA also uses for all 'overhead' slides he writes for lectures. You, do not need to go this far to be able to profit from R and our suite, but the set up we will describe here, is what we currently use, it is by far

the best one we have encountered in 18 years of using and teaching how to use R.

We will not give software installation instructions in this handbook, but will keep a web page with up-to-date instructions. In the following sections we briefly describe the different components of a full and comfortable working environment, but there many alternatives and the only piece that you cannot replace is R itself.

4.3 The different pieces

4.3.1 R

You will not be able to profit from this handbook's 'Cook Book' part, unless you have access to R. R (also called Gnu S) is both the name of a software system, and a dialect of the language S. The language S, although designed with data analysis and statistics in mind, is a computer language that is very powerful in its own way. It allows object oriented programming. Being based in a programming language, and being able to call and being called by programs and subroutine libraries written in several other programming languages, makes it easily extensible.

R has a well defined mechanism for "add-ons" called packages, that are kept in the computer where R is running, in disk folders that conform the library. There is a standard mechanism for installing packages, that works across operating systems (OSs) and computer architectures. There is also a Comprehensive R Archive Network (CRAN) where publicly released versions of packages are kept. Packages can be installed and updated from CRAN and similar repositories directly from within R.

If you are not familiar with R, please, go through the Appendixes A, B, B, and D, and/or learn from some of the books listed in Appendix **??**, before delving into our 'Cook Book'.

4.3.2 RStudio

RStudio exists in two versions with identical user interface: a desktop version and a server version. The server version can be used remotely through a web browser. It can be for example run in the 'cloud', for example, as an AWS instance (Amazon Web Services) quite easily and cheaply, or on one's own server hardware.

4.3.3 Version control: Git and Subversion

Version control systems help with keeping track of the history of software development, data analysis, or even manuscript writing. They make it possible for several programmers, data analysts, authors and or editors to work on the same files in parallel and then merge their edits. They also allow easy transfer of whole 'projects' between computers. Git is very popular, and Github and Bitbucket are popular hosts for repositories. Git itself is free software, and can

be also run locally, or as one's own private server, either as an AWS instance or on other hosting service, or on your own hardware.

4.3.4 C++ compiler

Although R is an interpreted language, a few functions in our suite are written in C++ to achieve better performance. On OS X and Windows, the normal practice is to install binary packages, which are ready compiled. In other systems like Linux and Unix it is the normal practice to install source packages that are compiled at the time of installation.

4.3.5 LaTeX

LaTeX is built on top of TeX. TeX code and features were 'frozen' (only bugs are fixed) long ago. There are currently a few 'improved' derivatives: pdfTeX, XeTeX, and LuaTeX. Currently the most popular TeX in western countries is pdftex which can directly output PDF files. XeTeX can handle text both written from left to right and right to left, even in the same document, and is the most popular TeX engine in China and other Asian countries.

CHAPTER 5

Photobiology R packages

Abstract

In this chapter we describe the suite of R packages for photobiological calculations 'r4photobiology', and explain how to install them.

5.1 The design of the user interface

The design of the 'high level' interface is based on the idea of achieving simplicity of use by hiding the computational difficulties and exposing objects, functions and operators that map directly to physical concepts. Computations and plotting of spectral data centers on two types of objects: *spectra* and *wavebands*. Al spectra have in common that all observations are referenced to a wavelength value, there are different types spectral objects, e.g. for light sources and responses. Waveband objects include much more than information about a range of wavelengths, they can also include information about a transformation of the spectral data, like a biological spectral weighting function (BSWF). In addition to functions for calculating summary quantities like irradiance from spectral irradiance, the packages define operators for spectra and wavebands. The use of operators simplifies the syntax and makes the interface easier to use.

```
e_irrad(sun.spct * polyester.new.spc, CIE())
```

Is all what is needed to obtain the CIE98-weigthed energy irradiance simulating the effect of a polyester filter on the example solar spectrum, which of course, can be substituted by other spectral data.

When we say that we hide the computational difficulties what we mean, is that in the example above, the data for the two spectra do not need to be

available at the same wavelengths values, and the BSWF is defined as a function. Interpolation of the spectral data and calculation of weighting factors takes place automatically and invisibly. All functions and operators function without error with spectra with varying (even arbitrarily and randomly varying) wavelength steps. Integration is always used rather than summation for summarizing the spectral data.

There is lower layer of functions, used internally, but also exported, which allow improved performance at the expense of more complex scripts and commands. This user interface is not meant for the casual user, but for the user who has to analyse thousands of spectra and uses scripts for this. For such users performance is the main concern rather than easy of use and easy to remember syntax. Also these functions handle any wavelength mismatch by interpolation before applying operations or functions.

The suite also includes data for the users to try options and ideas, and helper functions for plotting spectra using other R packages available from CRAN, in particular `ggplot2`. There are some packages, not part of the suite itself, for data acquisition from Ocean Optics spectrometers, and application of special calibration and correction procedures to those data. A package will provide an interface to the TUV model to allow easy simulation of the solar spectrum.

5.2 The suite

The suite consists in several packages. The main package is `photobiology` which contains all the generally useful functions, including many used in the other, more specialized, packages (Table 5.1).

Although by default functions expect spectral data on energy units, this is just a default that can be changed by setting the parameter `unit.in = "photon"`. Across all data sets and functions wavelength vectors have name `w.length`, spectral (energy) irradiance `s.e.irrad`, photon spectral irradiance `s.q.irrad`[1], absorbance ($\log_{10}$-based) `A`, transmittance (fraction of one) `Tfr`, transmittance (%) `Tpc`, reflectance (fraction of one) `Rfr`, and reflectance (%) `Rpc`.

Wavelengths should always be in nm, and when conversion between energy and photon based units takes place no scaling factor is used (an input in $\mathrm{W\,m^{-2}\,nm^{-1}}$ yields an output in $\mathrm{mol\,m^{-2}\,s^{-1}\,nm^{-1}}$ rather than $\mathrm{\mu mol\,m^{-2}\,s^{-1}\,nm^{-1}}$).

The suite is still under active development. Even those packages marked as 'stable' are likely to acquire new functionality. By stability, we mean that we hope to be able to make most changes backwards compatible, in other words, we hope they will not break existing user code.

5.3 `r4photo` repository

I have created a small repository for the packages. This repository follows the CRAN folder structure, so now package installation can be done using

[1] `q` derives from 'quantum'.

Table 5.1: Packages in the r4photobiology suite. Packages not yet released are highlighted with a red bullet •, and those at 'beta' stage with a yellow bullet •, those relatively stable with a green bullet •.

	Package	Type	Contents
•	photobiology	funs + classes	basic functions, class definitions, class methods and example data
•	photobiologyWavebands	definitions	quantification of radiation
•	photobiologySun	data	spectral data for solar radiation
•	photobiologyLamps	data	spectral data for lamps
•	photobiologyLEDs	data	spectral data for LEDs
•	photobiologyFilters	data	transmittance data for filters
•	photobiologySensors	data	response data for broadband sensors
•	photobiologyPhy	funs + data	phytochromes
•	photobiologyCry	funs + data	cryptochromes
•	photobiologyPhot	funs + data	phototropins
•	photobiologyUVR8	funs + data	UVR8
•	photobiologygg	functions	extensions to package `ggplot2`
•	rTUV	funs + data	TUV model interface
•	rOmniDriver	functions	control of Ocean Optics spectrometers

just the normal R commands. This means that dependencies are installed automatically, and that automatic updates are possible. The build most suitable for the current system and R version is also picked automatically if available. It is normally recommended that you do installs and updates on a clean R session (just after starting R or RStudio).For easy installation and updates of packages, the r4photo repository can be added to the list of repositories that R knows about.

Whether you use RStudio or not it is possible to add the r4photo repository to the current session as follows, which will give you a menu of additional repositories to activate:

```
setRepositories(graphics = getOption("menu.graphics"),
                ind = NULL,
                addURLs = c(r4photo =
                      "http://www.mv.helsinki.fi/aphalo/R"))
```

If you know the indexes in the menu you can use this code, where '1' and '6' are the entries in the menu in the command above.

```
setRepositories(graphics = getOption("menu.graphics"),
                ind = c(1, 6),
                addURLs = c(r4photo =
                      "http://www.mv.helsinki.fi/aphalo/R"))
```

Be careful not to issue this command more than once per R session, otherwise the list of repositories gets corrupted by having two repositories with the same name.

Easiest is to create a text file and name it '`.Rprofile`'. The commands above (and any others you would like to run at R start up) should be included, but with the addition that the package names for the functions need to be prepended. The minimum needed is.

```
utils::setRepositories(graphics = getOption("menu.graphics"),
                       ind = c(1, 6),
                       addURLs = c(r4photo =
                             "http://www.mv.helsinki.fi/aphalo/R"))
```

The `.Rprofile` file located in the current folder is sourced at R start up. It is also possible to have such a file affecting all of the user's R sessions, but its location is operating system dependent, it is in most cases the what the OS considers the current user's *HOME* directory or folder (e.g. 'My Documents' in recent versions of MS-Windows). If you are using RStudio, after setting up this file, installation and updating of the packages in the suite can take place exactly as for any other package archived at CRAN.

The commands and examples below can be used at the R prompt and in scripts whether RStudio is used or not.

After adding the repository to the session, it will appear in the menu when executing this command:

```
setRepositories()
```

and can be enabled and disabled.

In RStudio, after adding the r4photo repository as shown above, the photobiology packages can be installed and uninstalled through the normal RStudio menus and dialogues, and will listed after typing the first few characters of their names. For example when you type 'photob' in the packages field, all the packages with names starting with 'photob' will be listed.

They can be also installed at the R command prompt with the following command:

```
install.packages(c("photobiologyAll", "photobiologygg"))
```

and updated with:

```
update.packages()
```

The added repository will persist only during the current R session. Adding it permanently requires editing the R configuration file, as discussed above. Take into consideration that .Rprofile is read by R itself, and will take effect whether you use RStudio or not. It is possible to have a user wide .Rprofile file, and a different one on those folders needing different settings. There many options that can be modified by means of commands in the .Rprofile file.

5.4 How to install the packages

The examples given in this page assume that 'r4photo' is not in the list of repositories known to the current R session. See the section 5.3 on the r4photo repository for a better alternative to the approach given here. We mention these

other commands because they may be useful in cases when the user does not have write access to his/hers home directory, or just wants to try the packages.

To install the latest version of one package (photobiology used as example) you just need to indicate the repository. However this simple command will only install the dependencies between the different photobiology packages.

```
install.packages("photobiology",
                 repos = "http://www.mv.helsinki.fi/aphalo/R")
```

To update what is already installed, this command is enough (even if the packages have been installed manually before):

```
update.packages(repos = "http://www.mv.helsinki.fi/aphalo/R")
```

The best way to install the packages is to specify both the r4photo repository and a normal CRAN repository, then all dependencies will be automatically installed. The package photobiolgyAll just loads and imports all the packages in the suite, except for photobiolygg. Because of this dependency all the packages are installed unless already installed by issuing this command.

```
install.packages(c("photobiologyAll", "photobiologygg"),
         repos = c(r4photo =
                     "http://www.mv.helsinki.fi/aphalo/R",
                   CRAN =
                     "http://cran.rstudio.com"))
```

This example also shows how one can use an array of package names (in this example all currently available “photobiology” packages) in the call to the function install.packages, this is useful if you want to install only a subset of the files, or if you want to make sure that any older install of the packages is overwritten:

```
photobiology_packages <- c("photobiology",
    "photobiologyWavebands",
    "photobiologyCry", "photobiologyPhy",
    "photobiologyLamps", "photobiologyLEDs",
    "photobiologySun", "photobiologygg",
    "photobiologyFilters",  "photobiologySensors")

install.packages(photobiology_packages,
         repos = c(r4photo =
                     "http://www.mv.helsinki.fi/aphalo/R",
                   CRAN =
                     "http://cran.rstudio.com"))
```

The commands above install all packages in the suite and all their dependencies from CRAN if needed. The following command will update all the packages currently installed (if new versions are available) and install any new dependencies.

```
update.packages(repos =
                  c(r4photo =
                      "http://www.mv.helsinki.fi/aphalo/R",
                    CRAN =
                      "http://cran.rstudio.com"))
```

The instructions above should work under Windows as long as you have a supported version of R (3.0.0 or later) because I have built suitable binaries, under other OSs you may need to add type="source" unless this is already the default. We will try to build OS X binaries for Mac so that installation is easier. Meanwhile if installation fails try adding type="source" to the commands given above. For example the first one would become:

```
install.packages("photobiology",
                 repos = "http://www.mv.helsinki.fi/aphalo/R",
                 type="source")
```

When using type=”source” you may need to install some dependencies like the splus2R package beforehand from CRAN if building it from sources fails.

Part II

Cookbook of calculations

CHAPTER 6

Radiation physics

Abstract

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

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(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)
```

6.2 Introduction

6.3 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):

$$L_s(\lambda, T) = \frac{2hc^2}{\lambda^5} \cdot \frac{1}{e^{(hc/k_B T\lambda)} - 1} \tag{6.1}$$

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

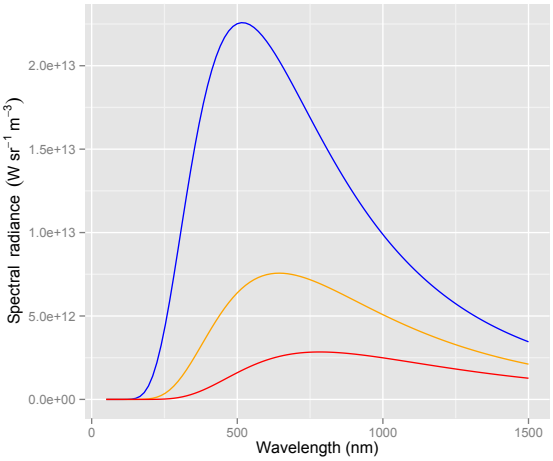
We aware that if two vectors are supplied, then the elements in each one are matched and recycled[1]:

[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.

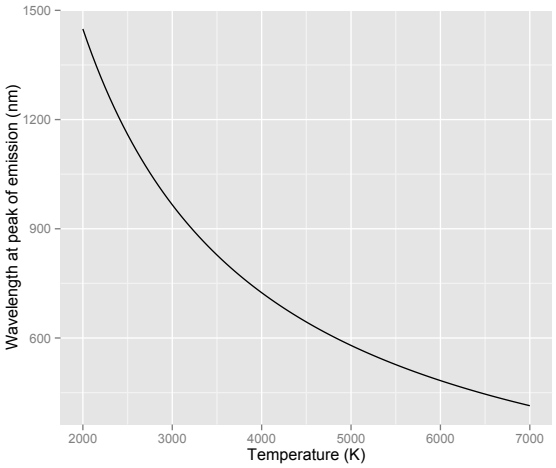
$$\lambda_{max} \cdot T = 2.898 \times 10^6 \,\mathrm{nm\,K} \tag{6.2}$$

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 7

Astronomy

Abstract

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

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

7.2 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
```

7.3 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
```

7.4 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 21:56:23 EET"
##
## $azimuth
## [1] 322.1188
##
## $elevation
## [1] -44.41485
##
## $diameter
## [1] 0.5404017
##
## $distance
## [1] 0.9866734
```

7.5 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,
```

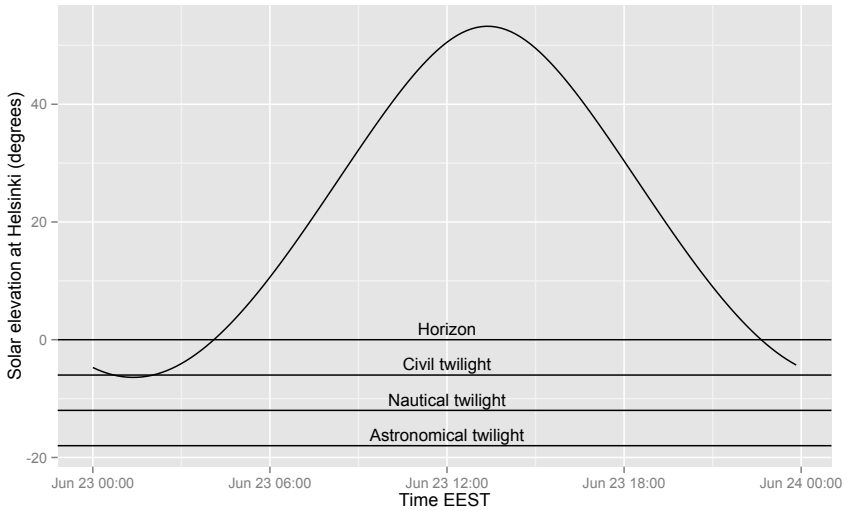
```
                              elevation = elevations,
                              location = "Helsinki",
                              lon = geo_code_He[["lon"]],
                             lat = geo_code_He[["lat"]])
```

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")
```



7.6 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")
```

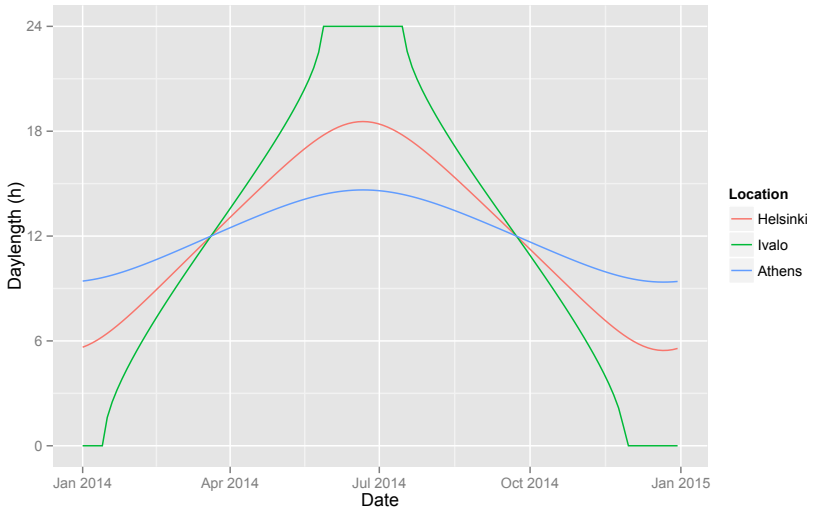
To calculate the length of each day, we need to use an explicit loop as function `day_night` is not vectorized. We repeat the calculations for three locations at different latitudes, then row bind the data frames into a single data frame. Each individual data frame contains information to identify the 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 8

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.

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

8.2 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

8.3 Spectra

8.3.1 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.

8.3.2 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 8.1 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.

8.3.3 What operators are available for operations between spectra?

All operations with spectral objects affect only the required components listed in Table 8.1, 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 8.1: 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	$W\,m^{-2}\,nm^{-1}$
	—	s.q.irrad	—	$mol\,m^{-2}\,s^{-1}\,nm^{-1}$
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
```

8.3.4 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
```

8.3.5 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
```

8.3.6 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`.

8.3.7 Examples

Package `photobiologyFilters` 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 15. 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.

8.3.8 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.

8.3.9 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.

8.3.10 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
```

8.3.11 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.

8.3.12 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))
```

8.3.13 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,

$$q = h \cdot \nu = h \cdot \frac{c}{\lambda} \tag{8.1}$$

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 (8.1) 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

$$q' = h' \cdot \nu = h' \cdot \frac{c}{\lambda} \tag{8.2}$$

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 8.1 while `as_quantum_mol` uses equation 8.2. 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 8.1.

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
```

8.3.14 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 8.1 and 8.2 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
```

8.3.15 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.

8.4 Wavebands

8.4.1 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.

8.4.2 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 9 on page 55, in particular sections 9.4, 9.3, and 9.5 for further examples, and a more in-depth discussion of the creation and use of *unweighted* `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 10 on page 73, in particular sections **??**, **??**, and **??** for further examples, and a more in-depth discussion of the creation and use of *weighted* `waveband` objects.

8.4.3 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`.

8.4.4 Operators and functions

Several functions described in chapters 9, 10, 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
```

8.5 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 9

Unweighted irradiance

Abstract

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

9.1 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)
```

9.2 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 9.6) on page 62 for details).

9.3 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 57

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"
```

9.4 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"
```

9.5 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
```

9.6 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.

$$E_{\lambda_1<\lambda<\lambda_2} = \int_{\lambda_1}^{\lambda_2} E(\lambda)\ \mathrm{d}\ \lambda \tag{9.1}$$

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.

9.7 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 9.1 and 8.2 we obtain:

$$Q_{\lambda_1<\lambda<\lambda_2} = \int_{\lambda_1}^{\lambda_2} E(\lambda) \frac{h' \cdot c}{\lambda} \mathrm{d}\,\lambda \tag{9.2}$$

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 $\lambda_1 = 400\text{nm}$ and $\lambda_1 = 700\text{nm}$. In this example we will use example data for sunlight to calculate $E_{400\,\text{nm}<\lambda<700\,\text{nm}}$. The function used for this task when working with spectral objects is `q_irrad`, returning photon irradiance in $\text{mol}\,\text{m}^{-2}\,\text{s}^{-1}$. 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 $\mu\text{mol}\,\text{m}^{-2}\,\text{s}^{-1}$ 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 9.6 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 9.6). 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.

9.8 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 $\mu\mathrm{mol\,m^{-2}\,s^{-1}}$.

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
```

9.9 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]:

$$\zeta = \frac{Q_{655\text{nm}<\lambda<665\text{nm}}}{Q_{725\text{nm}<\lambda<735\text{nm}}} \tag{9.3}$$

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
```

9.10 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 9.9 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.

9.11 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:

$$\bar{q}' = \frac{Q_{\lambda_1<\lambda<\lambda_2}}{E_{\lambda_1<\lambda<\lambda_2}} \tag{9.4}$$

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 $\mu mol\,J^{-1}$ from spectral data in $W\,m^{-2}\,nm^{-1}$ 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
```

9.12 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 10

Weighted and effective irradiance

Abstract

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

10.1 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)
```

10.2 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 9. In particular it is important that you read sections **??**, 9.7, on the calculation of irradiances from spectral irradiances and sections 9.3, and 9.4 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.

10.3 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"
```

10.4 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 9.3, 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.

10.5 Task: define wavebands that use weighting functions

In sections ?? and 8.4 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.

10.6 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.

10.7 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"
```

10.8 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 to 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 11

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.

11.1 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)
```

11.2 Introduction

11.3 Task: absorbance and transmittance

Transmittance is defined as:

$$\tau(\lambda) = \frac{I}{I_0} = \frac{E(\lambda)}{E_0(\lambda)} = \frac{Q(\lambda)}{Q_0(\lambda)} \tag{11.1}$$

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:

$$A(\lambda) = -\log_{10}\frac{I}{I_0} \tag{11.2}$$

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:

$$A(\lambda) = -\log_{10}\tau(\lambda) \tag{11.3}$$

which in S language is:

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

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

$$\tau(\lambda) = 10^{-A(\lambda)} \tag{11.4}$$

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
```

11.4 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")
```

11.5 Task: spectral transmittance from spectral absorbance

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

11.6 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"
```

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

11.8 Task: combined spectral transmittance of two or more filters

11.8.1 Ignoring reflectance

11.8.2 Considering reflectance

11.9 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 12

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.

12.1 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)
```

12.2 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.

12.3 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"
```

12.4 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"
```

12.5 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))
```

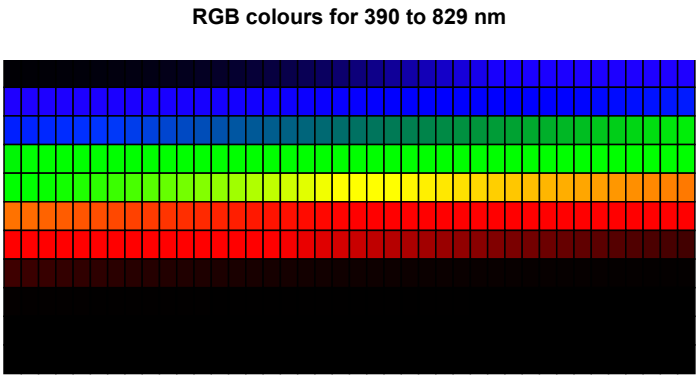
```
## [1] "#59534F"

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.

12.6 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)])
}
```




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

CHAPTER 13

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.

13.1 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

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

13.2 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 D on page 227 before continuing reading the present chapter.

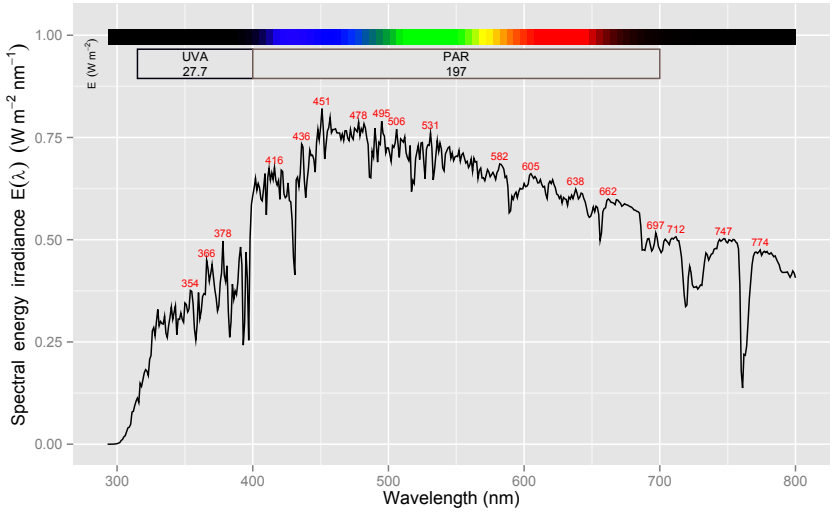
13.3 Task: simple plotting of spectra

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

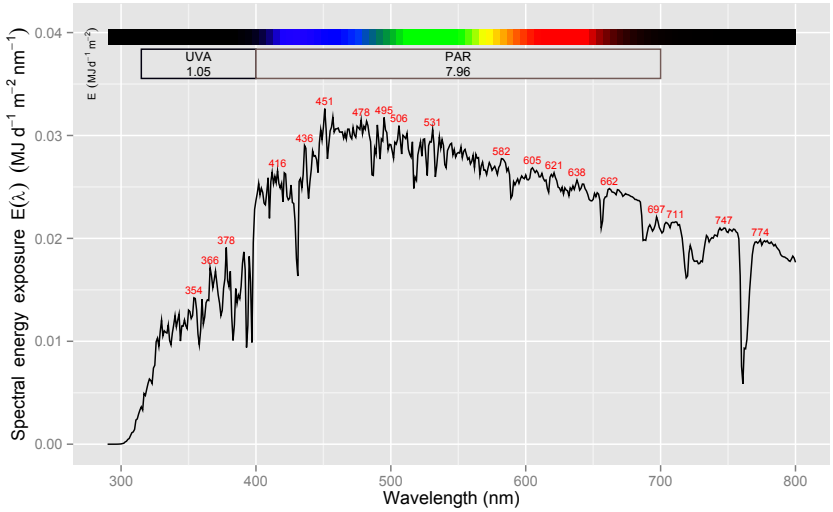
[1] `ggplot2` is feature-frozen, in other words the user interface defined by the functions and their arguments will not change in future versions. Consequently it is a good basis for adding application-specific functionality through separate packages. `ggplot2` uses the *grammar of graphics* for describing 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)
```

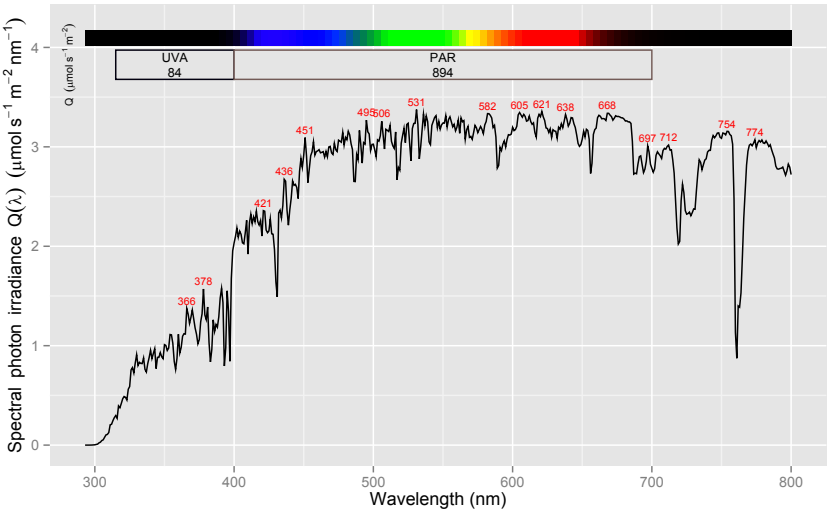



```
plot(sun.daily.spct)
```



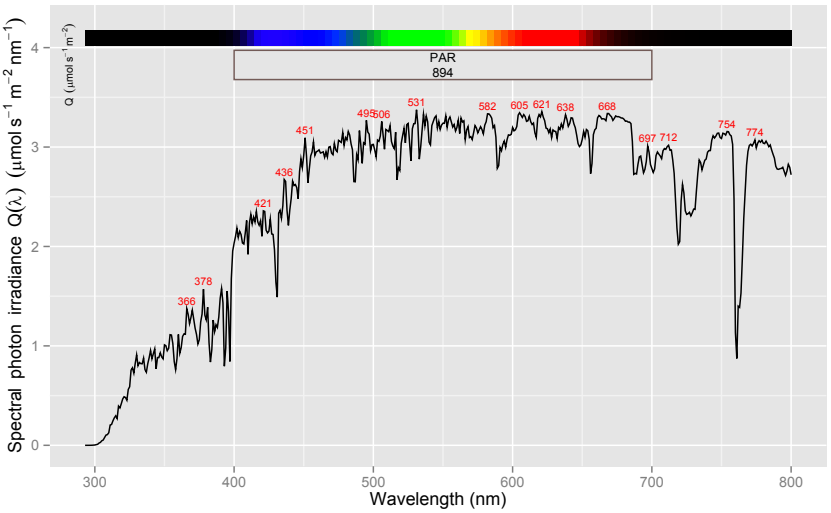
The parameter `unit` can be set to `"photon"` to obtain a plot depicting spectral photon irradiance. This works irrespective of whether the `source.spct` object contains the spectral data in photon or energy units.

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

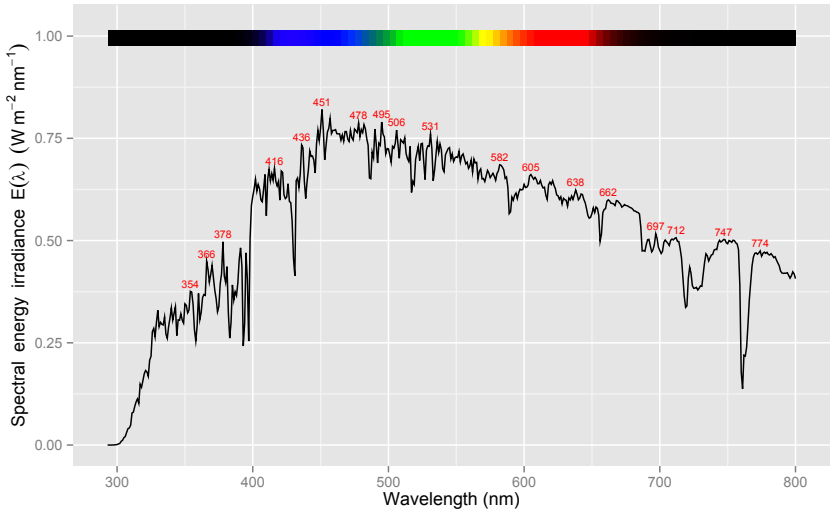


A list of wave bands, or a single wave band, to be used for annotation can be supplied through the `bands` parameter. A `NULL` waveband results in no waveband labels, while the next example shows how to obtain the total irradiance.

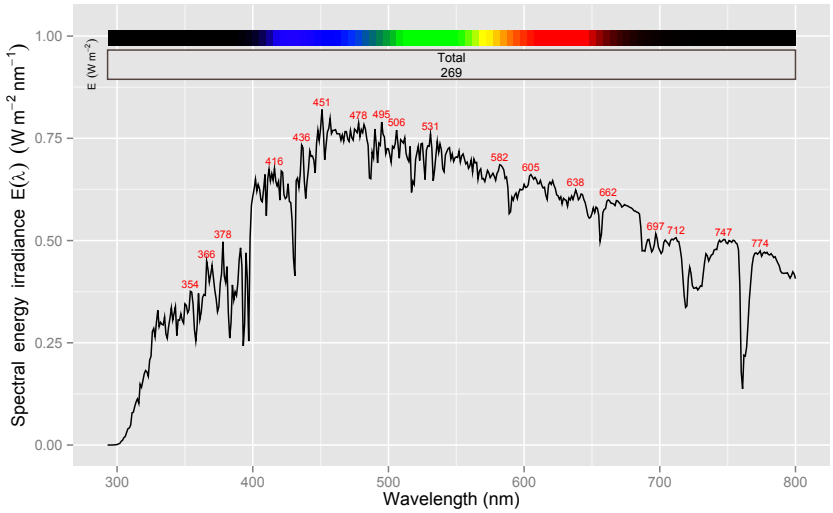
```
plot(sun.spct, bands=PAR(), unit="photon")
```




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

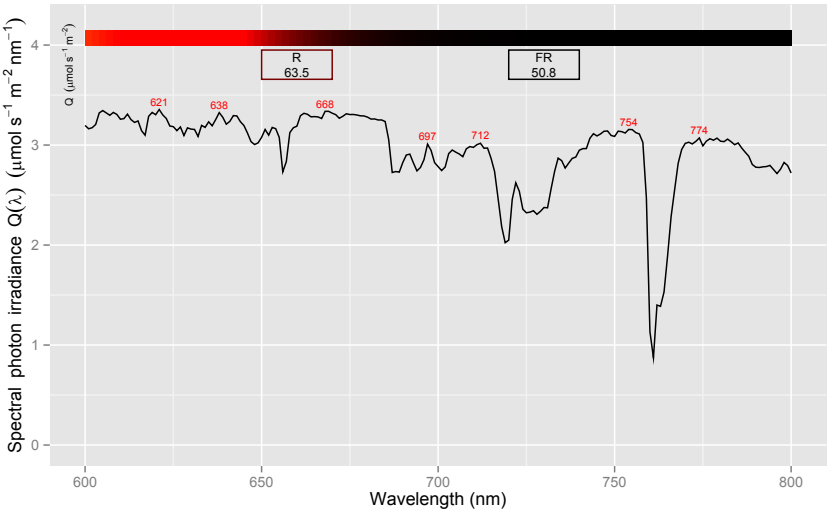



```
plot(sun.spct, bands=waveband(sun.spct))
```



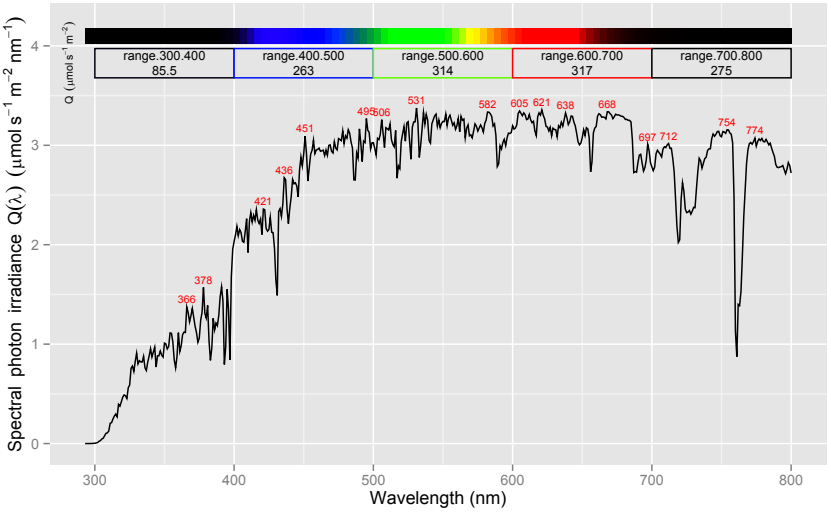
Of course the arguments to these parameters can be supplied in different combinations, and combined with other functions as need. This last example shows how to plot using photon-based units, selecting only a specific region of the spectrum, annotated with the red and far-red photon irradiances, using 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")
```

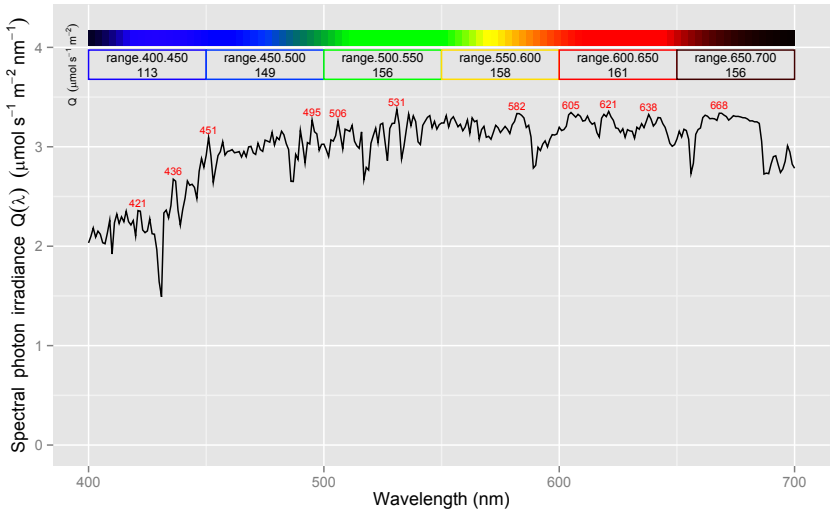


Two final examples show how to annotate a spectrum plot by equal sized wavebands.

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




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



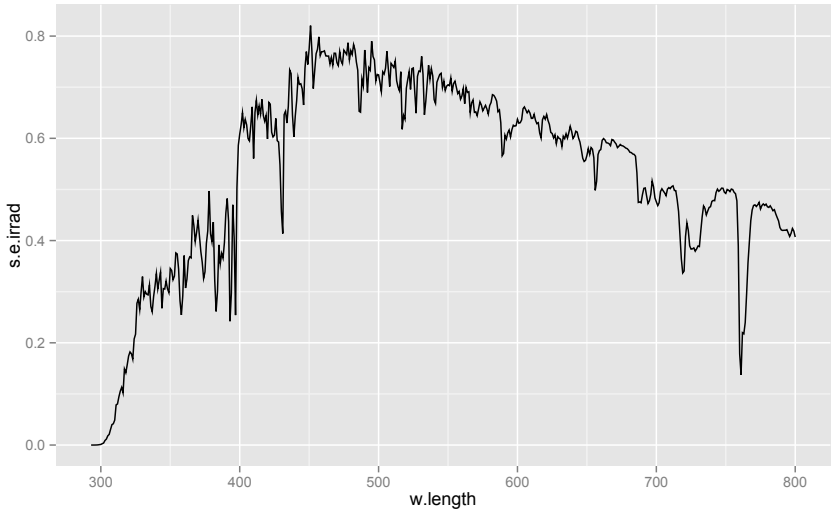
As the current implementation uses annotations rather than a `ggplot` 'statistic', waveband irradiance annotations ignore global aesthetics and facets. If used for simultaneous plotting of several spectra (stored in a single R object), then parameter `bands` should given `NULL` as argument.

13.4 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.

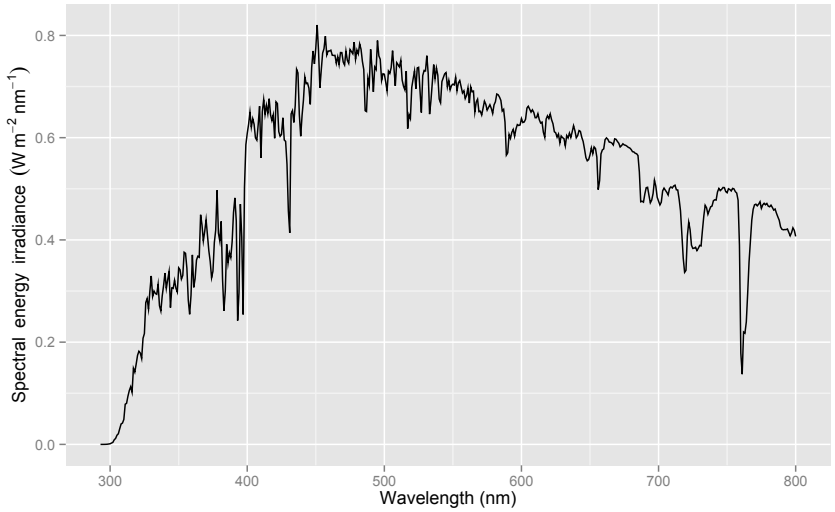
```
fig_sun.e0 <-
  ggplot(data=sun.spct, aes(x=w.length, y=s.e.irrad)) +
  geom_line()

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 D for more details.

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



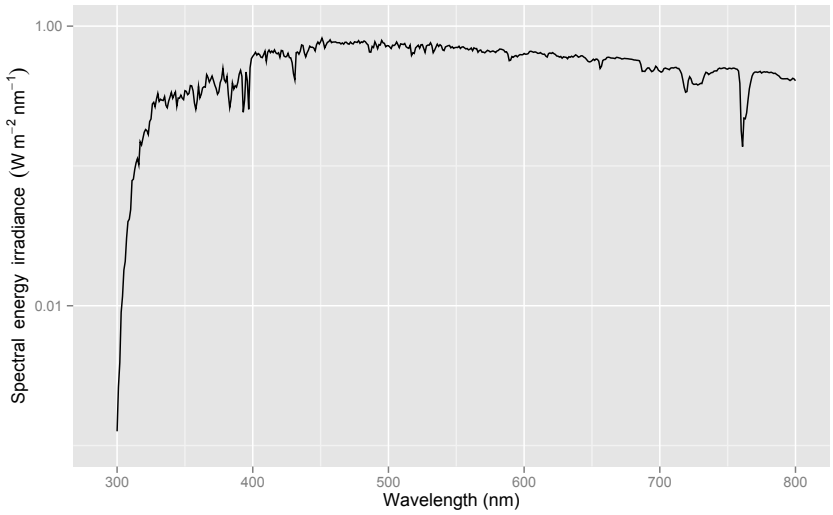
As we are going to re-use the same axis-labels in later plots, it is handy to save their definitions to variables. These definitions will be used in many of this chapter's plots. We also add `atop` to two of the expressions to making shorter versions by setting the spectral irradiance units on a second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

13.5 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.

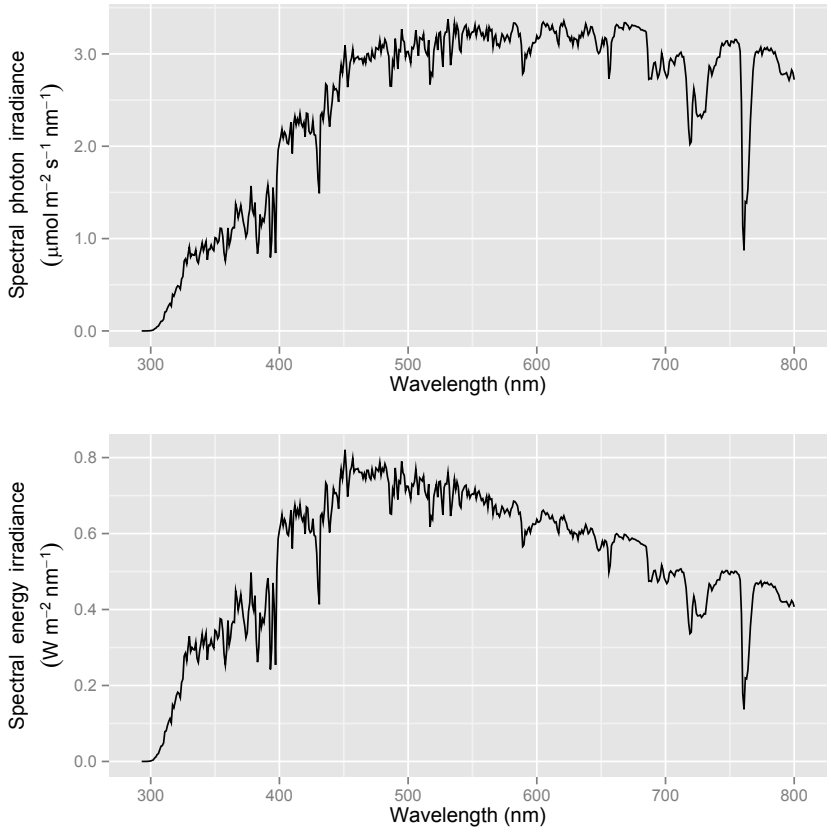
13.6 Task: compare energy and photon spectral units

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.

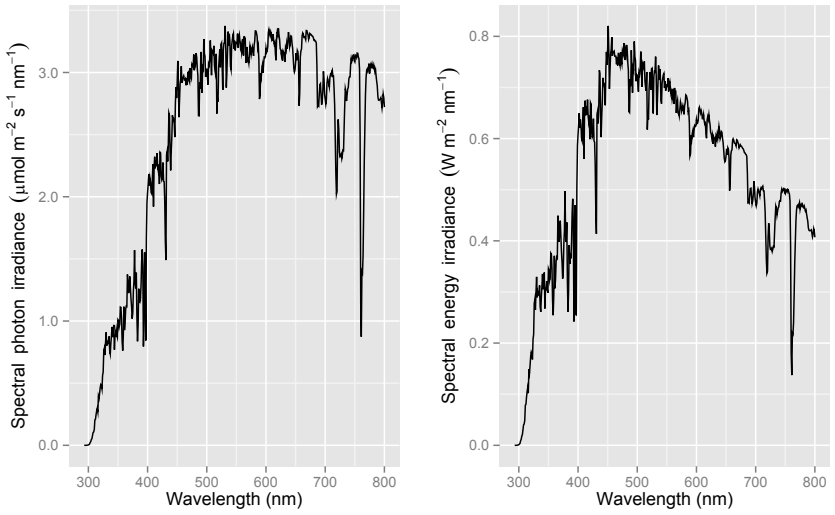
```
num_one_dec <- function(x, ...) {
  format(x, nsmall=1, trim=FALSE, width=4, ...)
  }

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)
```



13.7 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 $\mu mol\,m^{-2}\,s^{-1}\,nm^{-1}$ to 0.82 $\mu mol\,m^{-2}\,s^{-1}\,nm^{-1}$, causes any peaks having a spectral irradiance of less than 0.62 $\mu mol\,m^{-2}\,s^{-1}\,nm^{-1}$ 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.

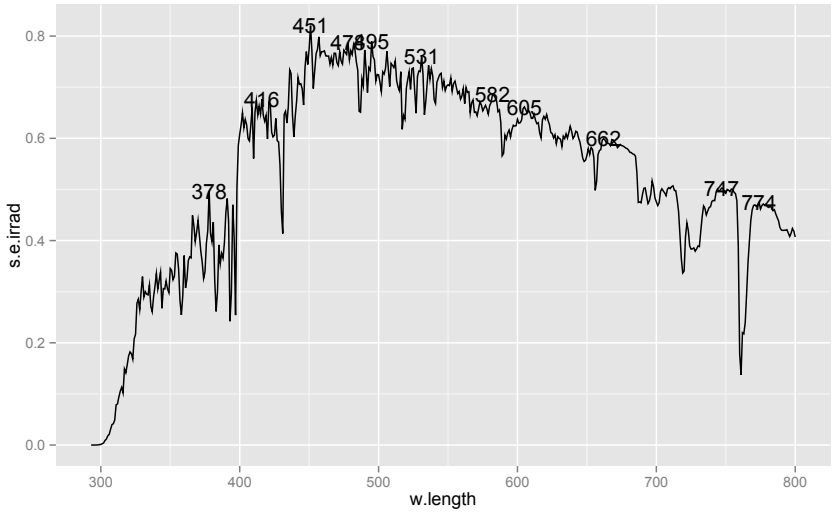
```
head(with(sun.spct,
          get_valleys(w.length, s.e.irrad, span=51)))

##     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 13.8.

13.8 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 13.7. We reuse once more `fig_sun.e` saved in section 13.4.

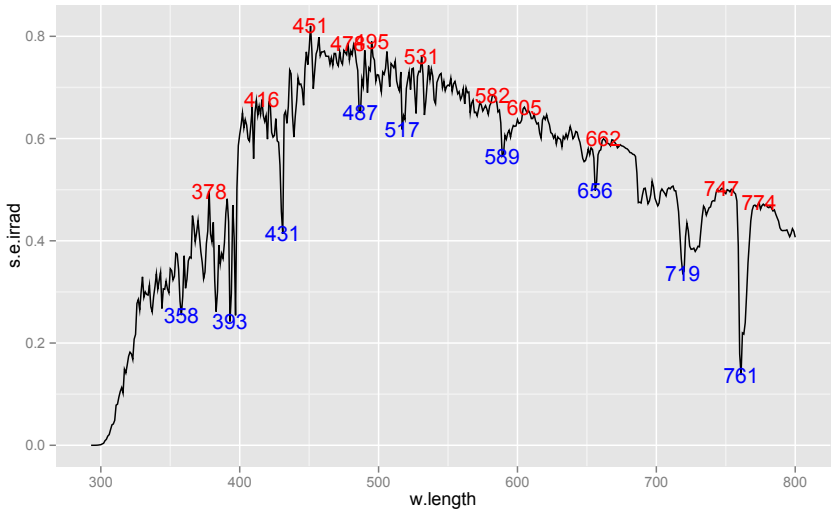
```
fig_sun.e0 + stat_peaks(span=31)
```



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.

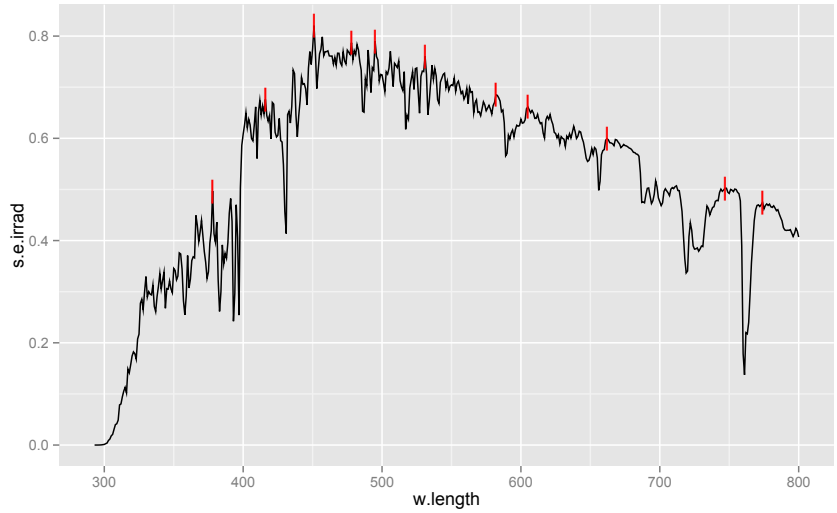
We can change aesthetics, for example the colour:

```
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.

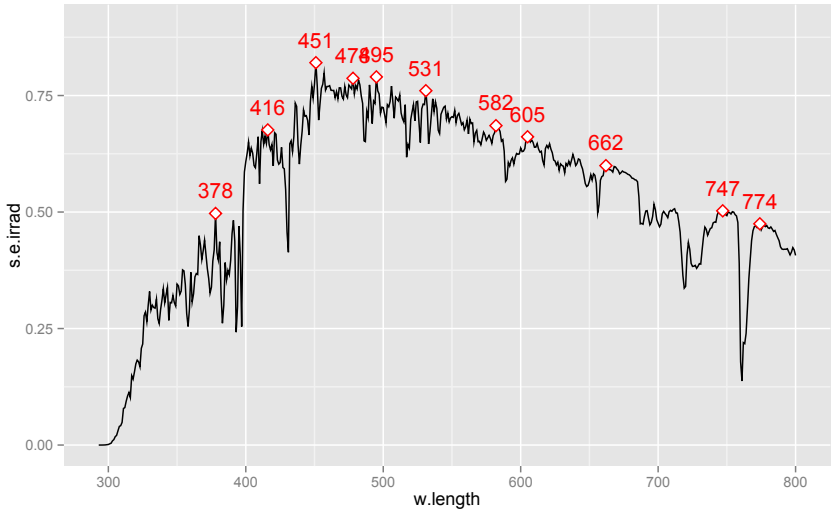
```
fig_sun.e0 +
  stat_peaks(colour="red", geom="point",
             shape="|", size=6, span=31)
```



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.

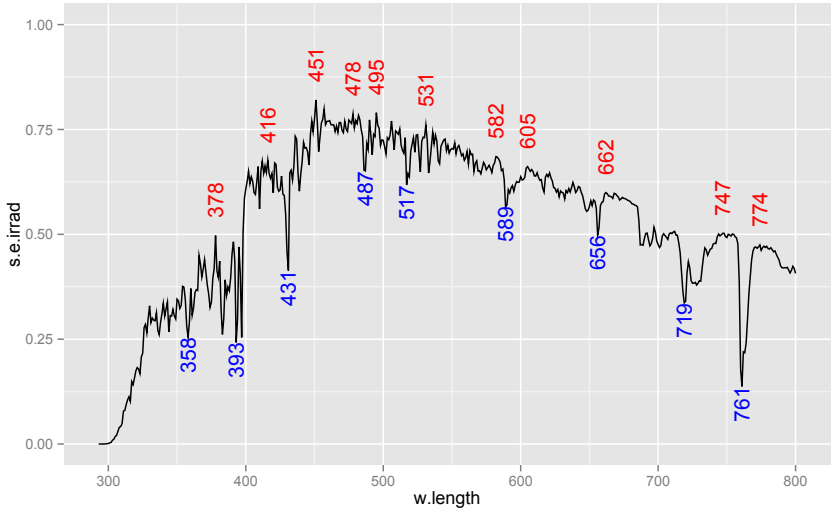
```
fig_sun.e0 +
  stat_peaks(colour="red", geom="point", shape=23,
             fill="white", size=3, span=31) +
  stat_peaks(colour="red", vjust=-1, span=31) +
  expand_limits(y=0.9)
```

[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)
```



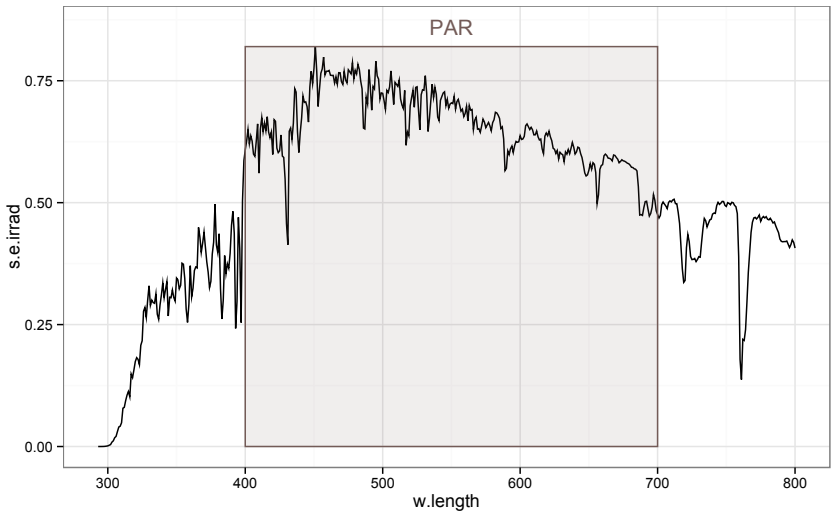
See section **??** in chapter 14 for an example these stats together with facets.

13.9 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`.

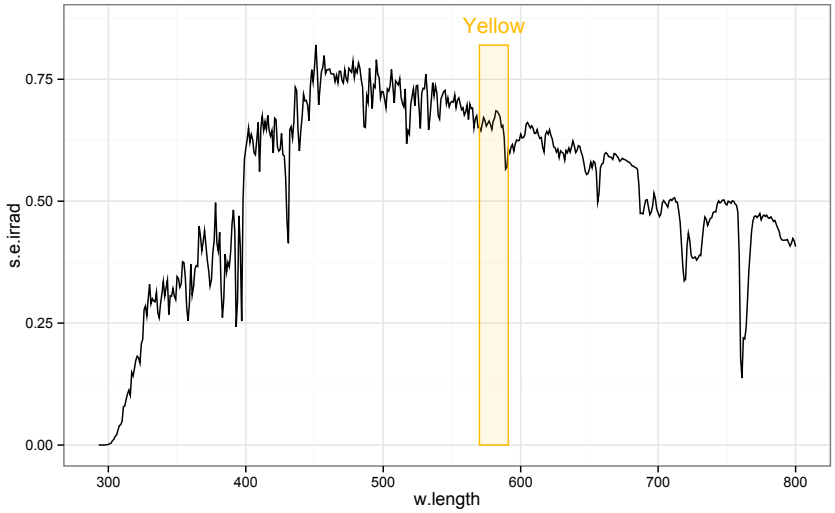
Here is an example for PAR using defaults, and with arguments supplied only for parameters with no defaults. The example does the annotation using two different 'geoms', `"rect"` for marking the region, and `"text"` for the labels.

```
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.

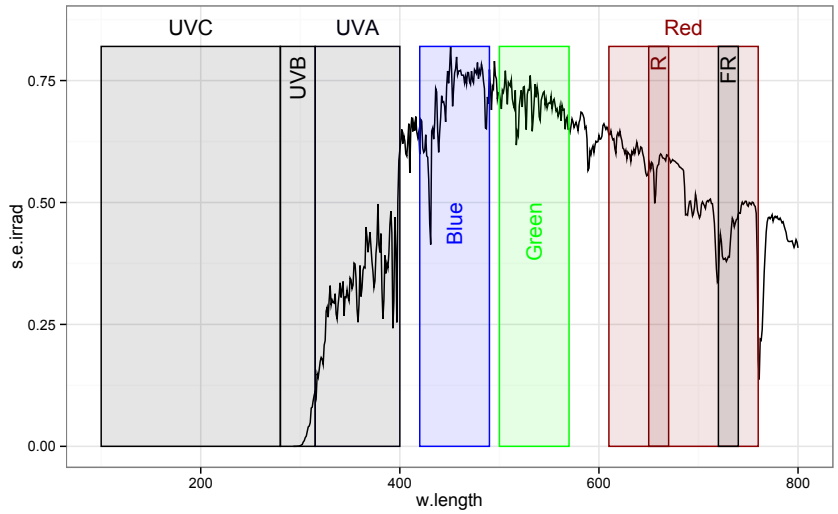
```
figvl <- fig_sun.e0 + annotate_waveband(Yellow(), "rect", ymax=0.82) +
                      annotate_waveband(Yellow(), "text", y=0.86)

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 134.

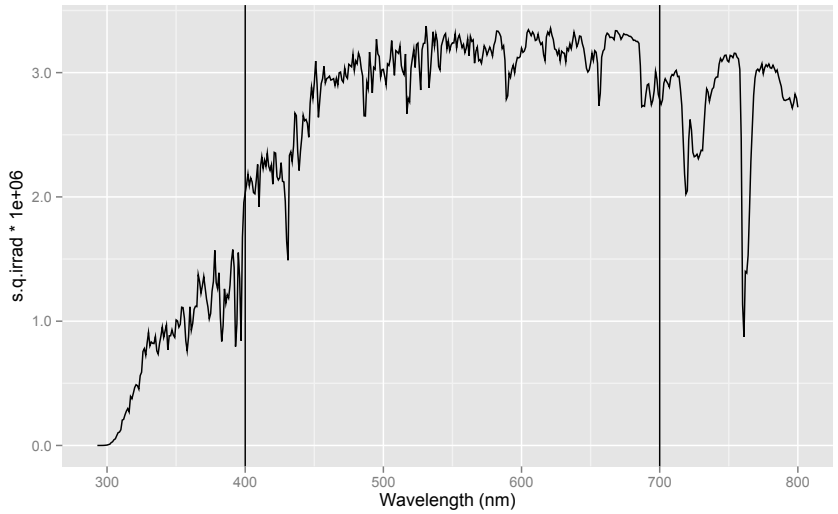
```
figv2 <- fig_sun.e0 +
  annotate_waveband(UVC(), "rect",
                    ymax=0.82) +
  annotate_waveband(UVC(), "text",
                    y=0.86) +
  annotate_waveband(UVB(), "rect",
                    ymax=0.82) +
  annotate_waveband(UVB(), "text",
                    y=0.80, angle=90, hjust=1) +
  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()
```



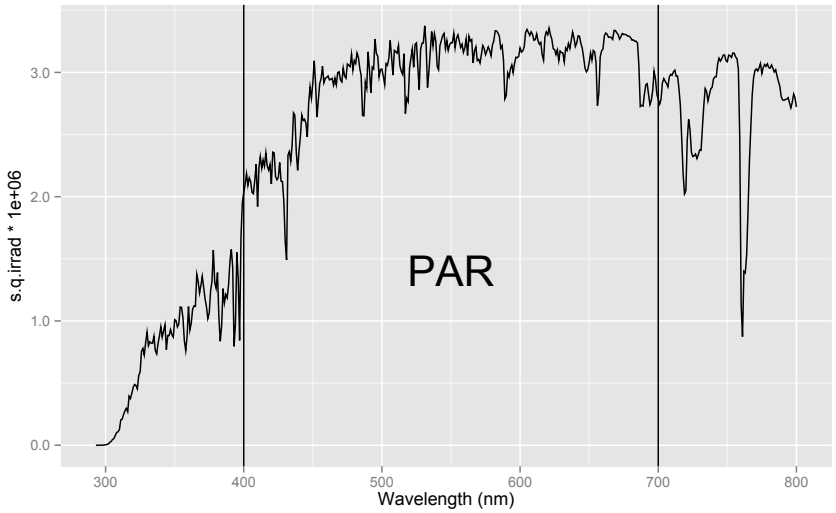
A simple example using `geom_vline`:

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

figvl3
```



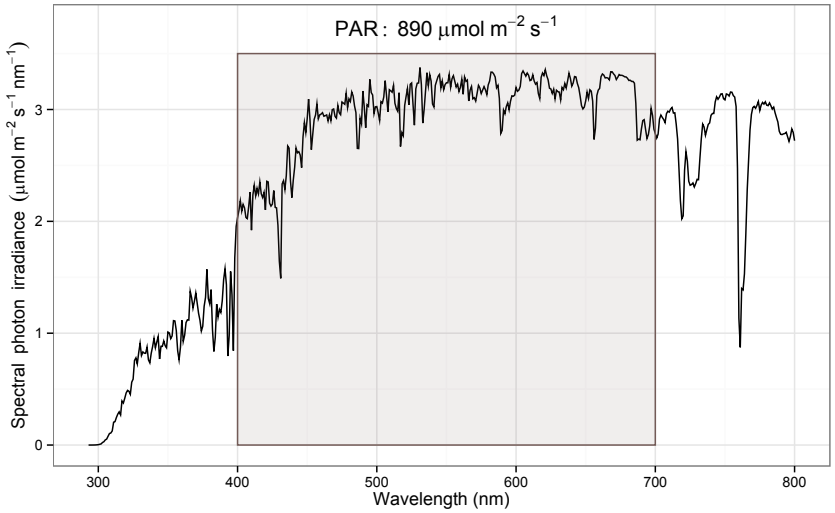
And one where we change some of the aesthetics, and add a label:

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

figvl4
```



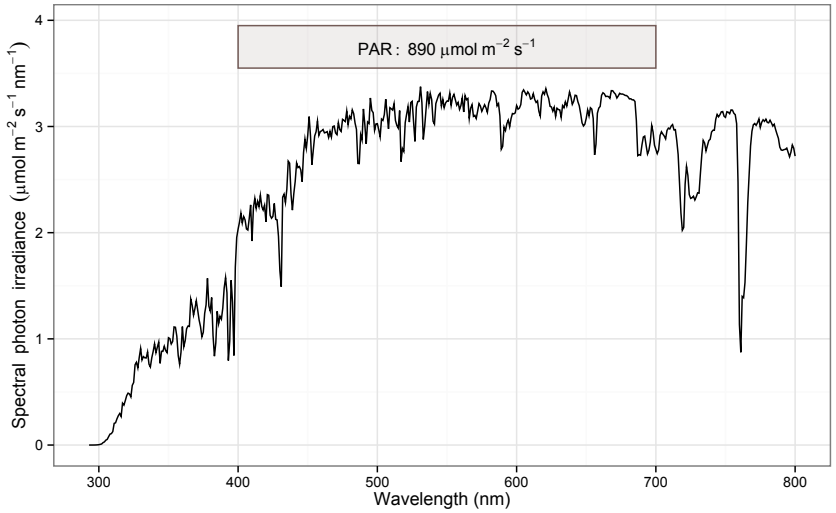
Now including calculated values in the label, first with a simple example with only PAR. Because of using expressions to obtain superscripts we need to add `parse=TRUE` to the call. In addition as we are expressing the integral in photon based units, we also change the type of units used for plotting 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`.

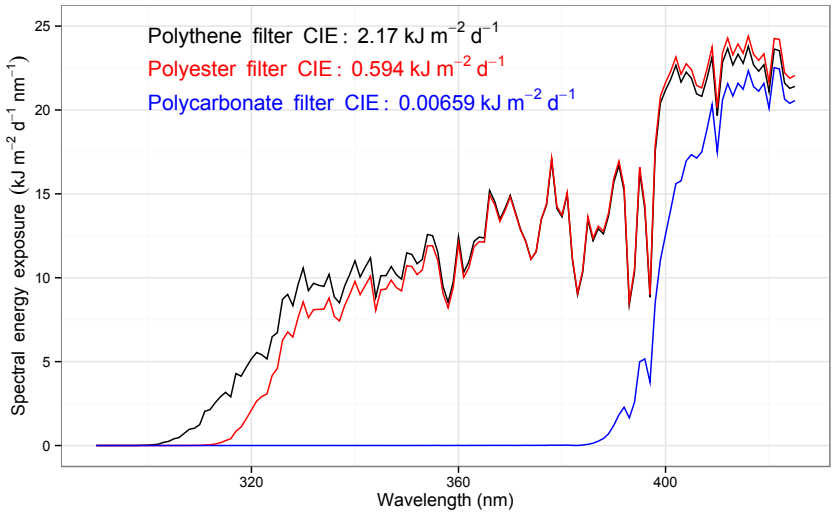
```
fig_sun <- ggplot(data=sun.spct,
                  aes(x=w.length, y=s.q.irrad * 1e6)) +
  geom_line() +
  labs(y = ylab_umol,
       x = "Wavelength (nm)")

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()
```



13.10 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 13.10.2 we show code based on tagging spectral data, and in section 13.10.3 the case of using different data for plotting the guide or key is described.

13.10.1 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,
                           ...)
}
```

13.10.2 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
```

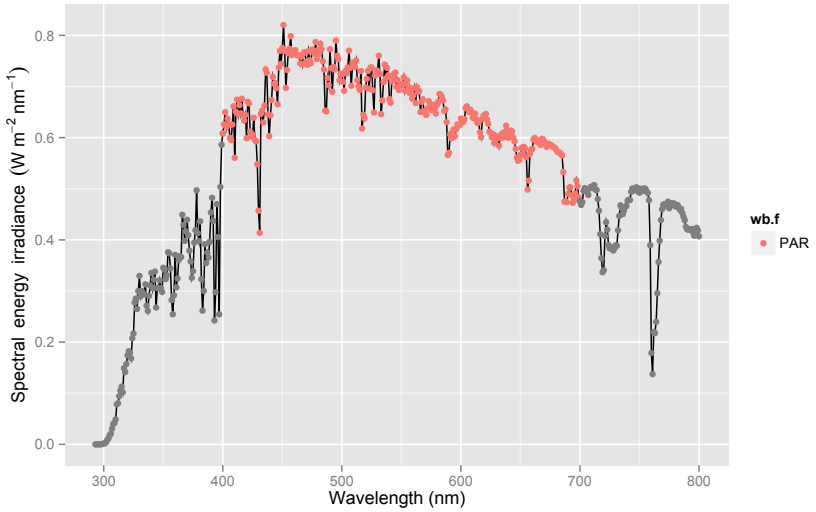
We now tag a spectrum for use in our first plot example.

```
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.

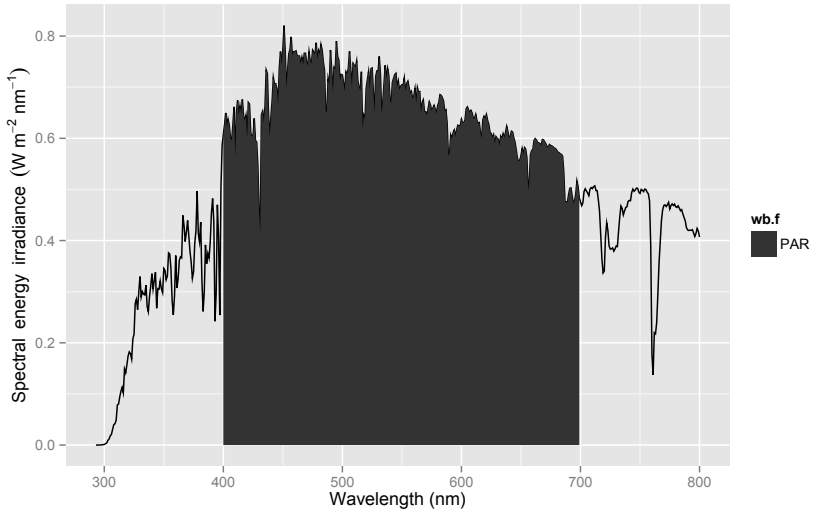
```
fig_sun.t00 <-
  ggplot(data=par.sun.spct,
         aes(x=w.length, y=s.e.irrad)) +
  geom_line() +
```

```
  geom_point(aes(color=wb.f)) +
  labs(
    y = ylab_watt,
    x = "Wavelength (nm)")

fig_sun.t00
```



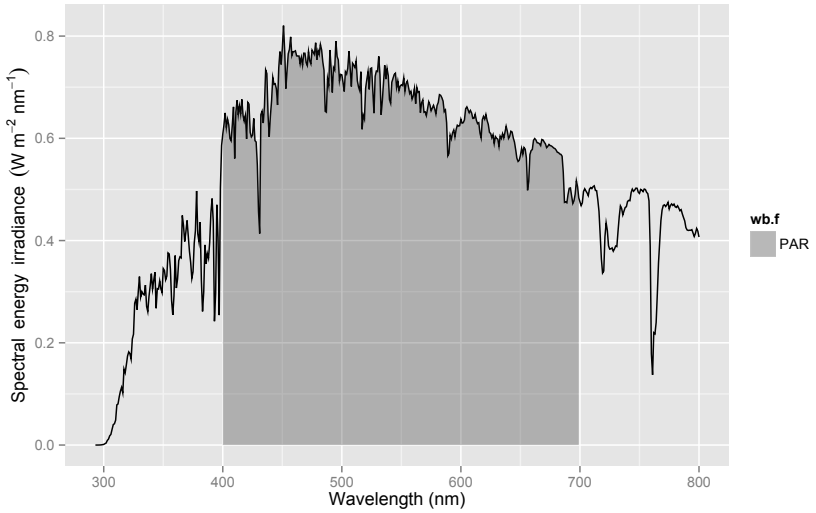
We can also use other geoms like `geom_area` in the next chunk, together with, as an example, a grey fill scale from `ggplot2`.

```
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.

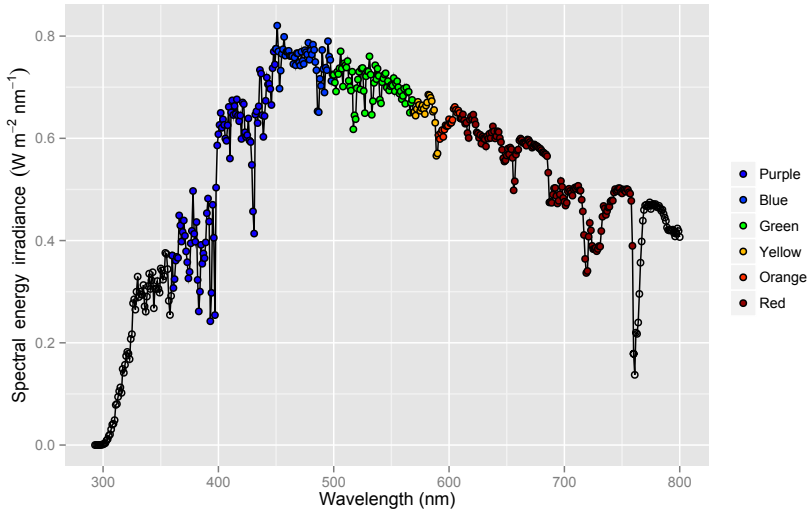
```
tg.sun.spct <- copy(sun.spct)
tag(tg.sun.spct, VIS_bands())
```

See section 13.10.1 on page 113 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'.

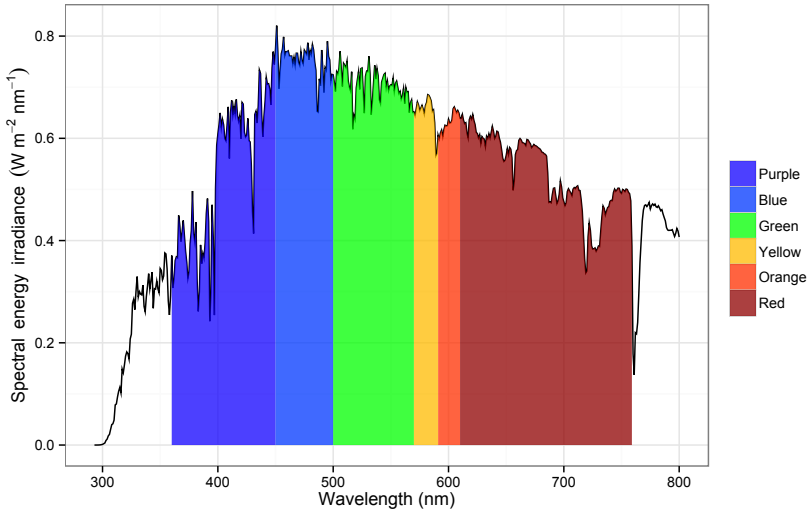
```
fig_sun.t02 <-
  ggplot(data=tg.sun.spct,
         aes(x=w.length, y=s.e.irrad)) +
  geom_line() +
  scale_fill_tgspct(tg.spct=tg.sun.spct) +
  geom_point(aes(fill=wb.f), shape=21)  +
  labs(
    y = ylab_watt,
    x = "Wavelength (nm)")

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()
```

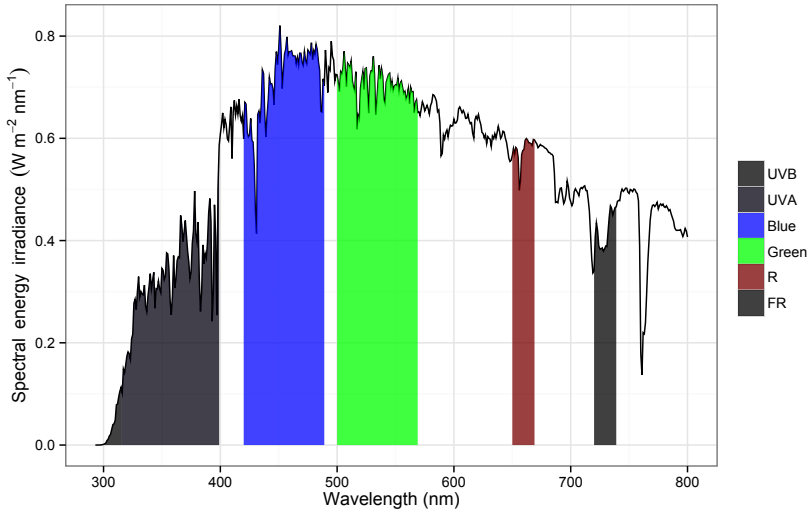


In the next example we tag the solar spectrum with colours using the definitions of plant sensory 'colours'.

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

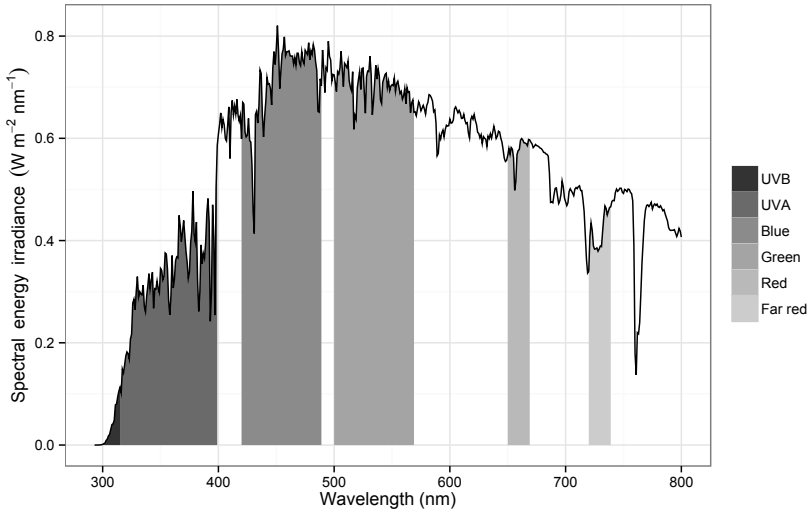
Here we plot the wavebands corresponding to plant sensory 'colours', using the spectrum we tagged in the previous code chunk.

```
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.

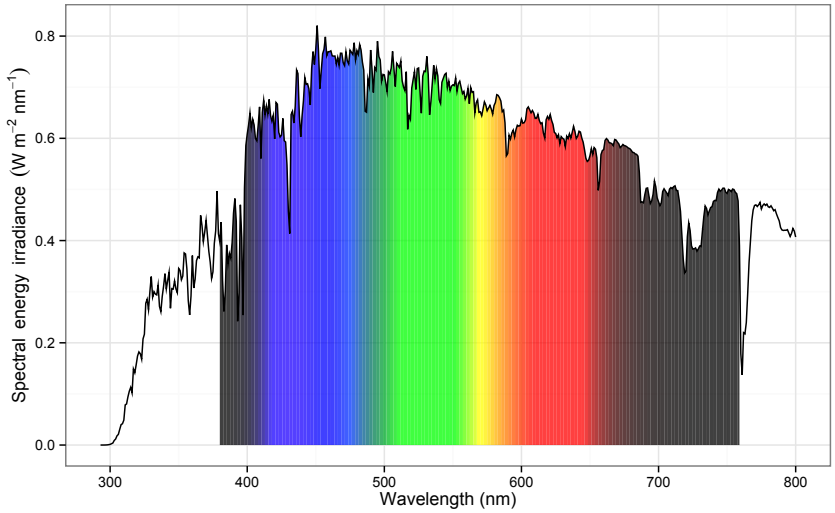
```
interpolate_spct(sun.spct, length.out=800)
```

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()
```

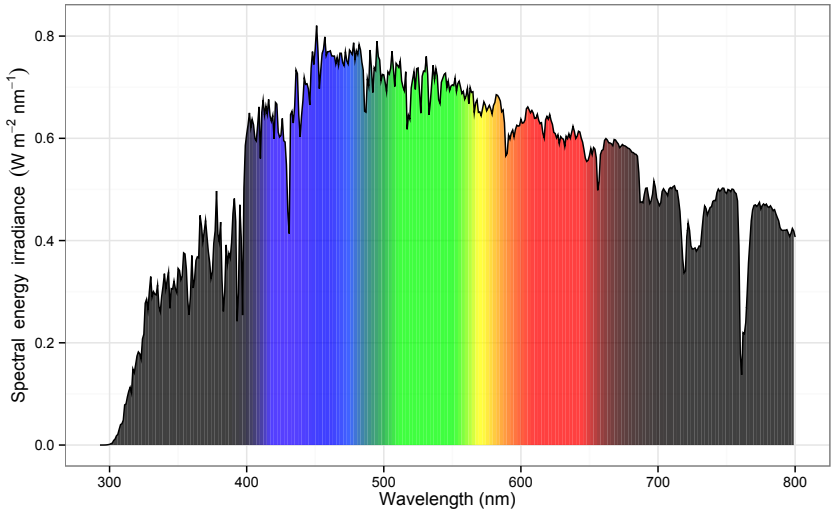


In this other example we tag the whole spectrum, dividing it into 200 wavebands.

```
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))
```

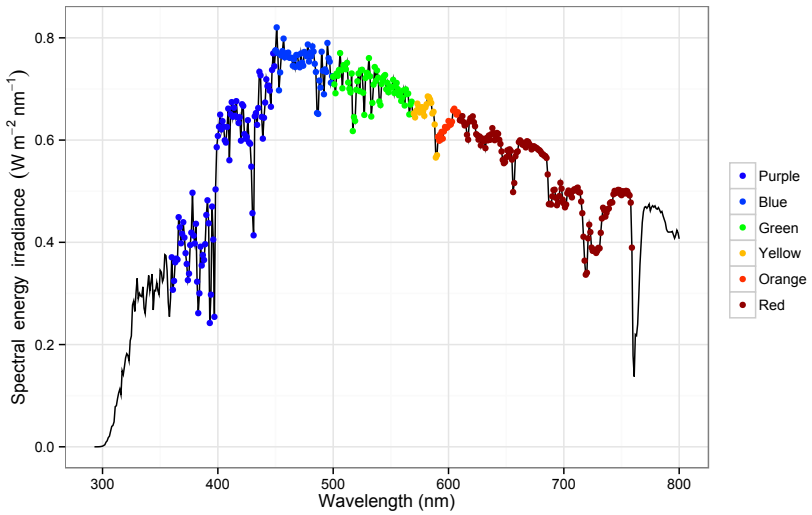
We use `geom_area` and `fill`, and colour the area under the curve. This does not work with `geom_line` because there would not be anything to fill, here we use `geom_area` instead.

```
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.

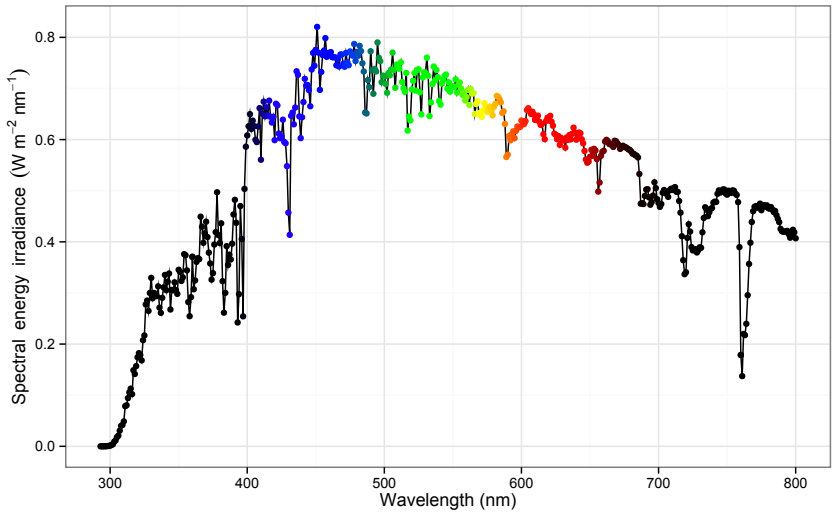
```
fig_sun.tg1 <-
  ggplot(tg.sun.spct,
         aes(x=w.length, y=s.e.irrad)) +
  scale_colour_tgspct(tg.spct=tg.sun.spct) +
  geom_line() +
  geom_point(aes(colour=wb.f)) +
  labs(
    y = ylab_watt,
    x = "Wavelength (nm)")

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.

13.10.3 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.

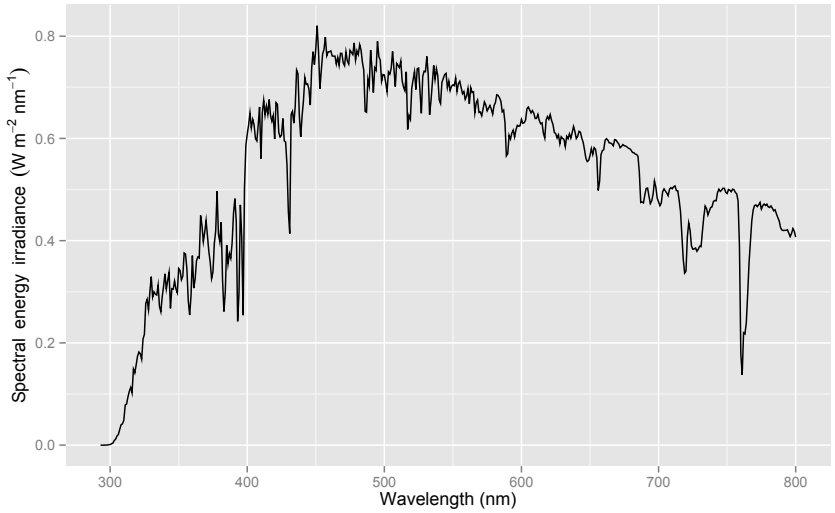
```
attr(wb2rect_spct(PAR()), "spct.tags")
```

```
## $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.

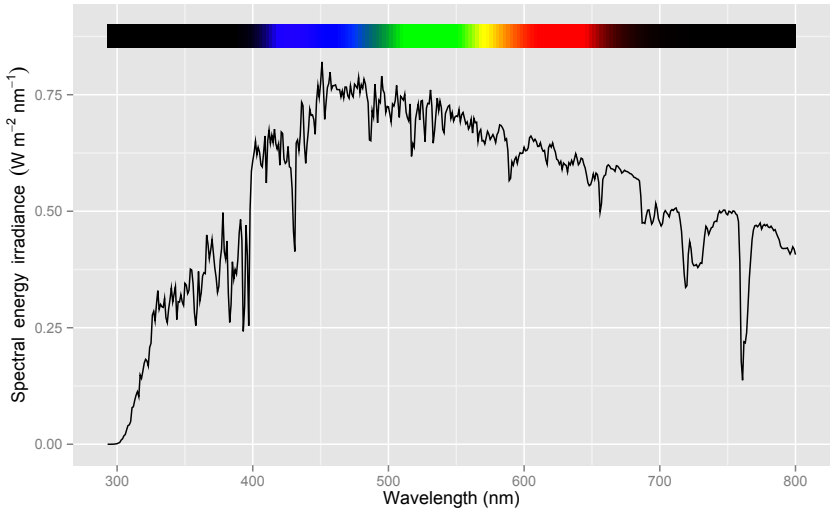
```
fig_sun.z0 <-
  ggplot(data=sun.spct,
         aes(x=w.length, y=s.e.irrad)) +
  geom_line() +
  labs(
    y = ylab_watt,
    x = "Wavelength (nm)")

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.

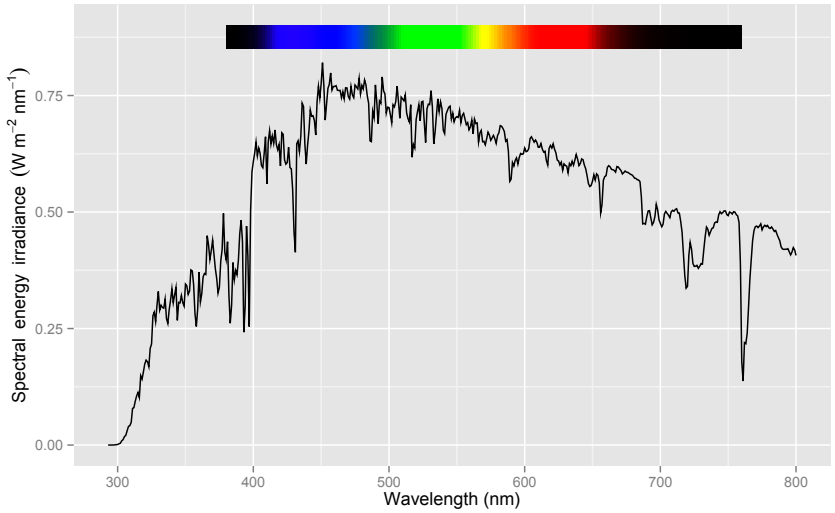
```
wl.guide.spct <-
  wb2rect_spct(split_bands(sun.spct,
```

```
                              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.

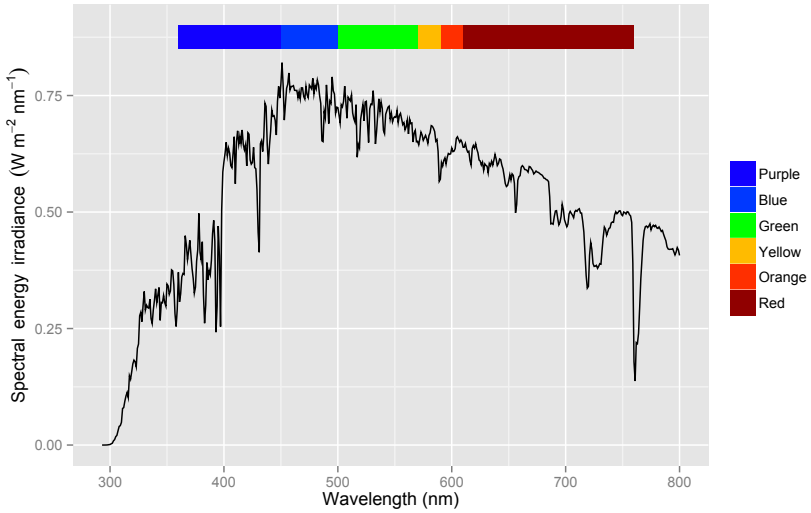
```
wl.guide.spct <- wb2rect_spct(split_bands(VIS(), length.out=200))

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.

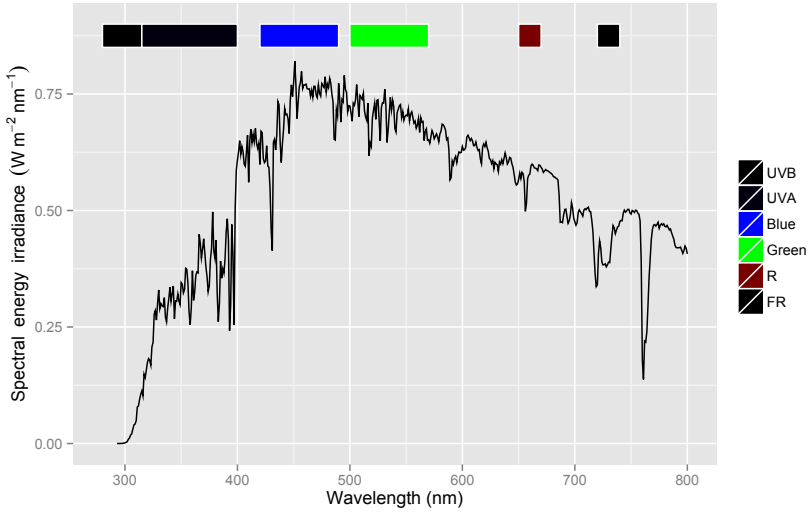
To demonstrate this we replace in the previous example, the previous tagged spectrum with one based on ISO colours. We need to do this replacement in the calls to both `geom_rect` and `scale_fill_tgspct`.

```
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.

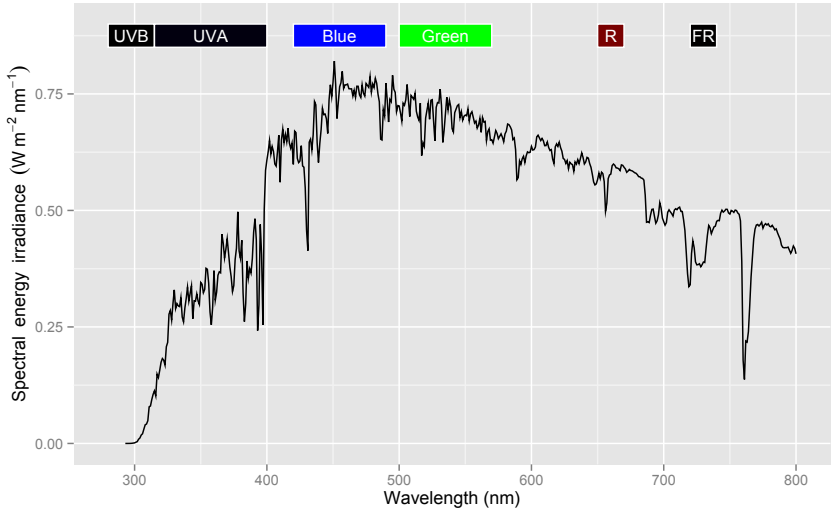
```
plant.guide.spct <- wb2rect_spct(Plant_bands())

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

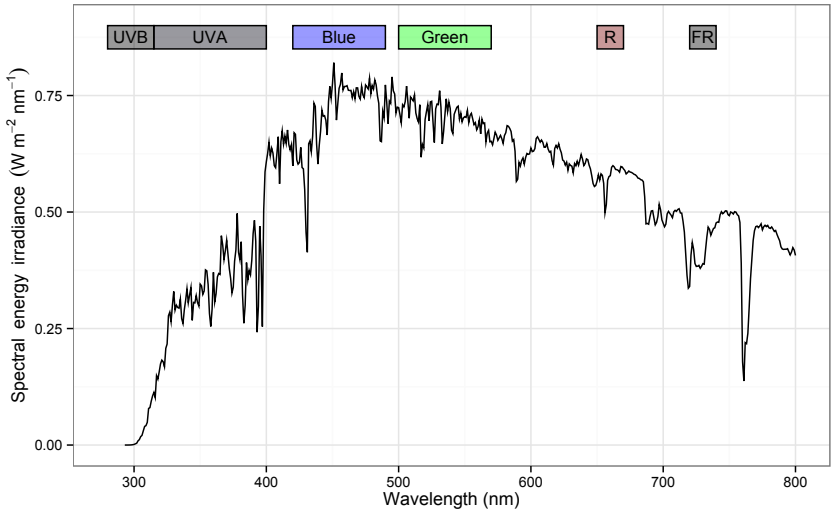
clear. As we are adding labels, the 'guide' or key becomes redundant and we remove it by adding `guide="none"` to the fill scale.

```
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
```



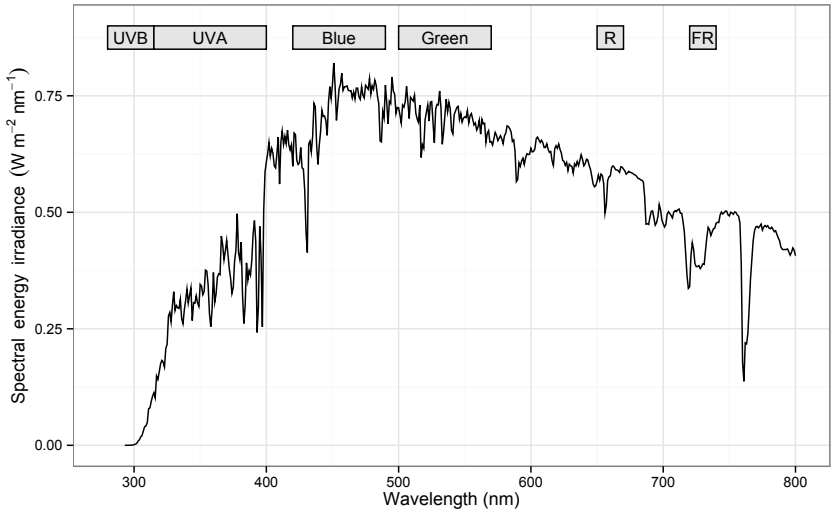
Here we add `alpha` or transparency to make the colours paler, and use black text and lines.

```
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()
```



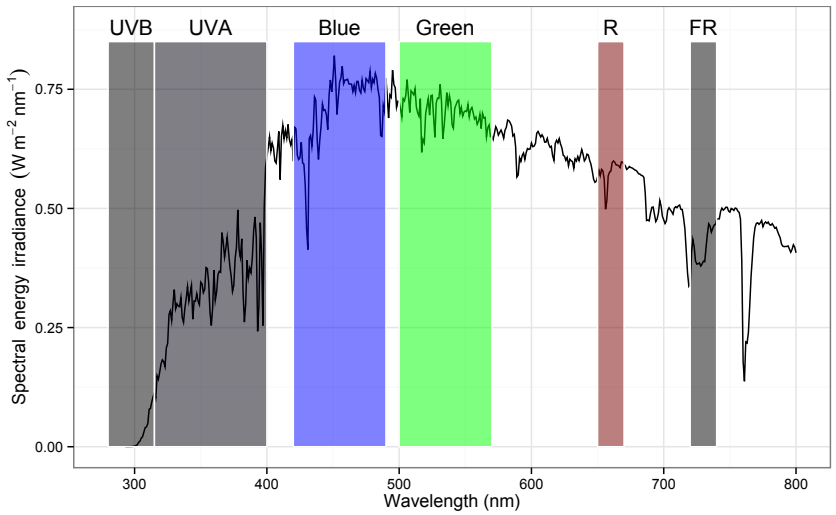
We change the guide so that all rectangles are filled with the same shade of grey by moving `fill` out of `aes` and setting it to a constant.

```
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.

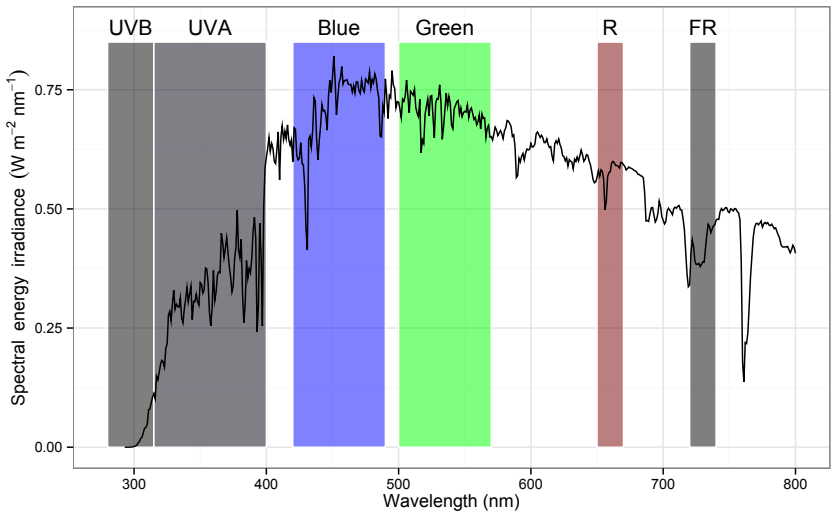
```
plant.guide.spct <- wb2rect_spct(Plant_bands())

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.

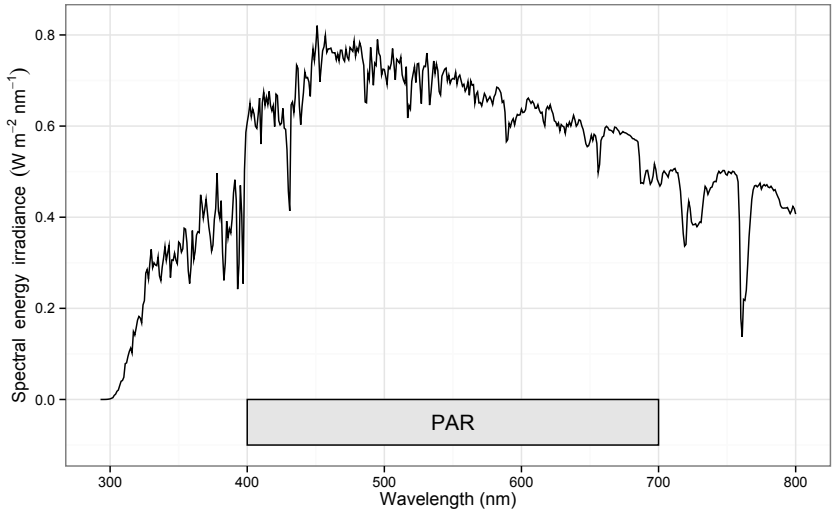
```
plant.guide.spct <- wb2rect_spct(Plant_bands())

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()
```



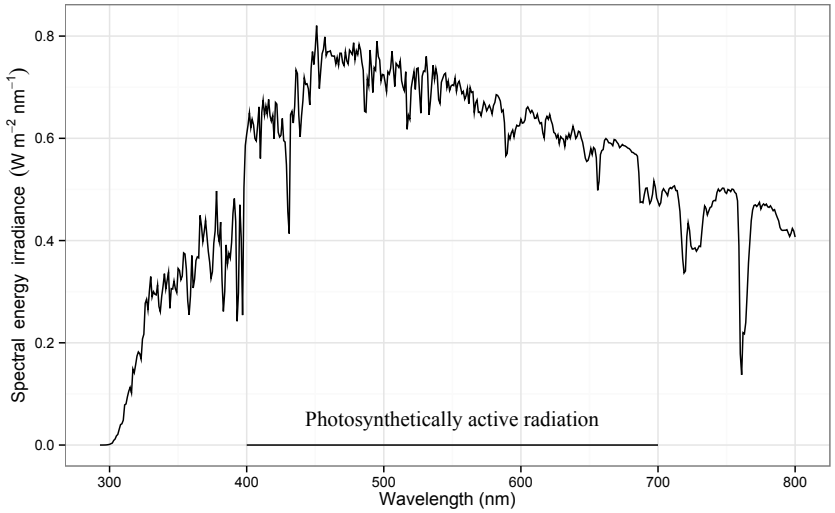
In the examples above we used predefined lists of wavebands, but one can, of course, use any list of waveband definitions, for example explicitly created with `list` and `new_waveband`, or `list` and any combination of user-defined and predefined wavebands. Even single waveband definitions are allowed.

```
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()
```



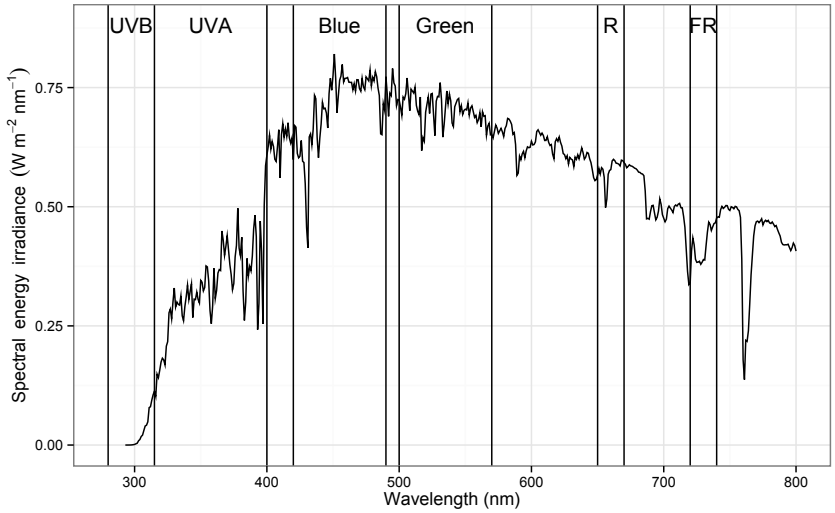
We can also use `geom_segment` to draw lines, including arrows. In this example we also set a different font `family` and label text. We can replace the label text which is by default obtained from the waveband definition by assigning a name to the waveband as member of the list. We use single quotes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



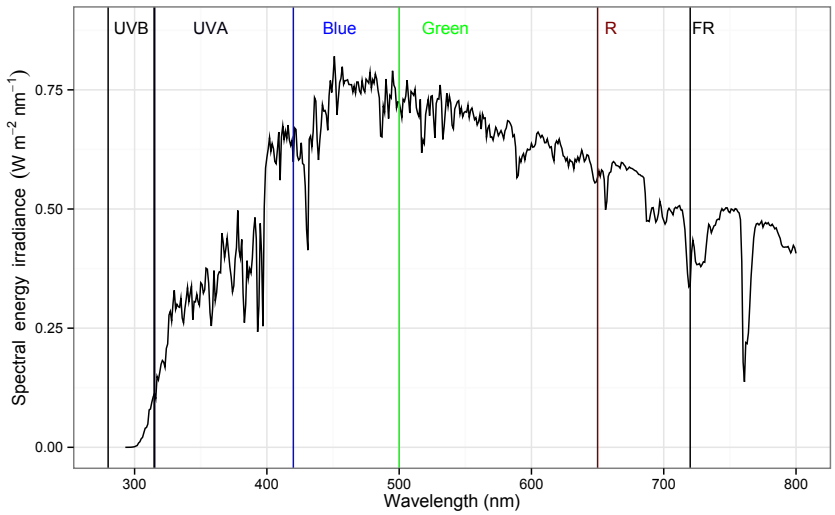
In this section we have used until now function `wb2rect_spct` to create 'spectral' annotation data from waveband definitions. Two other functions are available, that are needed or easier to use in some cases. One such case is when we have a list of wavebands and we would like to mark their boundaries 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.

```
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()
```



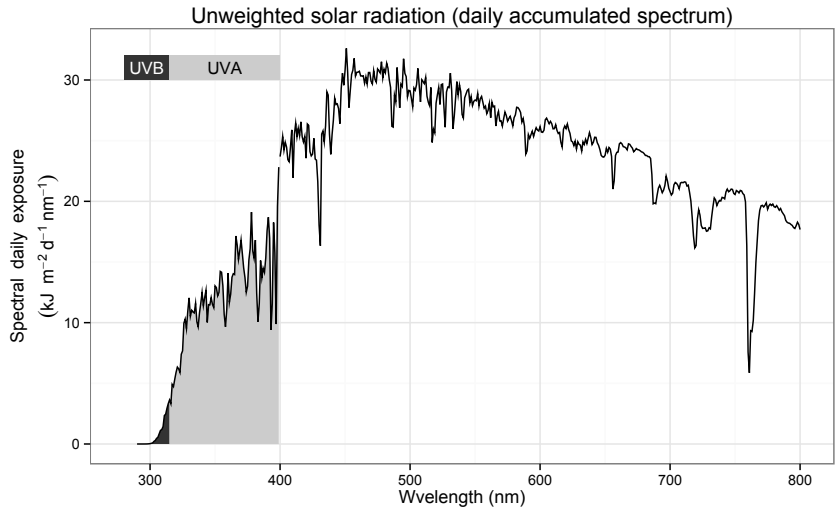
Function `wb2tagged_spct` returns the same data as `wb2spct` but 'tagged'. As shown in the next code chunk, tagging allows us to use waveband-dependent colours to the vertical lines.

```
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.

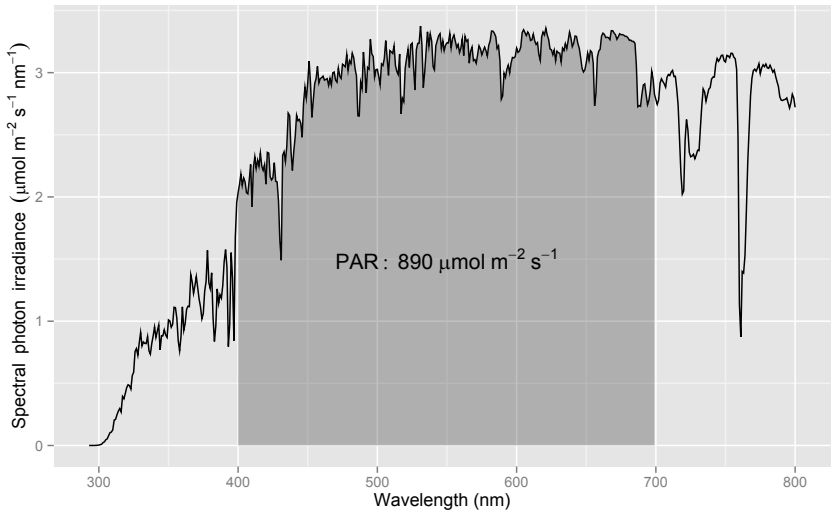
```
my.sun.spct <- sun.daily.spct
tag(my.sun.spct, list(UVB(), UVA()))
annotation.spct <- wb2rect_spct(list(UVB(), UVA()))
fig_sun.uv1 <- ggplot(my.sun.spct,
                      aes(x=w.length,
                          y=s.e.irrad * 1e-3,
                          fill=wb.f)) +
  scale_fill_grey(na.value=NA, 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

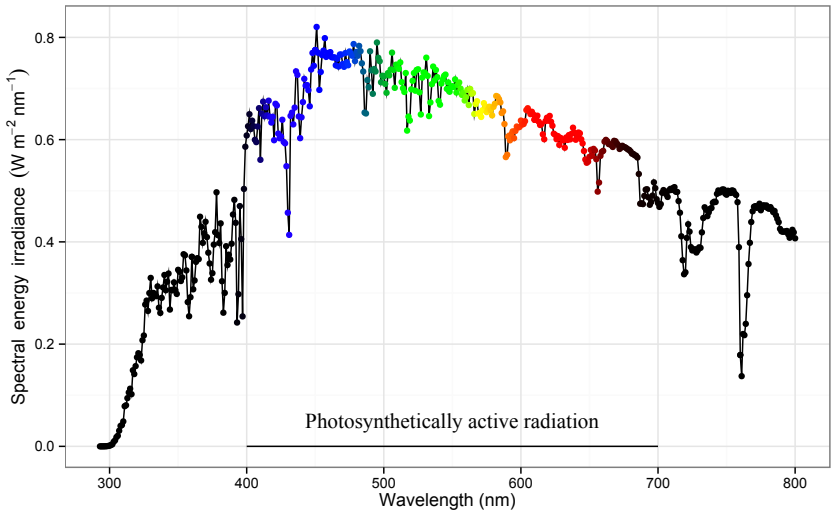
we show new versions of two plots from the previous section, one using a filled area to label the PAR region, and another one using symbols with colours according to their wavelength, to which we add a guide for PAR.

```
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()
```

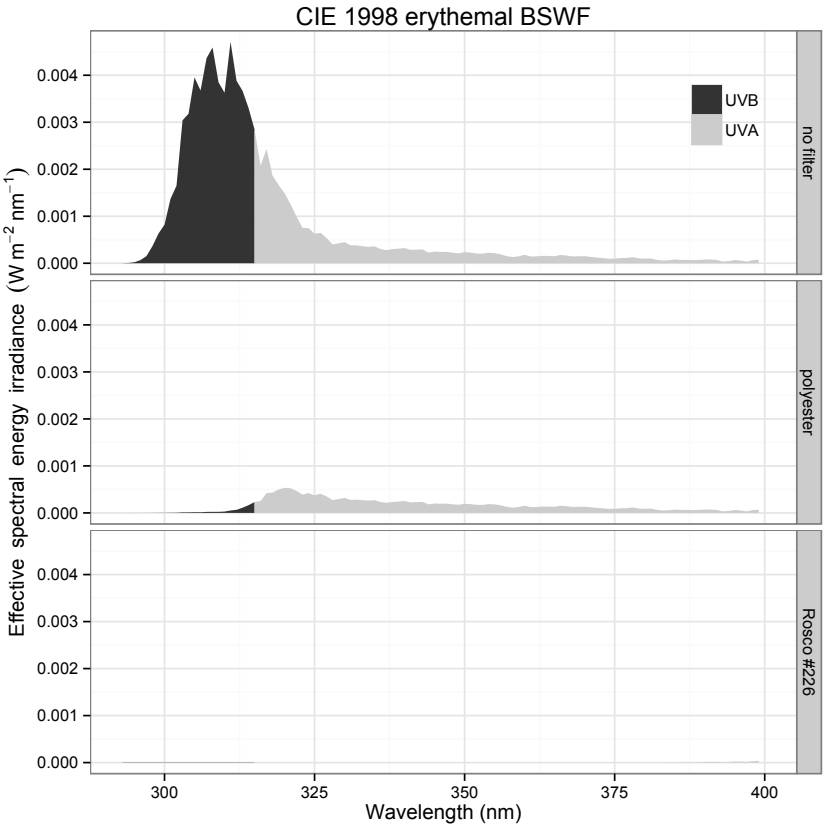


13.11 Task: plotting effective spectral irradiance

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())
invisible(sun.eff.cie.nf.spct[ , filter := 'no filter'])
invisible(sun.eff.cie.pe.spct[ , filter := 'polyester'])
invisible(sun.eff.cie.226.spct[ , filter := 'Rosco #226'])
sun.eff.cie.spct <- rbindspct(list(sun.eff.cie.nf.spct,
                                   sun.eff.cie.pe.spct,
                                   sun.eff.cie.226.spct))
invisible(sun.eff.cie.spct[ , filter := factor(filter)])

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.

13.12 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.

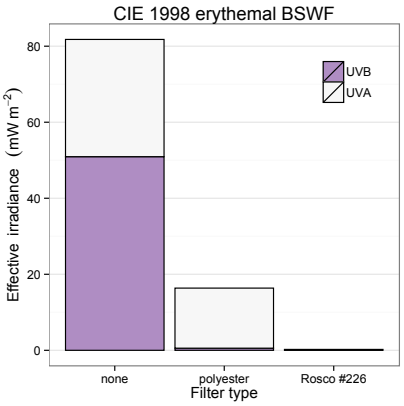
```
cie.nf.irrad <- e_irrad(sun.spct * CIE(),
                        list(UVB(), UVA()))
cie.pe.irrad <- e_irrad(sun.spct * polyester.new.spct * CIE(),
                        list(UVB(), UVA()))
cie.226.irrad <- e_irrad(sun.spct * uv.226.new.spct * CIE(),
                         list(UVB(), UVA()))
```

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')) )
```

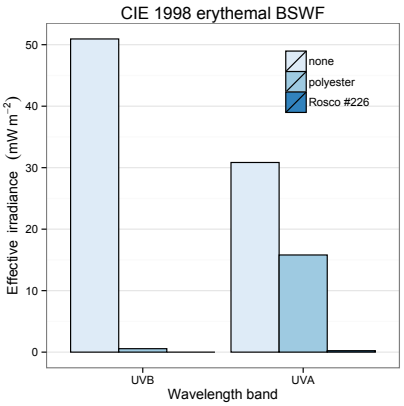
Now we plot stacked bars using `geom_bar`, however as the default `stat` of this geom is not suitable for our data, we specify `stat="identity"` to have the data plotted as is. We set a specific palette for fill, and add a black border to the bars by means of `color="black"`, we remove the grid lines 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
```



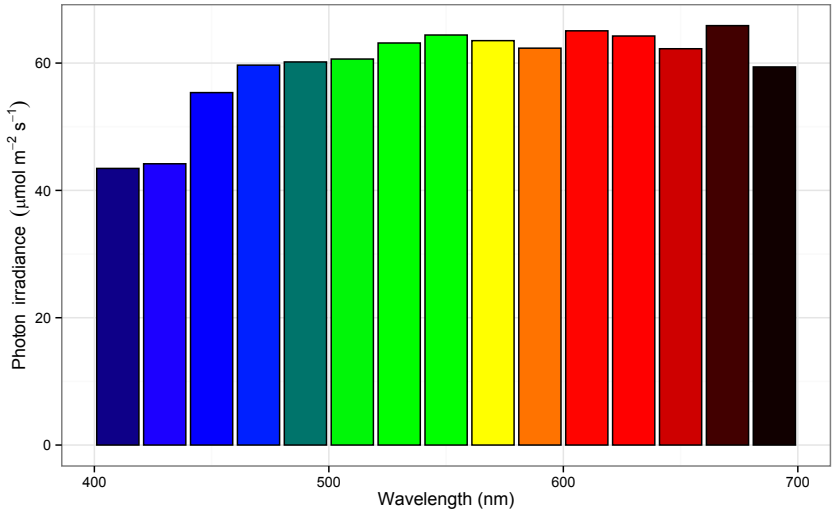
13.13 Task: plotting a spectrum using colour bars

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.

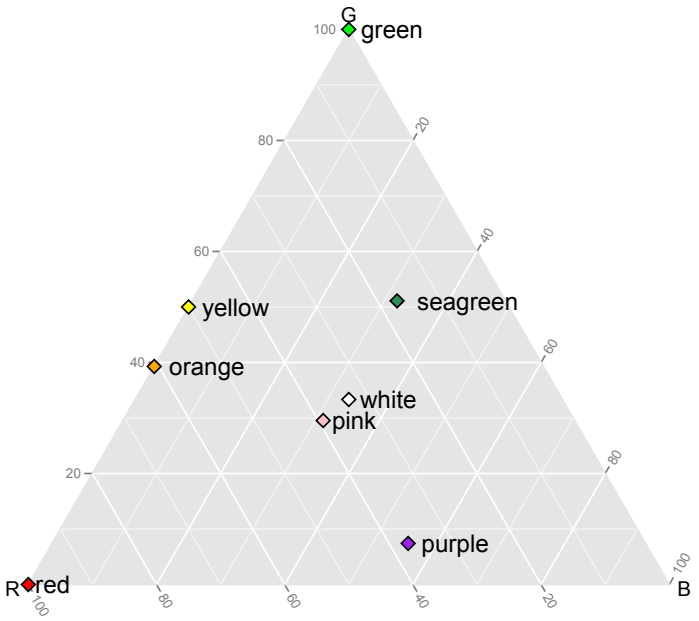
13.14 Task: plotting colours in Maxwell's triangle

13.14.1 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
```



13.15 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.

```
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

CHAPTER 14

Radiation sources

Abstract

In this chapter we explain how to use the spectral data for light sources.

14.1 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(photobiologySun)
library(photobiologyLamps)
library(photobiologyLEDs)
library(photobiologyWavebands)
library(ggplot2)
library(ggtern)
library(photobiologygg)
```

14.2 Introduction

14.3 Task: using the data

14.4 Task: extraterrestrial solar radiation spectra

14.5 Task: terrestrial solar radiation spectra

14.6 Task: incandescent lamps

14.7 Task: discharge lamps

14.8 Task: LEDs

CHAPTER 15

Filters

Abstract

In this chapter we explain how to use spectral data for filters and how to convolute it spectral data for light sources.

15.1 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(photobiologyFilters)
library(photobiologySun)
library(photobiologyLamps)
library(photobiologyLEDs)
library(photobiologyWavebands)
library(ggplot2)
library(ggtern)
library(photobiologygg)
```

15.2 Introduction

15.3 Task: using the data

15.4 Task: spectral transmittance for optical glass filters

15.5 Task: spectral transmittance for plastic films

15.6 Task: spectral transmittance for plastic sheets

CHAPTER 16

Photoreceptors

Abstract

In this chapter we explain how to .

16.1 Task:

Part IV

Data acquisition and modelling

CHAPTER 17

Calibration

Abstract

In this chapter we explain how to .

17.1 Task:

CHAPTER 18

Simulation

Abstract

In this chapter we explain how to .

18.1 Task:

CHAPTER 19

Measurement

Abstract

In this chapter we explain how to .

19.1 Task:

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 pp. 9, 47).

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. x).

Chang, W. (2013). *R Graphics Cookbook*. 1-2. Sebastopol: O'Reilly Media, p. 413. ISBN: 9781449316952. URL: http://medcontent.metapress.com/index/A65RM03P4874243N.pdf (cit. on p. 228).

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. x).

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. ix).

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. x).

Part V

Appendixes

R as a powerful calculator

A.1 Working in the R console

I assume that you are already familiar with RStudio. These examples use only the console window, and results a printed to the console. The values stored in the different variables are also visible in the Environment tab in RStudio.

In the console can type commands at the > prompt. When you end a line by pressing the return key, if the line can be interpreted as an R command, the result will be printed in the console, followed by a new > prompt. If the command is incomplete a + continuation prompt will be shown, and you will be able to type-in the rest of the command. For example if the whole calculation that you would like to do is 1 + 2 + 4, if you enter in the console `1 + 2 +` in one line, you will get a continuation prompt where you will be able to type `3`. However, if you type `1 + 2`, the result will be calculated, and printed.

When working at the command prompt, results are printed by default, but other cases you may need to use the function `print` explicitly. The examples here rely on the automatic printing.

The idea with these examples is that you learn by working out how different commands work based on the results of the example calculations listed. The examples are designed so that they allow the rules, and also a few quirks, to be found by 'detective work'. This should hopefully lead to better understanding than just studying rules.

A.2 Examples with numbers

When working with arithmetic expression the normal precedence rules are followed and parentheses can be used to alter this order. In addition parentheses can be nested.

```
1 + 1

## [1] 2

2 * 2

## [1] 4

2 + 10 / 5

## [1] 4

(2 + 10) / 5

## [1] 2.4

10^2 + 1

## [1] 101

sqrt(9)

## [1] 3

pi # whole precision not shown when printing

## [1] 3.141593

print(pi, digits=22)

## [1] 3.141592653589793115998

sin(pi) # oops! Read on for explanation.

## [1] 1.224606e-16

log(100)

## [1] 4.60517

log10(100)

## [1] 2

log2(8)

## [1] 3

exp(1)

## [1] 2.718282
```

One can use variables to store values. Variable names and all other names in R are case sensitive. Variables `a` and `A` are two different variables. Variable names can be quite long, but usually it is not a good idea to use very long names. Here I am using very short names, that is usually a very bad idea. However, in cases like these examples where the stored values have no real connection

to the real world and are used just once or twice, these names emphasize the abstract nature.

```
a <- 1
a + 1

## [1] 2

a

## [1] 1

b <- 10
b <- a + b
b

## [1] 11

3e-2 * 2.0

## [1] 0.06
```

There are some syntactically legal statements that are not very frequently used, but you should be aware that they are valid, as they will not trigger error messages, and may surprise you. The important thing is that you write commands consistently. `1 -> a` is valid but almost never used.

```
a <- b <- c <- 0.0
a

## [1] 0

b

## [1] 0

c

## [1] 0

1 -> a
a

## [1] 1

a = 3
a

## [1] 3
```

Numeric variables can contain more than one value. Even single numbers are vectors of length one. We will later see why this is important. As you have seen above the results of calculations were printed preceded with `[1]`. This is the index or position in the vector of the first number (or other value) displayed at the head of the line.

One can use `c` 'concatenate' to create a vector of numbers from individual numbers.

```
a <- c(3,1,2)
a

## [1] 3 1 2

b <- c(4,5,0)
b

## [1] 4 5 0

c <- c(a, b)
c

## [1] 3 1 2 4 5 0

d <- c(b, a)
d

## [1] 4 5 0 3 1 2
```

One can also create sequences, or repeat values:

```
a <- -1:5
a

## [1] -1  0  1  2  3  4  5

b <- 5:-1
b

## [1]  5  4  3  2  1  0 -1

c <- seq(from = -1, to = 1, by = 0.1)
c

##  [1] -1.0 -0.9 -0.8 -0.7 -0.6 -0.5 -0.4 -0.3 -0.2
## [10] -0.1  0.0  0.1  0.2  0.3  0.4  0.5  0.6  0.7
## [19]  0.8  0.9  1.0

d <- rep(-5, 4)
d

## [1] -5 -5 -5 -5
```

Now something that makes R different from most other programming languages: vectorized arithmetic.

```
a + 1 # we add one to vector a defined above

## [1] 0 1 2 3 4 5 6

(a + 1) * 2

## [1]  0  2  4  6  8 10 12
```

```
a + b

## [1] 4 4 4 4 4 4 4

a - a

## [1] 0 0 0 0 0 0 0
```

It can be seen in first line above, another peculiarity of R, that frequently called recycling: as vector `a` is of length 6, but the constant 1 is a vector of length 1, this 1 is extended by recycling into a vector of the same length as the longest vector in the statement.

Make sure you understand what calculations are taking place in the chunk above, and also the one below.

```
a <- rep(1, 6)
a

## [1] 1 1 1 1 1 1

a + 1:2

## [1] 2 3 2 3 2 3

a + 1:3

## [1] 2 3 4 2 3 4

a + 1:4

## Warning in a + 1:4:  longer object length is not a multiple
of shorter object length

## [1] 2 3 4 5 2 3
```

A couple on useful things to know: a vector can have length zero. One can remove variables from the workspace with `rm`. One can use `ls()` to list all objects in the environment, or by supplying a `pattern` argument, only the objects with names matching the `pattern`. The pattern is given as a regular expression, with `[]` enclosing alternative matching characters, `^` and `$` indicating the extremes of the name. For example `"^z$"` matches only the single character 'z' while `"^z"` matches any name starting with 'z'. In contrast `"^[zy]$"` matches both 'z' and 'y' but neither 'zy' nor 'yz', and `"^[a-z]"` matches any name starting with a lower case ASCII letter. If you are using RStudio, all objects are listed in the Environment pane, and the search box of the panel can be used to find a given object.

```
z <- numeric(0)
z

## numeric(0)

ls(pattern="^z$")

## [1] "z"
```

```
rm(z)
try(z)
ls(pattern="^z$")

## character(0)
```

There are some special values available for numbers. NA meaning 'not available' is used for missing values. Calculations can yield also the following values `NaN` 'not a number', `Inf` and `-Inf` for ∞ and $-\infty$. As you will see below, calculations yielding these values do **not** trigger errors or warnings, as they are arithmetically valid.

```
a <- NA
a

## [1] NA

-1 / 0

## [1] -Inf

1 / 0

## [1] Inf

Inf / Inf

## [1] NaN

Inf + 4

## [1] Inf
```

One thing to be aware of, and which we will discuss again later, is that numbers in computers are almost always stored with finite precision. This means that they not always behave as Real numbers as defined in mathematics. In R the usual numbers are stored as `double-precision floats`, which means that there are limits to the largest and smallest numbers that can be represented (approx. $-1 \cdot 10^{308}$ and $1 \cdot 10^{308}$), and the number of significant digits that can be stored (usually described as ϵ (epsilon, abbreviated `eps`, defined as the largest number for which $1 + \epsilon = 1$)). This can be sometimes important, and can generate unexpected results in some cases, especially when testing for equality. In the example below, the result of the subtraction is still exactly 1.

```
1 - 1e-20

## [1] 1
```

It is usually safer not to test for equality to zero when working with numeric values. One alternative is comparing against a suitably small number, which will depend on the situation, although `eps` is usualy a safe bet, unless the expected range of values is known to be small.

```
abs(x) < eps
abs(x) < 1e-100
```

The same applies to tests for equality, so whenever possible according to the logic of the calculations, it is best to test for inequalities, for example using `x <= 1.0` instead of `x == 1.0`. If this is not possible, then the tests should be treated as above, for example replacing `x == 1.0` with `abs(x - 1.0) < eps`.

When comparing integer values these problems do not exist, as integer arithmetic is not afected by loss of precision in calculations restricted to integers (the `L` comes from 'long' a name sometimes used for a machine represenation of intergers):

```
1L + 3L

## [1] 4

1L * 3L

## [1] 3

1L %/% 3L

## [1] 0

1L / 3L

## [1] 0.3333333
```

The last example above, using the 'usual' division operator yields a floating-point `numeric` result, while the integer division operator `%/%` yields an integer result.

A.3 Examples with logical values

What in maths are usually called Boolean values, are called `logical` values in R. They can have only two values `TRUE` and `FALSE`, in addition to `NA`. They are vectors. There are also logical operators that allow boolean algebra (and some support for set operations that we will not describe here).

```
a <- TRUE
b <- FALSE
a

## [1] TRUE

!a # negation

## [1] FALSE

a && b # logical AND

## [1] FALSE
```

```
a || b # logical OR

## [1] TRUE
```

Again vectorization is possible. I present this here, and will come back again to this, because this is one of the most troublesome aspects of the R language. The two types of 'equivalent' logical operators behave very differently, but use very similar syntax! The vectorized operators have single-character names & and |, while the non vectorized ones have two double-character names && and ||. There is only one version of the negation operator ! that is vectorized.

```
a <- c(TRUE,FALSE)
b <- c(TRUE,TRUE)
a

## [1]  TRUE FALSE

b

## [1] TRUE TRUE

a & b # vectorized AND

## [1]  TRUE FALSE

a | b # vectorized OR

## [1] TRUE TRUE

a && b # not vectorized

## [1] TRUE

a || b # not vectorized

## [1] TRUE
```

Functions `any` and `all` take a logical vector as argument, and return a single logical value 'summarizing' the logical values in the vector. `all` returns TRUE only if every value in the argument is TRUE, and `any` returns TRUE unless every value in the argument is FALSE.

```
any(a)

## [1] TRUE

all(a)

## [1] FALSE

any(a & b)

## [1] TRUE

all(a & b)

## [1] FALSE
```

Another important thing to know about logical operators is that they 'short-cut' evaluation. If the result is known from the first part of the statement, the rest of the statement is not evaluated. Try to understand what happens when you enter the following commands.

```
TRUE || NA

## [1] TRUE

FALSE || NA

## [1] NA

TRUE && NA

## [1] NA

FALSE && NA

## [1] FALSE

TRUE && FALSE && NA

## [1] FALSE

TRUE && TRUE && NA

## [1] NA
```

When using the vectorized operators on vectors of length greater than one, 'short-cut' evaluation still applies for the result obtained.

```
a & b & NA

## [1]    NA FALSE

a & b & c(NA, NA)

## [1]    NA FALSE

a | b | c(NA, NA)

## [1] TRUE TRUE
```

A.4 Comparison operators

Comparison operators yield as a result logical values.

```
1.2 > 1.0

## [1] TRUE

1.2 >= 1.0

## [1] TRUE
```

```
1.2 == 1.0 # be aware that here we use two = symbols

## [1] FALSE

1.2 != 1.0

## [1] TRUE

1.2 <= 1.0

## [1] FALSE

1.2 < 1.0

## [1] FALSE

a <- 20
a < 100 && a > 10

## [1] TRUE
```

Again these operators can be used on vectors of any length, the result is a logical vector.

```
a <- 1:10
a > 5

##  [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE
##  [8]  TRUE  TRUE  TRUE

a < 5

##  [1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE
##  [8] FALSE FALSE FALSE

a == 5

##  [1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
##  [8] FALSE FALSE FALSE

all(a > 5)

## [1] FALSE

any(a > 5)

## [1] TRUE

b <- a > 5
b

##  [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE
##  [8]  TRUE  TRUE  TRUE

any(b)

## [1] TRUE

all(b)

## [1] FALSE
```

Be once more aware of 'short-cut evaluation'. If the result would not be affected by the missing value then the result is returned. If the presence of the NA makes the end result unknown, then NA is returned.

```
c <- c(a, NA)
c > 5

##  [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE
##  [8]  TRUE  TRUE  TRUE    NA

all(c > 5)

## [1] FALSE

any(c > 5)

## [1] TRUE

all(c < 20)

## [1] NA

any(c > 20)

## [1] NA

is.na(a)

##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [8] FALSE FALSE FALSE

is.na(c)

##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [8] FALSE FALSE FALSE  TRUE

any(is.na(c))

## [1] TRUE

all(is.na(c))

## [1] FALSE
```

This behaviour can be changed by using the optional argument `na.rm` which removes NA values **before** the function is applied. (Many functions in R have this optional parameter.)

```
all(c < 20)

## [1] NA

any(c > 20)

## [1] NA

all(c < 20, na.rm=TRUE)
```

```
## [1] TRUE

any(c > 20, na.rm=TRUE)

## [1] FALSE
```

You may skip this on first read, see page 174.

```
1e20 == 1 + 1e20

## [1] TRUE

1 == 1 + 1e-20

## [1] TRUE

0 == 1e-20

## [1] FALSE
```

In many situations, when writing programs one should avoid testing for equality of floating point numbers ('floats'). Here we show how to handle gracefully rounding errors.

```
a == 0.0 # may not always work

##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [8] FALSE FALSE FALSE

abs(a) < 1e-15 # is safer

##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [8] FALSE FALSE FALSE

sin(pi) == 0.0 # angle in radians, not degrees!

## [1] FALSE

sin(2 * pi) == 0.0

## [1] FALSE

abs(sin(pi)) < 1e-15

## [1] TRUE

abs(sin(2 * pi)) < 1e-15

## [1] TRUE

sin(pi)

## [1] 1.224606e-16

sin(2 * pi)

## [1] -2.449213e-16
```

```
.Machine$double.eps # see help for .Machine for explanation

## [1] 2.220446e-16

.Machine$double.neg.eps

## [1] 1.110223e-16
```

A.5 Character values

Character variables can be used to store any character. Character constants are written by enclosing characters in quotes. There are three types of quotes in the ASCII character set, double quotes ", single quotes ', and back ticks `. The first two types of quotes can be used for delimiting characters.

```
a <- "A"
b <- letters[2]
c <- letters[1]
a

## [1] "A"

b

## [1] "b"

c

## [1] "a"

d <- c(a, b, c)
d

## [1] "A" "b" "a"

e <- c(a, b, "c")
e

## [1] "A" "b" "c"

h <- "1"
try(h + 2)
```

Vectors of characters are not the same as character strings.

```
f <- c("1", "2", "3")
g <- "123"
f == g

## [1] FALSE FALSE FALSE

f

## [1] "1" "2" "3"
```

```
g

## [1] "123"
```

One can use the 'other' type of quotes as delimiter when one want to include quotes in a string. Pretty-printing is changing what I typed into how the string is stored in R: I typed `b <- 'He said "hello" when he came in'`, try it.

```
a <- "He said 'hello' when he came in"
a

## [1] "He said 'hello' when he came in"

b <- 'He said "hello" when he came in'
b

## [1] "He said \"hello\" when he came in"
```

The outer quotes are not part of the string, they are 'delimiters' used to mark the boundaries. As you can see when `b` is printed special characters can be represented using 'escape sequences'. There are several of them, and here we will show just a few.

```
c <- "abc\ndef\txyz"
print(c)

## [1] "abc\ndef\txyz"

cat(c)

## abc
## def xyz
```

Above, you will not see any effect of these escapes when using `print`: `\n` represents 'new line' and `\t` means 'tab' (tabulator). The *scape codes* work only in some contexts, as when using `cat` to generate the output. They also are very useful when one wants to split an axis-label, title or label in a plot into two or more lines.

A.6 Finding the 'mode' of objects

Variables have *mode* that determines what can be stored in them. But differently to other languages, assignment of a variable of a different mode is allowed. However, there is a restriction that all elements in a vector, array or matrix, must be of the same mode, while this is not required for lists. Functions with names starting with `is.` are tests returning TRUE, FALSE or NA.

```
my_var <- 1:5
mode(my_var)

## [1] "numeric"
```

```
is.numeric(my_var)

## [1] TRUE

is.logical(my_var)

## [1] FALSE

is.character(my_var)

## [1] FALSE

my_var <- "abc"
mode(my_var)

## [1] "character"
```

A.7 Type conversions

The least intuitive ones are those related to logical values. All others are as one would expect. By convention, functions used to convert objects from one mode to a different one have names starting with `as.`.

```
as.character(1)

## [1] "1"

as.character(3.0e10)

## [1] "3e+10"

as.numeric("1")

## [1] 1

as.numeric("5E+5")

## [1] 5e+05

as.numeric("A")

## Warning:  NAs introduced by coercion
## [1] NA

as.numeric(TRUE)

## [1] 1

as.numeric(FALSE)

## [1] 0

TRUE + TRUE

## [1] 2
```

```
TRUE + FALSE

## [1] 1

TRUE * 2

## [1] 2

FALSE * 2

## [1] 0

as.logical("T")

## [1] TRUE

as.logical("t")

## [1] NA

as.logical("TRUE")

## [1] TRUE

as.logical("true")

## [1] TRUE

as.logical(100)

## [1] TRUE

as.logical(0)

## [1] FALSE

as.logical(-1)

## [1] TRUE
```

```
f <- c("1", "2", "3")
g <- "123"
as.numeric(f)

## [1] 1 2 3

as.numeric(g)

## [1] 123
```

Some tricks useful when dealing with results. Be aware that the printing is being done by default, these functions return numerical values.

```
round(0.0124567, 3)

## [1] 0.012
```

```
round(0.0124567, 1)

## [1] 0

round(0.0124567, 5)

## [1] 0.01246

signif(0.0124567, 3)

## [1] 0.0125

round(1789.1234, 3)

## [1] 1789.123

signif(1789.1234, 3)

## [1] 1790

a <- 0.12345
b <- round(a, 2)
a == b

## [1] FALSE

a - b

## [1] 0.00345

b

## [1] 0.12
```

A.8 Vectors

You already know how to create a vector. Now we are going to see how to get individual numbers out of a vector. They are accessed using an index. The index indicates the position in the vector, starting from one, following the usual mathematical tradition. What in maths would be x_i for a vector x, in R is represented as `x[i]`. (In R indexes (or subscripts) always start from one, while in some other programming languages indexes start from zero.)

```
a <- letters[1:10]
a

##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"

a[2]

## [1] "b"

a[c(3,2)]

## [1] "c" "b"
```

```
a[10:1]

##  [1] "j" "i" "h" "g" "f" "e" "d" "c" "b" "a"
```

The examples below demonstrate what is the result of using a longer vector of indexes than the indexed vector. The length of the indexing vector has no restriction, but the acceptable range of values for the indexes is given by the length of the indexed vector.

```
a[c(3,3,3,3)]

## [1] "c" "c" "c" "c"

a[c(10:1, 1:10)]

##  [1] "j" "i" "h" "g" "f" "e" "d" "c" "b" "a" "a"
## [12] "b" "c" "d" "e" "f" "g" "h" "i" "j"
```

Negative indexes have a special meaning, they indicate the positions at which values should be excluded.

```
a[-2]

## [1] "a" "c" "d" "e" "f" "g" "h" "i" "j"

a[-c(3,2)]

## [1] "a" "d" "e" "f" "g" "h" "i" "j"
```

Results from indexing with out-of-range values may be surprising.

```
a[11]

## [1] NA

a[1:11]

##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" NA
```

Results from indexing with special values may be surprising.

```
a[ ]

##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"

a[numeric(0)]

## character(0)

a[NA]

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

a[c(1, NA)]

## [1] "a" NA
```

```
a[NULL]

## character(0)

a[c(1, NULL)]

## [1] "a"
```

Another way of indexing, which is very handy, but not available in most other programming languages, is indexing with a vector of logical values. In practice, the vector of logical values used for 'indexing' is in most cases of the same length as the vector from which elements are going to be selected. However, this is not a requirement, and if the logical vector is shorter it is 'recycled' as discussed above in relation to operators.

```
a[TRUE]

##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"

a[FALSE]

## character(0)

a[c(TRUE, FALSE)]

## [1] "a" "c" "e" "g" "i"

a[c(FALSE, TRUE)]

## [1] "b" "d" "f" "h" "j"

a > "c"

##  [1] FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE
##  [8]  TRUE  TRUE  TRUE

a[a > "c"]

## [1] "d" "e" "f" "g" "h" "i" "j"

selector <- a > "c"
a[selector]

## [1] "d" "e" "f" "g" "h" "i" "j"

which(a > "c")

## [1]  4  5  6  7  8  9 10

indexes <- which(a > "c")
a[indexes]

## [1] "d" "e" "f" "g" "h" "i" "j"

b <- 1:10
b[selector]

## [1]  4  5  6  7  8  9 10

b[indexes]

## [1]  4  5  6  7  8  9 10
```

A.9 Factors

Factors are used for indicating categories, most frequently the factors describing the treatments in an experiment, or categories in a survey. They can be created either from numerical or character vectors. The different possible values are called *levels*. Normal factors created with `factor` are unordered or categorical. R has ordered factors, that can be created with function `ordered`.

```
my.vector <- c("treated", "treated", "control", "control", "control", "treated")
my.factor <- factor(my.vector)
my.factor <- factor(my.vector, levels=c("treatment", "control"))
```

It is always preferable to use meaningful names for levels, although it is possible to use numbers. The order of levels becomes important when plotting data, as it affects the order of the levels along the axes, or in legends. Converting factors to numbers, even if the levels look like numbers when displayed, they are just character strings.

```
my.vector2 <- rep(3:5, 4)
my.factor2 <- factor(my.vector2)
as.numeric(my.factor2)

##  [1] 1 2 3 1 2 3 1 2 3 1 2 3

as.numeric(as.character(my.factor2))

##  [1] 3 4 5 3 4 5 3 4 5 3 4 5
```

Internally factor levels are stored as running numbers starting from zero, and those are the numbers returned by `as.numeric` applied to a factor.

Factors are very important in R. In contrast to other statistical software in which the role of a variable is set when defining a model to be fitted or setting up a test, in R models are specified exactly in the same way for ANOVA and regression analysis, as linear models. What 'decides' what type of model is fitted is whether the explanatory variable is a factor (giving ANOVA) or a numerical variable (giving regression). This makes a lot of sense, as in most cases, considering an explanatory variable as categorical or not, depends on the design of the experiment or survey, in other words, is a property of the data rather than of the analysis.

A.10 Lists

Elements of a `list` are not ordered, and can be of different type. Lists can be also nested. Elements in list are named, and normally are accessed by name. List are defined using function `list`.

```
a.list <- list(x = 1:6, y = "a", z = c(TRUE, FALSE))
a.list

## $x
## [1] 1 2 3 4 5 6
##
```

```
## $y
## [1] "a"
##
## $z
## [1]  TRUE FALSE

str(a.list)

## List of 3
##  $ x: int [1:6] 1 2 3 4 5 6
##  $ y: chr "a"
##  $ z: logi [1:2] TRUE FALSE

a.list$x

## [1] 1 2 3 4 5 6

a.list[["x"]]

## [1] 1 2 3 4 5 6

a.list[[1]]

## [1] 1 2 3 4 5 6

a.list[1]

## $x
## [1] 1 2 3 4 5 6

a.list[c(1,3)]

## $x
## [1] 1 2 3 4 5 6
##
## $z
## [1]  TRUE FALSE

try(a.list[[c(1,3)]])

## [1] 3
```

Using double square brackets for indexing gives the element stored in the list, in its original mode, in the example above, `a.list[["x"]]` returns a numeric vector, while `a.list[1]` returns a list containing the numeric vector `x`. `a.list$x` returns the same value as `a.list[["x"]]`, a numeric vector. While `a.list[c(1,3)]` returns a list of length two, `a.list[[c(1,3)]]`.

A.11 Data frames

Data frames are a special type of list, in which each element is a vector or a factor of the same length. The are crested with function `data.frame` with a syntax similar to that used for lists. When a shorter vector is supplied as

argument, it is recycled, until the full length of the variable is filled. This is very different to what we obtained in the previous section when we created a list.

```
a.df <- data.frame(x = 1:6, y = "a", z = c(TRUE, FALSE))
a.df

##   x y     z
## 1 1 a  TRUE
## 2 2 a FALSE
## 3 3 a  TRUE
## 4 4 a FALSE
## 5 5 a  TRUE
## 6 6 a FALSE

str(a.df)

## 'data.frame': 6 obs. of  3 variables:
##  $ x: int  1 2 3 4 5 6
##  $ y: Factor w/ 1 level "a": 1 1 1 1 1 1
##  $ z: logi  TRUE FALSE TRUE FALSE TRUE FALSE

a.df$x

## [1] 1 2 3 4 5 6

a.df[["x"]]

## [1] 1 2 3 4 5 6

a.df[[1]]

## [1] 1 2 3 4 5 6

class(a.df)

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

R is an object oriented language, and objects belong to classes. With function `class` we can query the class of an object. As we saw in the two previous chunks lists and data frames objects belong to two different classes.

We can add also to lists and data frames.

```
a.df$x2 <- 6:1
a.df$x3 <- "b"
a.df

##   x y     z x2 x3
## 1 1 a  TRUE  6  b
## 2 2 a FALSE  5  b
## 3 3 a  TRUE  4  b
## 4 4 a FALSE  3  b
## 5 5 a  TRUE  2  b
## 6 6 a FALSE  1  b
```

We have added two columns to the data frame, and in the case of column `x3` recycling took place. Data frames are extremely important to anyone analysing

or plotting data in R. One can think of data frames as tightly structured worksheets, or as lists. As you may have guessed from the examples earlier in this section, there several different ways of accessing columns, rows, and individual observations stored in a data frame. The columns can to some extent be treated as elements in a list, and can be accessed both by name or index (position). When accessed by name, using $ or double square brackets a single column is returned as a vector or factor. In contrast to lists, data frames are 'rectangular' and for this reason the values stored can be also accessed in a way similar to how elements in a matrix are accessed, using two indexes. As we saw for vectors indexes can be vectors of integer numbers or vectors of logical values. For columns they can be vectors of character strings matching the names of the columns. When using indexes it is extremely important to remember that the indexes are always given **row first**.

```
a.df[ , 1]   # first column

## [1] 1 2 3 4 5 6

a.df[ , "x"] # first column

## [1] 1 2 3 4 5 6

a.df[1, ]    # first row

##   x y    z x2 x3
## 1 1 a TRUE  6  b

a.df[1:2, c(FALSE, FALSE, TRUE, FALSE, FALSE)]

## [1]  TRUE FALSE

              # first two rows of the third column
a.df[a.df$z , ] # the rows for which z is true

##   x y    z x2 x3
## 1 1 a TRUE  6  b
## 3 3 a TRUE  4  b
## 5 5 a TRUE  2  b

a.df[a.df$x > 3, -3] # the rows for which x > 3 for

##   x y x2 x3
## 4 4 a  3  b
## 5 5 a  2  b
## 6 6 a  1  b

                  # all columns except the third one
```

When the names of data frames are long, complex conditions become awkward to write. In such cases `subset` is handy because evaluation is done in the 'environment' of the data frame, i.e. the names of the columns are recognized if entered directly.

```
subset(a.df, x > 3)
```

```
##   x y     z x2 x3
## 4 4 a FALSE  3  b
## 5 5 a  TRUE  2  b
## 6 6 a FALSE  1  b
```

When calling functions that return a vector, data farme, or other structure, the square brackets can be appended to the rightmost parenthesis of the function call, in the same way as to the name of a variable holding the same data.

```
subset(a.df, x > 3)[ , -3]

##   x y x2 x3
## 4 4 a  3  b
## 5 5 a  2  b
## 6 6 a  1  b

subset(a.df, x > 3)$x

## [1] 4 5 6
```

None of the examples in the last three code chunks alter the original data frame `a.df`. We can store the returned value using a new name, if we want to preserve `a.df` unchanged, or we can assign the result to `a.df` deleting in the process the original `a.df`. The next to examples do assignment to `a.df`, but either to only one columns, or by indexing the individual values in both the 'right side' and 'left side' of the assignment. Another way to delete a column from a data frame is to assign NULL to it.

```
a.df[["x2"]] <- NULL
a.df$x3 <- NULL
a.df

##   x y     z
## 1 1 a  TRUE
## 2 2 a FALSE
## 3 3 a  TRUE
## 4 4 a FALSE
## 5 5 a  TRUE
## 6 6 a FALSE
```

In the previous code chuck we deleted the last two columns of the data frame `a.df`. Finally an esoteric trick for you think about.

```
a.df[1:6, c(1,3)] <- a.df[6:1, c(3,1)]
a.df

##   x y z
## 1 0 a 6
## 2 1 a 5
## 3 0 a 4
## 4 1 a 3
## 5 0 a 2
## 6 1 a 1
```

A.12 Simple built-in statistical functions

Being R's main focus in statistics, it provides functions for both simple and complex calculations, going from means and variances to fitting very complex models. we will start with the simple ones.

```
x <- 1:20
mean(x)

## [1] 10.5

var(x)

## [1] 35

median(x)

## [1] 10.5

mad(x)

## [1] 7.413

sd(x)

## [1] 5.91608

range(x)

## [1]  1 20

max(x)

## [1] 20

min(x)

## [1] 1

length(x)

## [1] 20
```

A.13 Functions and execution flow control

Although functions can be defined and used at the command prompt, we will discuss them when looking at scripts. We will do the same in the case of flow-control statements (e.g. repetition and conditional execution).

APPENDIX B

R Scripts and Programming

B.1 What is a script?

We call *script* to a text file that contains the same commands that you would type at the console prompt. A true script is not for example an MS-Word file where you have pasted or typed some R commands. A script file has the following characteristics.

- The script is a text file (ASCII or some other encoding e.g. UTF-8 that R uses in your set-up).
- The file contains valid R statements (including comments) and nothing else.
- Comments start at a # and end at the end of the line. (True end-of line as coded in file, the editor may wrap it or not at the edge of the screen).
- The R statements are in the file in the order that they must be executed.
- R scripts have file names ending in `.r`

It is good practice to write scripts so that they will run in a new R session, which means that the script should include library commands to load all the required packages.

B.2 How do we use a scrip?

A script can be sourced.

If we have a text file called `my.first.script.r`

```
# this is my first R script
print(3+4)
```

And then source this file:

```
source("my.first.script.r")

## [1] 7
```

The results of executing the statements contained in the file will appear in the console. The commands themselves are not shown (the sourced file is not echoed) and the results will not be printed unless you include an explicit `print` command. This also applies in many cases also to plots. A fig created with `ggplot` needs to be printed if we want to see it when the script is run.

From within RStudio, if you have an R script open in the editor, there will a "source" drop box (≠ DropBox) visible from where you can choose "source" as described above, or "source with echo" for the currently open file.

When a script is sourced, the output can be saved to a text file instead of being shown in the console. It is also easy to call R with the script file as argument directly at the command prompt of the operating system.

```
RScript my.first.script.r
```

You can open a 'shell' from the Tools menu in RStudio, to run this command. The output will be printed to the shell console. If you would like to save the output to a file, use redirection.

```
RScript my.first.script.r > my.output.txt
```

Sourcing is very useful when the script is ready, however, while developing a script, or sometimes when testing things, one usually wants to run (= execute) one or a few statements at a time. This can be done using the "run" button after either locating the cursor in the line to be executed, or selecting the text that one would like to run (the selected text can be part of a line, a whole line, or a group of lines, as long as it is syntactically valid).

B.3 How to write a script?

The approach used, or mix of approaches will depend on your preferences, and on how confident you are that the statements will work as expected.

If one is very familiar with similar problems One would just create a new text file and write the whole thing in the editor, and then test it. This is rather unusual.

If one if moderately familiar with the problem One would write the script as above, but testing it, part by part as one is writing it. This is usually what I do.

If ones mostly playing around Then if one is using RStudio, one type statements at the console prompt. As you should know by now, everything you run at the console is saved to the "History". In RStudio the History is displayed in its own pane, and in this pane one can select any previous

statement and by pressing a single having copy and pasted to either the console prompt, or the cursor position in the file visible in the editor. In this way one can build a script by copying and pasting from the history to your script file the bits that have worked as you wanted.

B.4 The need to be understandable to people

When you write a script, it is either because you want to document what you have done or you want re-use it at a later time. In either case, the script itself although still meaningful for the computer could become very obscure to you, and even more to someone seeing it for the first time.

How does one achieve an understandable script or program?

- Avoid the unusual. People using a certain programming language tend to use some implicit or explicit rules of style. As a minimum try to be consistent with yourself.
- Use meaningful names for variables, and any other object. What is meaningful depends on the context. Depending on common use a single letter may be more meaningful than a long word. However self explaining names are better: e.g. using `n.rows` and `n.cols` is much clearer than using `n1` and `n2` when dealing with a matrix of data. Probably `number.of.rows` and `number.of.columns` would just increase the length of the lines in the script, and one would spend more time typing without getting much in return.
- How to make the words visible in names: traditionally in R one would use dots to separate the words and use only lower case. Some years ago, it became possible to use underscores. The use of underscores is quite common nowadays because in some contexts is "safer" as in special situations a dot may have a special meaning. What we call "camel case" is very rarely used in R programming but is common in other languages like Pascal. An example of camel case is `NumCols`. In some cases it can become a bit confusing as in `UVMean` or `UvMean`.

B.5 Exercises

By now you should be familiar enough with R to be able to write your own script.

1. Create a new R script (in RStudio, from 'File' menu, "+" button, or by typing "Ctrl + Shift + N").
2. Save the file as "my.second.script.r".
3. Use the editor pane in RStudio to type some R commands and comments.
4. **Run** individual commands.
5. **Source** the whole file.

B.6 Functions

When writing scripts, or any program, one should avoid repeating code (groups of statements). The reasons for this are: 1) if the code needs to be changed, you have to make changes in more than one place in the file, or in more than one file. Sooner or later, some copies will remain unchanged by mistake. 2) it makes the script file longer, and this makes debugging, commenting, etc. more tedious, and error prone.

How do we avoid repeating bits of code? We write a function containing the statements that we would need to repeat, and then `call` the function in their place.

Functions are defined by means of **function**, and saved like any other object in R by assignment a variable. `x` is a parameter, the name used within the function for an object that will be supplied as "argument" when the function is called. One can think of parameter names as place-holders.

```
my.prod <- function(x, y){x * y}
my.prod(4, 3)

## [1] 12
```

First some basic knowledge. In R, arguments are passed by copy. This is something very important to remember. Whatever you do within a function to the passed argument, its value outside the function will remain unchanged.

```
my.change <- function(x){x <- NA}
a <- 1
my.change(a)
a

## [1] 1
```

Any result that needs to be made available outside the function must be returned by the function. If the function `return` is not explicitly used, the value returned by the last statement within the body of the function will be returned.

```
print.x.1 <- function(x){print(x)}
print.x.1("test")

## [1] "test"

print.x.2 <- function(x){print(x); return(x)}
print.x.2("test")

## [1] "test"
## [1] "test"

print.x.3 <- function(x){return(x); print(x)}
print.x.3("test")

## [1] "test"

print.x.4 <- function(x){return(); print(x)}
print.x.4("test")

## NULL
```

We can assign to a variable defined outside a function with operator «- but the usual recommendation is to avoid its use. This type of effects of calling a function are frequently called 'side-effects'.

Now we will define a useful function: a function for calculating the standard error of the mean from a numeric vector.

```
SEM <- function(x){sqrt(var(x)/length(x))}
a <- c(1, 2, 3, -5)
a.na <- c(a, NA)
SEM(x=a)

## [1] 1.796988

SEM(a)

## [1] 1.796988

SEM(a.na)

## [1] NA
```

For example in `SEM(a)` we are calling function `SEM` with `a` as argument.

The function we defined above may sometimes give a wrong answer because NAs will be counted by `length`, so we need to remove NAs before calling `length`.

```
SEM <- function(x) sqrt(var(x, na.rm=TRUE)/length(na.omit(x)))
a <- c(1, 2, 3, -5)
a.na <- c(a, NA)
SEM(x=a)

## [1] 1.796988

SEM(a)

## [1] 1.796988

SEM(a.na)

## [1] 1.796988
```

R does not have a function for standard error, so the function above would be generally useful. If we would like to make this function both safe, and consistent with other R functions, one could define it as follows, allowing the user to provide a second argument which is passed as an argument to `var`:

```
SEM <- function(x, na.rm=FALSE){sqrt(var(x, na.rm=na.rm)/length(na.omit(x)))}
SEM(a)

## [1] 1.796988

SEM(a.na)

## [1] NA

SEM(a.na, TRUE)
```

```
## [1] 1.796988

SEM(x=a.na, na.rm=TRUE)

## [1] 1.796988

SEM(TRUE, a.na)

## Warning in if (na.rm) "na.or.complete" else "everything":
the condition has length > 1 and only the first element will be
used

## [1] NA

SEM(na.rm=TRUE, x=a.na)

## [1] 1.796988
```

In this example you can see that functions can have more than one parameter, and that parameters can have default values to be used if no argument is supplied. In addition if the name of the parameter is indicated, then arguments can be supplied in any order, but if parameter names are not supplied, then arguments are assigned to parameters based on their position. Once one parameter name is given, all later arguments need also to be explicitly matched to parameters. Obviously if given by position, then arguments should be supplied explicitly for all parameters at 'intermediate' positions.

B.7 R built-in functions

B.7.1 Plotting

The built-in generic function `plot` can be used to plot data. It is a generic function, that has suitable methods for different kinds of objects.

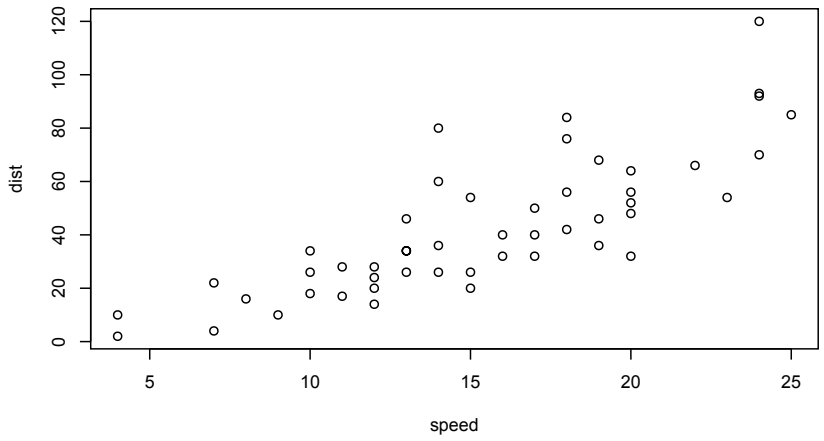
Before we can plot anything, we need some data.

```
data(cars)
names(cars)

## [1] "speed" "dist"

head(cars)

##   speed dist
## 1     4    2
## 2     4   10
## 3     7    4
## 4     7   22
## 5     8   16
## 6     9   10

tail(cars)

##    speed dist
## 45    23   54
## 46    24   70
```

```
## 47    24   92
## 48    24   93
## 49    24  120
## 50    25   85
```

`cars` is an example data set that is included in R. It is stored as a dataframe. Data frames are used for storing data, they consist in columns of equal length. The different columns can be different types (e.g. numeric and character). With `data` we load it; with `names` we obtain the names of the variables or columns. With head with can see the top several lines, and with tail the lines at the end.

```
plot(dist ~ speed, data=cars)
```



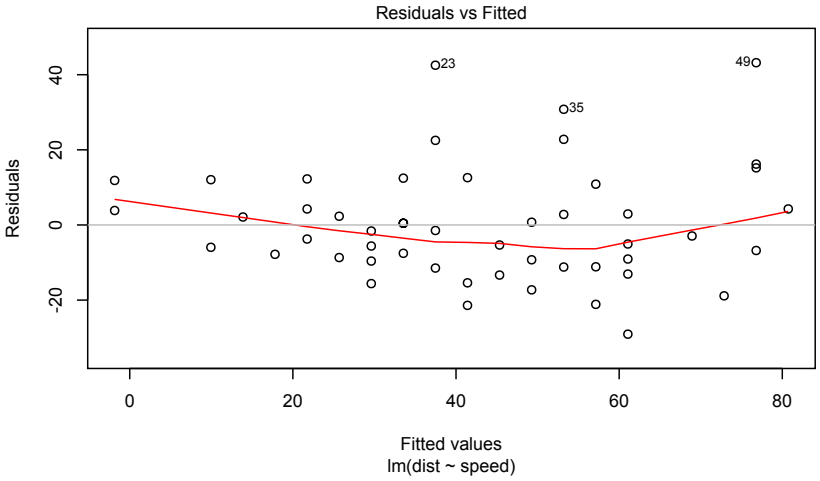
B.7.2 Fitting linear models

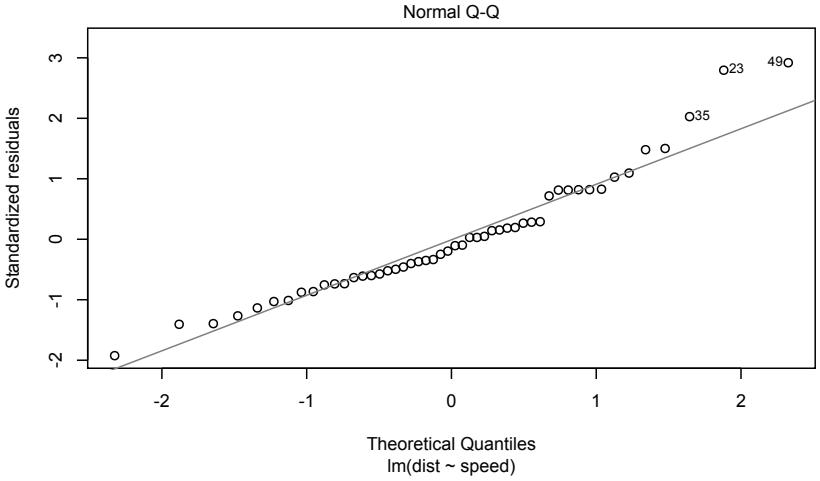
Regression

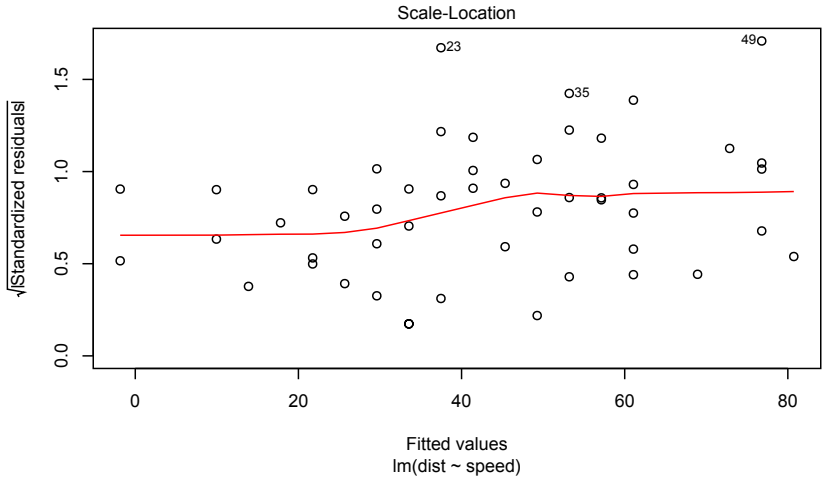
The R function `lm` is used next to fit a linear regression.

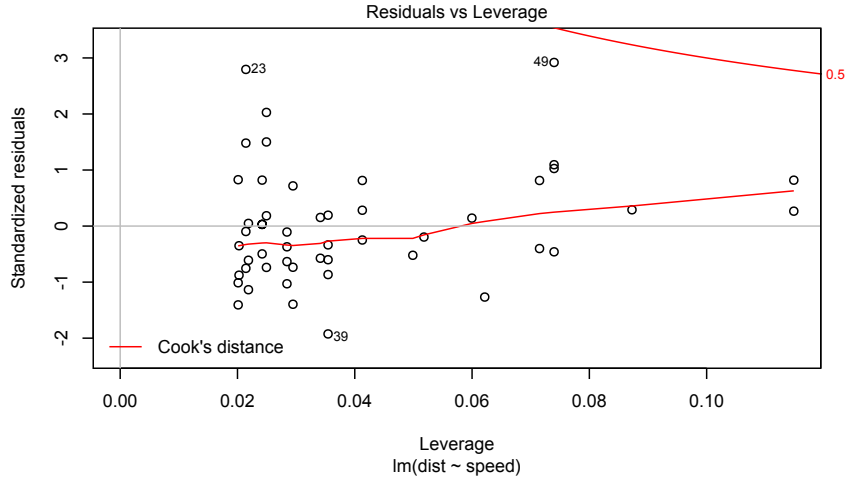
```
fm1 <- lm(dist ~ speed, data=cars) # we fit a model, and then save the result
plot(fm1) # we produce diagnosis plots
summary(fm1) # we inspect the results from the fit

##
## Call:
## lm(formula = dist ~ speed, data = cars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -29.069  -9.525  -2.272   9.215  43.201
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -17.5791     6.7584  -2.601   0.0123
## speed         3.9324     0.4155   9.464 1.49e-12
##
```

```
## (Intercept) *
## speed       ***
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 15.38 on 48 degrees of freedom
## Multiple R-squared:  0.6511,Adjusted R-squared:  0.6438
## F-statistic: 89.57 on 1 and 48 DF,  p-value: 1.49e-12

anova(fm1) # we calculate an ANOVA

## Analysis of Variance Table
##
## Response: dist
##           Df Sum Sq Mean Sq F value   Pr(>F)
## speed      1  21186 21185.5  89.567 1.49e-12 ***
## Residuals 48  11354   236.5
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```


Residuals vs Fitted
Residuals
Fitted values
lm(dist ~ speed)
23
35
49
40
20
0
-20
0
20
40
60
80


Normal Q-Q
Standardized residuals
Theoretical Quantiles
lm(dist ~ speed)
23
35
49
3
2
1
0
-1
-2
-2
-1
0
1
2

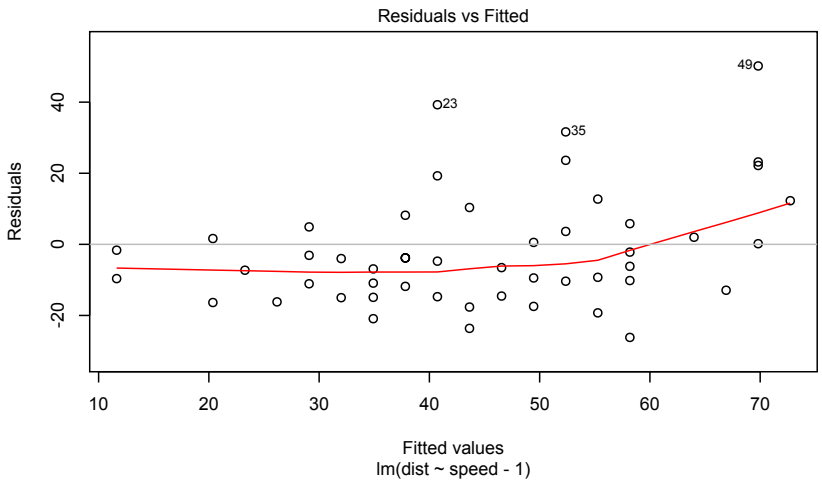


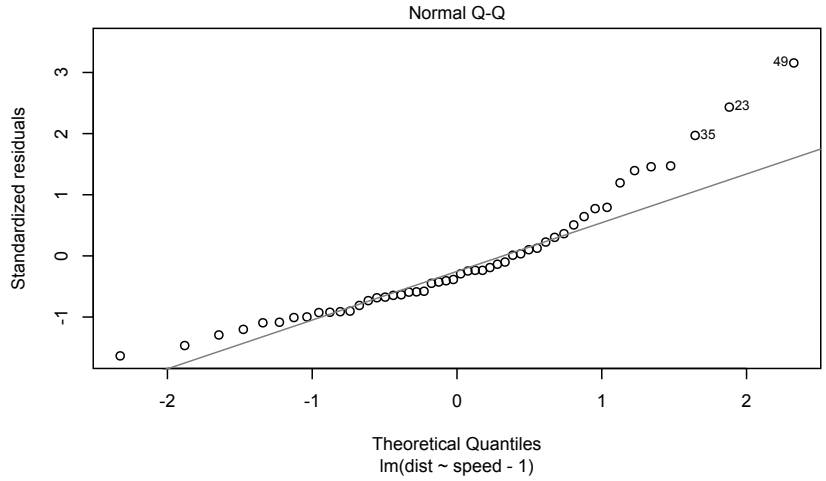


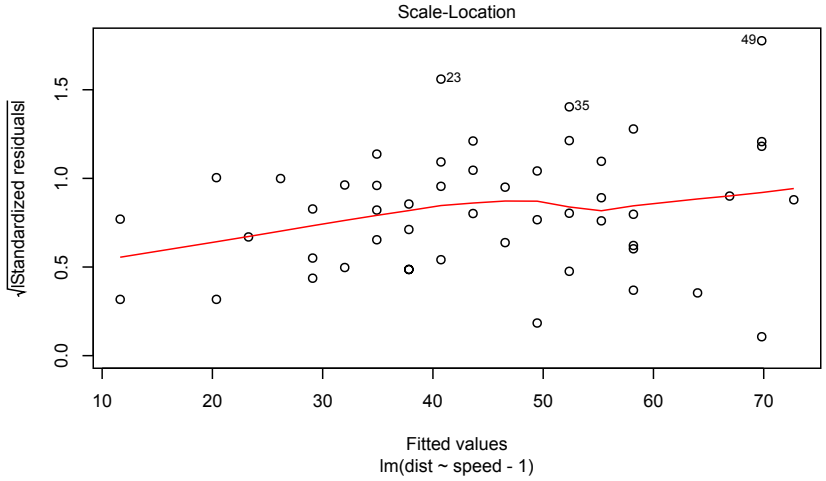
Let's look at each step separately: `dist ~ speed` is the specification of the model to be fitted. The intercept is always implicitly included. To 'remove' this implicit intercept from the earlier model we can use `dist ~ speed - 1`.

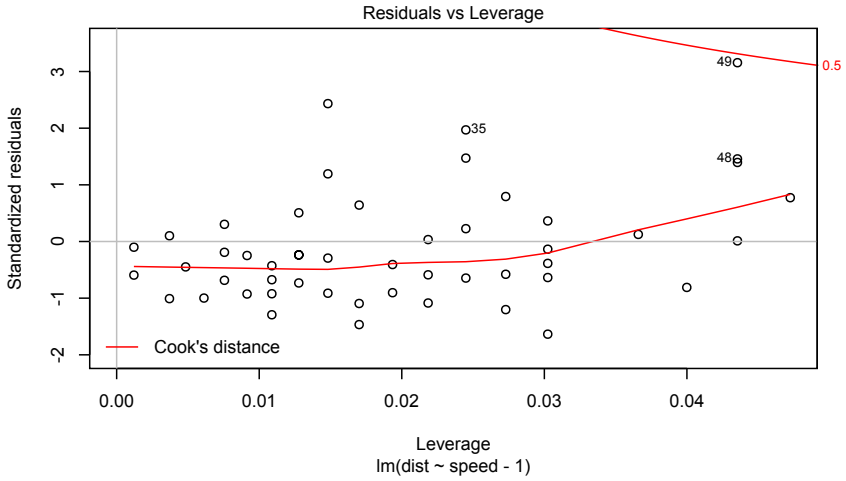
```
fm2 <- lm(dist ~ speed - 1, data=cars) # we fit a model, and then save the result
plot(fm2) # we produce diagnosis plots
summary(fm2) # we inspect the results from the fit

##
## Call:
## lm(formula = dist ~ speed - 1, data = cars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -26.183 -12.637  -5.455   4.590  50.181
##
## Coefficients:
```

```
##       Estimate Std. Error t value Pr(>|t|)
## speed   2.9091     0.1414   20.58   <2e-16 ***
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 16.26 on 49 degrees of freedom
## Multiple R-squared:  0.8963,Adjusted R-squared:  0.8942
## F-statistic: 423.5 on 1 and 49 DF,  p-value: < 2.2e-16

anova(fm2) # we calculate an ANOVA

## Analysis of Variance Table
##
## Response: dist
##           Df Sum Sq Mean Sq F value    Pr(>F)
## speed      1 111949  111949  423.47 < 2.2e-16 ***
## Residuals 49  12954     264
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```



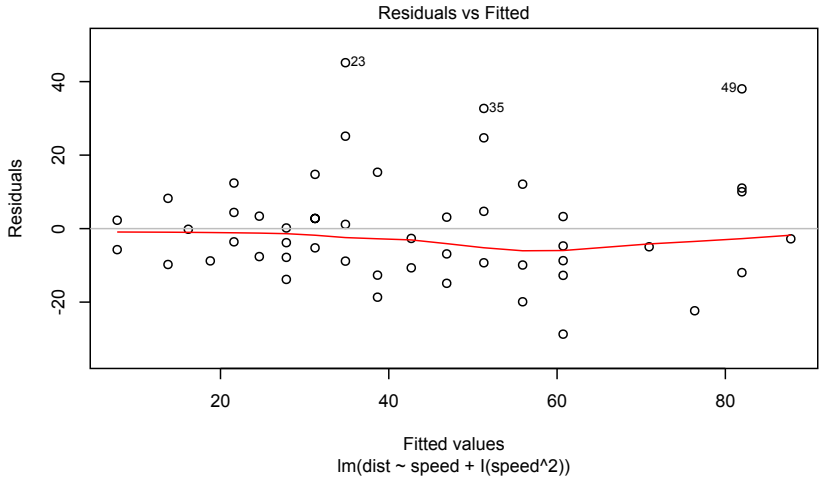

Normal Q-Q
Standardized residuals
Theoretical Quantiles
lm(dist ~ speed - 1)
49
23
35

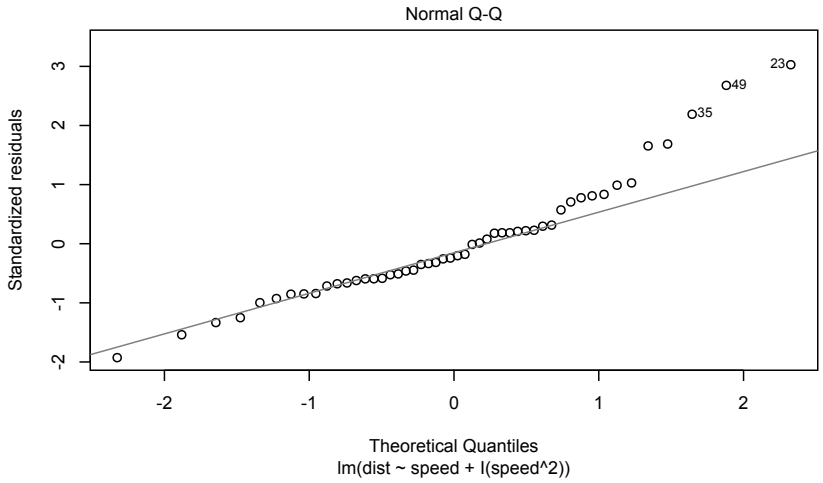

Scale-Location
√|Standardized residuals|
Fitted values
lm(dist ~ speed - 1)
49
23
35

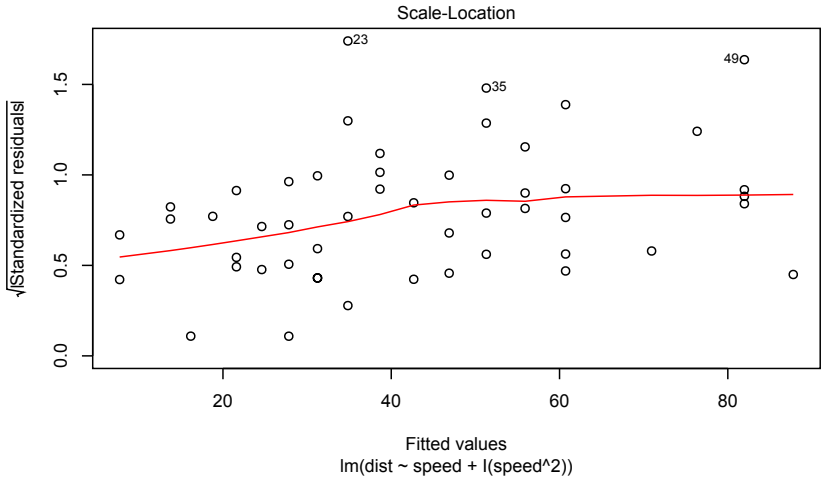


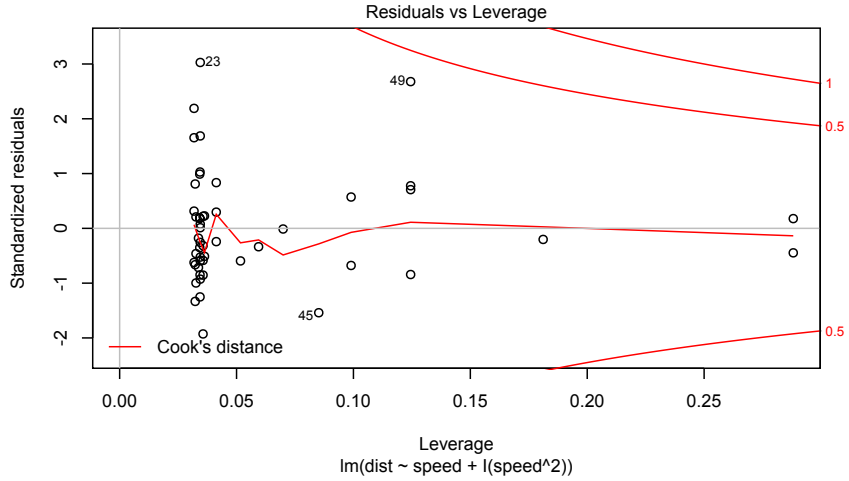
We now we fit a second degree polynomial.

```
fm3 <- lm(dist ~ speed + I(speed^2), data=cars) # we fit a model, and then save the result
plot(fm3) # we produce diagnosis plots
summary(fm3) # we inspect the results from the fit

##
## Call:
## lm(formula = dist ~ speed + I(speed^2), data = cars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -28.720  -9.184  -3.188   4.628  45.152
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  2.47014   14.81716   0.167    0.868
## speed        0.91329    2.03422   0.449    0.656
## I(speed^2)   0.09996    0.06597   1.515    0.136
##
## Residual standard error: 15.18 on 47 degrees of freedom
## Multiple R-squared:  0.6673,Adjusted R-squared:  0.6532
## F-statistic: 47.14 on 2 and 47 DF,  p-value: 5.852e-12

anova(fm3) # we calculate an ANOVA

## Analysis of Variance Table
##
## Response: dist
##            Df  Sum Sq Mean Sq F value    Pr(>F)
## speed       1 21185.5 21185.5  91.986 1.211e-12
## I(speed^2)  1   528.8   528.8   2.296    0.1364
## Residuals  47 10824.7   230.3
##
## speed      ***
## I(speed^2)
## Residuals
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```


Residuals vs Fitted
Residuals
Fitted values
lm(dist ~ speed + I(speed^2))
23
35
49


Normal Q-Q
Standardized residuals
Theoretical Quantiles
lm(dist ~ speed + I(speed^2))
23
49
35





We can also compare the two models.

```
anova(fm2, fm1)

## Analysis of Variance Table
##
## Model 1: dist ~ speed - 1
## Model 2: dist ~ speed
##   Res.Df   RSS Df Sum of Sq      F  Pr(>F)
## 1     49 12954
## 2     48 11354  1    1600.3 6.7655 0.01232 *
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Or three or more models. But be careful, as the order of the arguments matters.

```
anova(fm2, fm1, fm3)

## Analysis of Variance Table
##
## Model 1: dist ~ speed - 1
## Model 2: dist ~ speed
## Model 3: dist ~ speed + I(speed^2)
##   Res.Df   RSS Df Sum of Sq      F  Pr(>F)
## 1     49 12954
## 2     48 11354  1   1600.26 6.9482 0.01133 *
## 3     47 10825  1    528.81 2.2960 0.13640
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

We can use different criteria to choose the best model: significance based on *P*-values or information criteria (AIC, BIC) that penalize the result based on the number of parameters in the fitted model. In the case of AIC and BIC, a smaller value is better, and values returned can be either positive or negative, in which case more negative is better.

B.8 Control of execution flow

B.8.1 Conditional execution

Non-vectorized

R has two types of "if" statements, non-vectorized and vectorized. We will start with the non-vectorized one, which is similar to what is available in most other computer programming languages.

Before this we need to explain compound statements. Individual statements can be grouped into compound statements by enclosed them in curly braces.

```
print("A")

## [1] "A"

{
  print("B")
  print("C")
}

## [1] "B"
## [1] "C"
```

The example above is pretty useless, but becomes useful when used together with 'control' constructs. The `if` construct controls the execution of one statement, however, this statement can be a compound statement of almost any length or complexity. Play with the code below by changing the value assigned to `printing`, including NA, and logical(0).

```
printing <- TRUE
if (printing) {
  print("A")
```

```
  print("B")
}

## [1] "A"
## [1] "B"
```

The condition '( )' can be anything yielding a logical vector, however, as this is not vectorized, only the first element will be used. Play with this example by changing the value assigned to `a`.

```
a <- 10.0
if (a < 0.0) print("'a' is negative") else print("'a' is not negative")

## [1] "'a' is not negative"

print("This is always printed")

## [1] "This is always printed"
```

As you can see above the statement immediately following `else` is executed if the condition is false. Later statements are executed independently of the condition.

Do you still remember the rules about continuation lines?

```
# 1
if (a < 0.0)
  print("'a' is negative") else
    print("'a' is not negative")
# 2 (not evaluated here)
if (a < 0.0) print("'a' is negative")
else print("'a' is not negative")
```

Why does only the second example above trigger an error?

Play with the use conditional execution, with both simple and compound statements, and also think how to combine `if` and `else` to select among more than two options.

There is in R a `switch` statement, that we will not describe here, that can be used to select among "cases", or several alternative statements, based on an expression evaluating to a number or a character string.

Vectorized

The vectorized conditional execution is coded by means of a **function** called `ifelse` (one word). This function takes three arguments: a logical vector, a result vector for TRUE, a result vector for FALSE. All three can be any construct giving the necessary argument as their result. In the case of result vectors, recycling will apply if they are not of the correct length. The length of the result is determined by the length of the logical vector in the first argument!.

```
a <- 1:10
ifelse(a > 5, 1, -1)

##  [1] -1 -1 -1 -1 -1  1  1  1  1  1
```

```
ifelse(a > 5, a + 1, a - 1)

##  [1]  0  1  2  3  4  7  8  9 10 11

ifelse(any(a>5), a + 1, a - 1) # tricky

## [1] 2

ifelse(logical(0), a + 1, a - 1) # even more tricky

## logical(0)

ifelse(NA, a + 1, a - 1) # as expected

## [1] NA
```

Try to understand what is going on in the previous example. Create your own examples to test how `ifelse` works.

Exercise: write using `ifelse` a single statement to combine numbers from a and b into a result vector d, based on whether the corresponding value in c is the character "a" or "b".

```
a <- rep(-1, 10)
b <- rep(+1, 10)
c <- c(rep("a", 5), rep("b", 5))
# your code
```

If you do not understand how the three vectors are built, or you cannot guess the values they contain by reading the code, print them, and play with the arguments, until you have clear what each parameter does.

B.8.2 Why using vectorized functions and operators is important

If you have written programs in other languages, it would feel to you natural to use loops (for, repeat while, repeat until) for many of the things for which we have been using vectorization. When using the R language it is best to use vectorization whenever possible, because it keeps the listing of scripts and programs shorter and easier to understand (at least for those with experience in R). However, there is another very important reason: execution speed. The reason behind this is that R is an interpreted language. In current versions of R it is possible to byte-compile functions, but this is rarely used for scripts, and even byte-compiled loops are much slower and vectorized functions.

However, there are cases were we need to repeatedly execute statements in a way that cannot be vectorized, or when we do not need to maximize execution speed. The R language does have loop constructs, and we will describe them next.

B.8.3 Repetition

The most frequently used type of loop is a `for` loop. These loops work in R are based on lists or vectors of values to act upon.

```
b <- 0
for (a in 1:5) b <- b + a
b

## [1] 15

b <- sum(1:5) # built-in function
b

## [1] 15
```

Here the statement `b <- b + a` is executed five times, with a sequentially taking each of the values in `1:5`. Instead of a simple statement used here, also a compound statement could have been used.

Here are a few examples that show some of the properties of `for` loops and functions, combined with the use of a function.

```
test.for <- function(x) {
  for (i in x) {print(i)}
}
test.for(numeric(0))
test.for(1:3)

## [1] 1
## [1] 2
## [1] 3

test.for(NA)

## [1] NA

test.for(c("A", "B"))

## [1] "A"
## [1] "B"

test.for(c("A", NA))

## [1] "A"
## [1] NA

test.for(list("A", 1))

## [1] "A"
## [1] 1

test.for(c("z", letters[1:4]))

## [1] "z"
## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
```

In contrast to other languages, in R function arguments are not checked for 'type' when the function is called. The only requirement is that the function code can handle the argument provided. In this example you can see that the

same function works with numeric and character vectors, and with lists. We haven't seen lists before. As earlier discussed all elements in a vector should have the same type. This is not the case for lists. It is also interesting to note that a list or vector of length zero is a valid argument, that triggers no error, but that as one would expect, causes the statements in the loop body to be skipped.

Some examples of use of `for` loops — and of how to avoid there use.

```
a <- c(1, 4, 3, 6, 8)
for(x in a) x*2 # result is lost
for(x in a) print(x*2) # print is needed!

## [1] 2
## [1] 8
## [1] 6
## [1] 12
## [1] 16

b <- for(x in a) x*2 # doesn't work as expected, but triggers no error
b

## NULL

for(x in a) b <- x*2 # a bit of a surprise, as b is not a vector!
b

## [1] 16

for(i in seq(along=a)) {
  b[i] <- a[i]^2
  print(b)
}

## [1] 1
## [1]  1 16
## [1]  1 16  9
## [1]  1 16  9 36
## [1]  1 16  9 36 64

b # is a vector!

## [1]  1 16  9 36 64

# a bit faster if we first allocate a vector of the required length
b <- numeric(length(a))
for(i in seq(along=a)) {
  b[i] <- a[i]^2
  print(b)
}

## [1] 1 0 0 0 0
## [1]  1 16  0  0  0
## [1]  1 16  9  0  0
## [1]  1 16  9 36  0
## [1]  1 16  9 36 64

b # is a vector!
```

```
## [1]  1 16  9 36 64

# vectorization is simplest and fastest
b <- a^2
b

## [1]  1 16  9 36 64
```

Look at the results from the above examples, and try to understand where does the returned value come from in each case.

We sometimes may not be able to use vectorization, or may be easiest to not use it. However, whenever working with large data sets, or many similar datasets, we will need to take performance into account. As vectorization usually also makes code simpler, it is good style to use it whenever possible.

```
b <- numeric(length(a)-1)
for(i in seq(along=b)) {
  b[i] <- a[i+1] - a[i]
  print(b)
}

## [1] 3 0 0 0
## [1]  3 -1  0  0
## [1]  3 -1  3  0
## [1]  3 -1  3  2

# although in this case there were alternatives, there
# are other cases when we need to use indexes explicitly
b <- a[2:length(a)] - a[1:length(a)-1]
b

## [1]  3 -1  3  2

# or even better
b <- diff(a)
b

## [1]  3 -1  3  2
```

`seq(along=b)` builds a new numeric vector with a sequence of the same length as the length as the vector given as argument for parameter 'along'.

`while` loops are quite frequently also useful. Instead of a list or vector, they take a logical argument, which is usually an expression, but which can also be a variable. For example the previous calculation could be also done as follows.

```
a <- c(1, 4, 3, 6, 8)
i <- 1
while (i < length(a)) {
  b[i] <- a[i]^2
  print(b)
  i <- i + 1
}

## [1]  1 -1  3  2
## [1]  1 16  3  2
```

```
## [1]  1 16  9  2
## [1]  1 16  9 36

b

## [1]  1 16  9 36
```

Here is another example. In this case we use the result of the previous iteration in the current one. In this example you can also see, that it is allowed to put more than one statement in a single line, in which case the statements should be separated by a semicolon (;).

```
a <- 2
while (a < 50) {print(a); a <- a^2}

## [1] 2
## [1] 4
## [1] 16

print(a)

## [1] 256
```

Make sure that you understand why the final value of `a` is larger than 50.

`repeat` is seldom used, but adds flexibility as `break` can be in the middle of the compound statement.

```
a <- 2
repeat{
  print(a)
  a <- a^2
  if (a > 50) {print(a); break()}
}

## [1] 2
## [1] 4
## [1] 16
## [1] 256

# or more elegantly
a <- 2
repeat{
  print(a)
  if (a > 50) break()
  a <- a^2
}

## [1] 2
## [1] 4
## [1] 16
## [1] 256
```

Please, make sure you understand what is happening in the previous examples.

B.8.4 Nesting

All the execution-flow control statements seen above can be nested. We will show an example with two `for` loops. We first need a matrix of data to work with:

```
A <- matrix(1:50, 10)
A

##       [,1] [,2] [,3] [,4] [,5]
##  [1,]    1   11   21   31   41
##  [2,]    2   12   22   32   42
##  [3,]    3   13   23   33   43
##  [4,]    4   14   24   34   44
##  [5,]    5   15   25   35   45
##  [6,]    6   16   26   36   46
##  [7,]    7   17   27   37   47
##  [8,]    8   18   28   38   48
##  [9,]    9   19   29   39   49
## [10,]   10   20   30   40   50

A <- matrix(1:50, 10, 5)
A

##       [,1] [,2] [,3] [,4] [,5]
##  [1,]    1   11   21   31   41
##  [2,]    2   12   22   32   42
##  [3,]    3   13   23   33   43
##  [4,]    4   14   24   34   44
##  [5,]    5   15   25   35   45
##  [6,]    6   16   26   36   46
##  [7,]    7   17   27   37   47
##  [8,]    8   18   28   38   48
##  [9,]    9   19   29   39   49
## [10,]   10   20   30   40   50

# argument names used for clarity
A <- matrix(1:50, nrow = 10)
A

##       [,1] [,2] [,3] [,4] [,5]
##  [1,]    1   11   21   31   41
##  [2,]    2   12   22   32   42
##  [3,]    3   13   23   33   43
##  [4,]    4   14   24   34   44
##  [5,]    5   15   25   35   45
##  [6,]    6   16   26   36   46
##  [7,]    7   17   27   37   47
##  [8,]    8   18   28   38   48
##  [9,]    9   19   29   39   49
## [10,]   10   20   30   40   50

A <- matrix(1:50, ncol = 5)
A

##       [,1] [,2] [,3] [,4] [,5]
##  [1,]    1   11   21   31   41
##  [2,]    2   12   22   32   42
##  [3,]    3   13   23   33   43
```

```
##  [4,]    4   14   24   34   44
##  [5,]    5   15   25   35   45
##  [6,]    6   16   26   36   46
##  [7,]    7   17   27   37   47
##  [8,]    8   18   28   38   48
##  [9,]    9   19   29   39   49
## [10,]   10   20   30   40   50

A <- matrix(1:50, nrow = 10, ncol = 5)
A

##       [,1] [,2] [,3] [,4] [,5]
##  [1,]    1   11   21   31   41
##  [2,]    2   12   22   32   42
##  [3,]    3   13   23   33   43
##  [4,]    4   14   24   34   44
##  [5,]    5   15   25   35   45
##  [6,]    6   16   26   36   46
##  [7,]    7   17   27   37   47
##  [8,]    8   18   28   38   48
##  [9,]    9   19   29   39   49
## [10,]   10   20   30   40   50
```

All the statements above are equivalent, but some are easier to read than others.

```
row.sum <- numeric() # slower as size needs to be expanded
for (i in 1:nrow(A)) {
  row.sum[i] <- 0
  for (j in 1:ncol(A))
    row.sum[i] <- row.sum[i] + A[i, j]
}
print(row.sum)

##  [1] 105 110 115 120 125 130 135 140 145 150
```

```
row.sum <- numeric(nrow(A)) # faster
for (i in 1:nrow(A)) {
  row.sum[i] <- 0
  for (j in 1:ncol(A))
    row.sum[i] <- row.sum[i] + A[i, j]
}
print(row.sum)

##  [1] 105 110 115 120 125 130 135 140 145 150
```

Look at the output of these two examples to understand what is happening differently with `row.sum`.

The code above is very general, it will work with any size of two dimensional matrix, which is good programming practice. However, sometimes we need more specific calculations. `A[1, 2]` selects one cell in the matrix, the one on the first row of the second column. `A[1, ]` selects row one, and `A[ , 2]` selects column two. In the example above the value of `i` changes for each iteration of the outer loop. The value of `j` changes for each iteration of the

inner loop, and the inner loop is run in full for each iteration of the outer loop. The inner loop index `j` changes fastest.

Exercises: 1) modify the example above to add up only the first three columns of A, 2) modify the example above to add the last three columns of A.

Will the code you wrote continue working as expected if the number of rows in A changed? and what if the number of columns in A changed, and the required results still needed to be calculated for relative positions? What would happen if A had fewer than three columns? Try to think first what to expect based on the code you wrote. Then create matrices of different sizes and test your code. After that think how to improve the code, at least so that wrong results are not produced.

Vectorization can be achieved in this case easily for the inner loop.

```
row.sum <- numeric(nrow(A)) # faster
for (i in 1:nrow(A)) {
  row.sum[i] <- sum(A[i, ])
}
print(row.sum)

##  [1] 105 110 115 120 125 130 135 140 145 150
```

`A[i, ]` selects row `i` and all columns. In R, the row index always comes first, which is not the case in all programming languages.

Full vectorization can be achieved with `apply` functions.

```
row.sum <- apply(A, MARGIN = 1, sum) # MARGIN=1 inidcates rows
print(row.sum)

##  [1] 105 110 115 120 125 130 135 140 145 150
```

How would you change this last example, so that only the last three columns are added up? (Think about use of subscripts to select a part of the matrix.)

There are many variants of `apply` functions, both in base R and in contributed packages.

B.9 Packages

In R speak 'library' is the location where 'packages' are installed. Packages are sets of functions, and data, specific for some particular purpose, that can be loaded into an R session to make them available so that they can be used in the same way as built-in R functions and data. The function `library` is used to load packages, already installed in the local R library, into the current session, while the function `install.packages` is used to install packages, either from a file, or directly from the internet into the library. When using RStudio it is easiest to use RStudio commands (which call `install.packages` and `update.packages`) to install and update packages.

```
library(graphics)
```

Currently there are thousands of packages available. The most reliable source of packages is CRAN, as only packages that pass strict tests and are

actively maintained are included. In some cases you may need or want to install less stable code, and this is also possible.

R packages can be installed either from source, or from already built 'binaries'. Installing from sources, depending on the package, may require quite a lot of additional software to be available. Under MS-Windows, very rarely the needed shell, commands and compilers are already available. Installing then is not too difficult (you will need RTools, and MiKTeX). For this reason it is the norm to install packages from binary .zip files. Under Linux most tools will be available, or very easy to install, so it is not unusual to install from sources. For OS X (Mac) the situation is somewhere in-between. If the tools are available, packages can be very easily installed from source from within RStudio.

The development of packages is beyond the scope of the current course, but it is still interesting to know a few things about packages. Using RStudio it is relatively easy to develop your own packages. Packages can be of very different sizes. Packages use a relatively rigid structure of folder for storing the different types of files, and there is a built-in help system, that one needs to use, so that the package documentation gets linked to the R help system when the package is loaded. In addition to R code, packages can call C, C++, FORTRAN, Java, etc. functions and routines, but some kind of 'glue' is needed, as data is stored differently. At least for C++, the recently developed Rcpp R package makes the gluing extremely easy.

In addition to some packages from CRAN, later in the course we will use a suite of packages for photobiology that I have developed during the last couple of years. Some of the functions in these packages are very simple, and others more complex. In one of the packages, I included some C++ functions to improve performance. Replacing some R for loops with C++ for loops and iterators, resulted in a huge speed increase. The reason for this is that R is an interpreted language and C++ is compiled into machine code. Recent versions of R allow byte-compilation which can give some speed improvement, without need to switch to another language.

The source code for the photobiology and many other packages is freely available, so if you are interested you can study it. For any function defined in R, typing at the command prompt the name of the function without the parentheses lists the code.

```
length  # a function defined in C within R itself

## function (x)  .Primitive("length")

SEM # the function we defined earlier

## function(x, na.rm=FALSE){sqrt(var(x, na.rm=na.rm)/length(na.omit(x)))}
```

One good way of learning how R works, is by experimenting with it, and whenever using a certain function looking at the help, to check what are all the available options.

APPENDIX C

Storing and manipulating data with R

C.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(data.table)
```

C.2 Introduction

Data frames have been discussed in A and data tables are also data frames. In other words they are 'derived' from data frames, so they can be used whenever data frames are expected as input. Package `data.frame` is under active development but it is already a better option than data frames in many contexts, specially when working with large data sets. The reason for this is that because of the way the R language is defined, data frames are are very frequently copied in whole even when a small part of the data is altered, or when passed as arguments to many functions. This has a very large impact on performance. The `data.table` avoids or delays copying as much as possible, and also implements fast search, sort, etc. operations. This makes a huge difference for large data sets. For smaller data set the main advantage is the new (additional) syntax that is more concise, though not in all cases easier to understand.

C.3 Differences between `data.tables` and `data.frames`

Data tables are also data frames, and if one operates on them with the usual data-frame syntax, in most cases they behave identically to data frames. To achieve the full advantage in performance one should be careful of one codes scripts and functions. Data tables can created in a similar way as data frames.

```
my.dt <- data.table(x = 1:10, y = rnorm(10))
class(my.dt)

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

We can 'convert' in place, without any copying, a data frame into a data table using `setDT`, and with `setDF` 'convert' a data table into a data frame.

```
my.df <- data.frame(x = 1:10, y = rnorm(10))
class(my.df)

## [1] "data.frame"

setDT(my.df)
class(my.df)

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

setDF(my.df)
class(my.df)

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

An assignment of a data frame is equivalent to a copy, and in most cases results in the whole data frame being copyed from one location in memory to a different one.

```
my.cp.df <- my.df
identical(my.cp.df, my.df)

## [1] TRUE

my.cp.df$y <- 1
identical(my.cp.df, my.df)

## [1] FALSE
```

With data tables, assignment with <- just creates a new name for the same object. However, if we use 'data.frame' syntax to alter the new name, a copy is done at that moment, and yields the same result as a true data frame.

```
my.cp.dt <- my.dt
identical(my.cp.dt, my.dt)

## [1] TRUE

my.cp.dt$y <- my.cp.dt$y + 1
identical(my.cp.dt, my.dt)

## [1] FALSE
```

However, if we use the special syntax introduced by the `data.frame` package, no copy is done, and both names continue pointing to the same, now modified data table.

```
my.cp.dt <- my.dt
identical(my.cp.dt, my.dt)

## [1] TRUE

my.cp.dt[ , y := y + 1]

##       x          y
##  1:   1  0.9483857
##  2:   2  1.0988303
##  3:   3  0.3966207
##  4:   4  2.4108128
##  5:   5 -0.1902492
##  6:   6 -0.4782063
##  7:   7  2.0325554
##  8:   8  1.4157829
##  9:   9  0.9652285
## 10:  10  0.5958212

identical(my.cp.dt, my.dt)

## [1] TRUE

my.cp.dt

##       x          y
##  1:   1  0.9483857
##  2:   2  1.0988303
##  3:   3  0.3966207
##  4:   4  2.4108128
##  5:   5 -0.1902492
##  6:   6 -0.4782063
##  7:   7  2.0325554
##  8:   8  1.4157829
##  9:   9  0.9652285
## 10:  10  0.5958212

my.dt

##       x          y
##  1:   1  0.9483857
##  2:   2  1.0988303
##  3:   3  0.3966207
##  4:   4  2.4108128
##  5:   5 -0.1902492
##  6:   6 -0.4782063
##  7:   7  2.0325554
##  8:   8  1.4157829
##  9:   9  0.9652285
## 10:  10  0.5958212
```

The assignemnt of the value '1' using the new syntax, changed the only object, pointed at by both names. When using data table syntax, if we really want a copy, then we should use the function `copy`.

```
my.cp.dt <- copy(my.dt)
identical(my.cp.dt, my.dt)

## [1] TRUE
```

```
my.cp.dt[ , y := y - 1]

##       x           y
##  1:   1 -0.05161428
##  2:   2  0.09883030
##  3:   3 -0.60337927
##  4:   4  1.41081282
##  5:   5 -1.19024924
##  6:   6 -1.47820628
##  7:   7  1.03255537
##  8:   8  0.41578295
##  9:   9 -0.03477154
## 10:  10 -0.40417881

identical(my.cp.dt, my.dt)

## [1] FALSE

my.cp.dt

##       x           y
##  1:   1 -0.05161428
##  2:   2  0.09883030
##  3:   3 -0.60337927
##  4:   4  1.41081282
##  5:   5 -1.19024924
##  6:   6 -1.47820628
##  7:   7  1.03255537
##  8:   8  0.41578295
##  9:   9 -0.03477154
## 10:  10 -0.40417881

my.dt

##       x          y
##  1:   1  0.9483857
##  2:   2  1.0988303
##  3:   3  0.3966207
##  4:   4  2.4108128
##  5:   5 -0.1902492
##  6:   6 -0.4782063
##  7:   7  2.0325554
##  8:   8  1.4157829
##  9:   9  0.9652285
## 10:  10  0.5958212
```

For fast access of large data sets one can set a 'key' based on one or more columns, using function `setkey`.

```
setkey(my.dt, y)
my.dt

##       x          y
##  1:   6 -0.4782063
##  2:   5 -0.1902492
##  3:   3  0.3966207
##  4:  10  0.5958212
##  5:   1  0.9483857
##  6:   9  0.9652285
```

```
##  7:  2  1.0988303
##  8:  8  1.4157829
##  9:  7  2.0325554
## 10:  4  2.4108128

setkey(my.dt, x)
my.dt

##       x          y
##  1:   1  0.9483857
##  2:   2  1.0988303
##  3:   3  0.3966207
##  4:   4  2.4108128
##  5:   5 -0.1902492
##  6:   6 -0.4782063
##  7:   7  2.0325554
##  8:   8  1.4157829
##  9:   9  0.9652285
## 10:  10  0.5958212
```

There is also an special `print` method for datables, that is used automatically by default, that instead of printing the whole data.table, only prints the 'head' and the 'tail' of the tables, unless the table has few rows. In the examples above all rows were printed because, there were not many of them.

```
data.table(x = 1:1000, y = runif(1000))

##           x          y
##     1:    1 0.07529289
##     2:    2 0.99999111
##     3:    3 0.16987118
##     4:    4 0.80733600
##     5:    5 0.47842172
##    ---
##   996:  996 0.64971811
##   997:  997 0.23213826
##   998:  998 0.53213829
##   999:  999 0.18064385
##  1000: 1000 0.43752848
```

C.4 Using `data.frames` and `data.tables`

Adding new columns based on other columns, or other variables, uses the same syntax shown above for modifying column 'y'.

```
my.cp.dt[ , z := y + x * 2]

##       x          y         z
##  1:   1 -0.05161428  1.948386
##  2:   2  0.09883030  4.098830
##  3:   3 -0.60337927  5.396621
##  4:   4  1.41081282  9.410813
##  5:   5 -1.19024924  8.809751
##  6:   6 -1.47820628 10.521794
##  7:   7  1.03255537 15.032555
```

```
##  8:  8  0.41578295 16.415783
##  9:  9 -0.03477154 17.965228
## 10: 10 -0.40417881 19.595821
```

```
try(detach(package:data.table))
```

Making publication quality plots with R

D.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, the last three packages seem to interfere with each other, in particular GGally does not function in ggtern is loaded, so the are loaded only in the sections where they are used:

```
library(plyr)
library(grid)
library(Hmisc)

## Loading required package:  lattice
## Loading required package:  survival
## Loading required package:  splines
## Loading required package:  Formula
##
## Attaching package:  'Hmisc'
##
## The following objects are masked from 'package:plyr':
##
##    is.discrete, summarize
##
## The following objects are masked from 'package:base':
##
##    format.pval, round.POSIXt, trunc.POSIXt,
##    units

library(ggplot2)
library(scales)
# library(rgdal)
# library(ggtern)
# library(ggmap)
# library(GGally)
```

D.2 Introduction

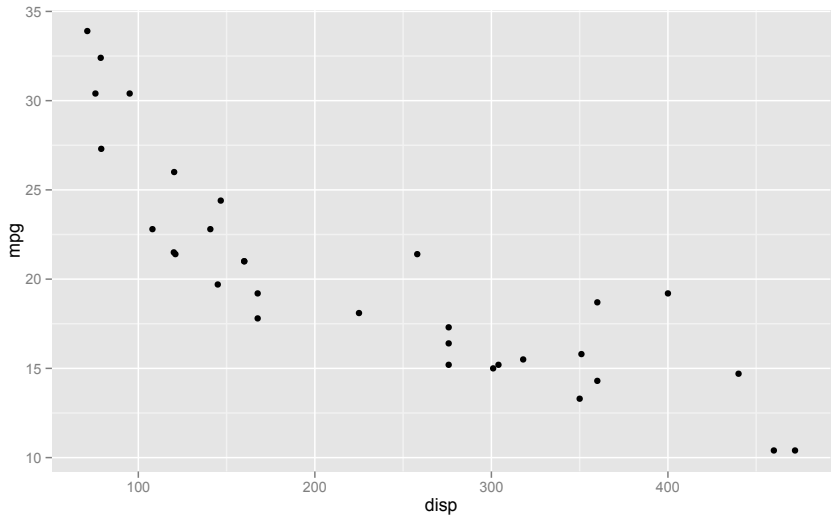
Being R extensible, in addition to the built-in plotting functions, there are several alternatives provided by packages. Of the general purpose ones, the most extensively used are `Lattice` and `ggplot2`. There are packages that add extra functionality to these packages.

In the examples in this handbook we mainly use `ggplot`, `ggmap` and `ggtern`. In this appendix we give an introduction to the 'grammar of graphics' and `ggplot2`. There is ample literature on the use of ggplot2, starting with very good reference documentation at `http://ggplot2.org/`. The book 'R Graphics Cookbook' Chang 2013 is very useful and should be always near you, when using the package, as it contains many worked out examples. There is some overlap between this appendix and the documents mentioned above. There is little well-organized literature on packages extending ggplot2, and as we make use of several of them in this handbook, we have included some examples of their use in this appendix.

D.3 Bases of plotting with `ggplot2`

The grammar of graphics is based on aesthetics (`aes`) as for example color, geometric elements `geom_...` such as lines, and points, statistics `stat_...`, scales `scale_...`, labels `labs`, and themes `theme_...`. Plots are assembled from these elements, we start with a plot with two aesthetics, and one geometry. In the examples that follow we will use the `mtcars` data set included in R.

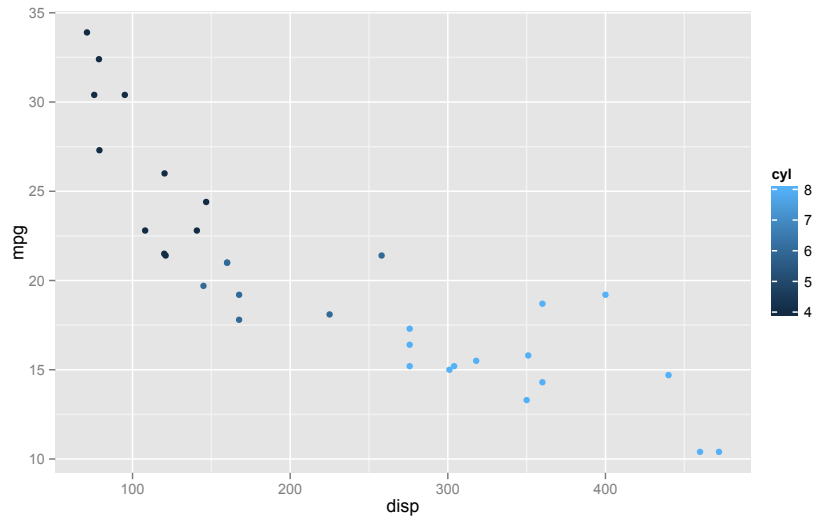
```
ggplot(mtcars, aes(x=disp, y=mpg)) +
  geom_point()
```



Aesthetics can be 'linked' to data variables, either continuous (numeric) or categorical (factor). Variable `cyl` is encoded in the `mtcars` dataframe as

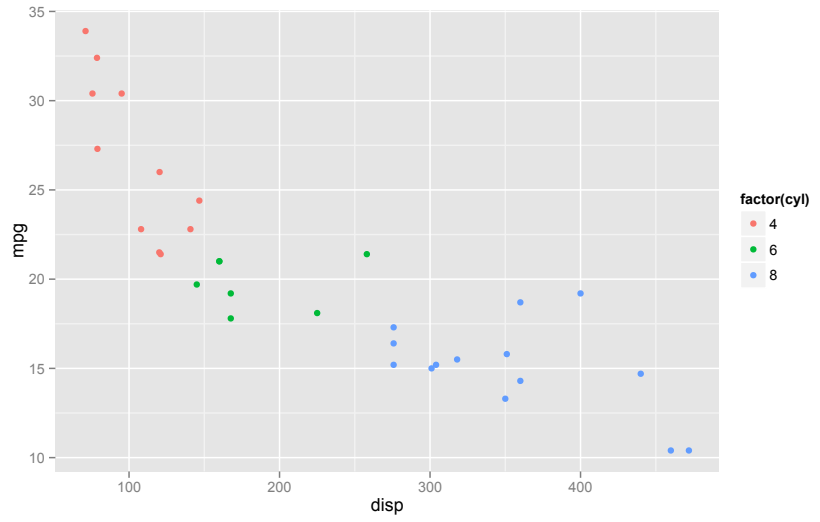
numeric values. Even though only three values are present, a continuous color scale is used.

```
ggplot(mtcars, aes(x=disp, y=mpg, colour=cyl)) +
  geom_point()
```



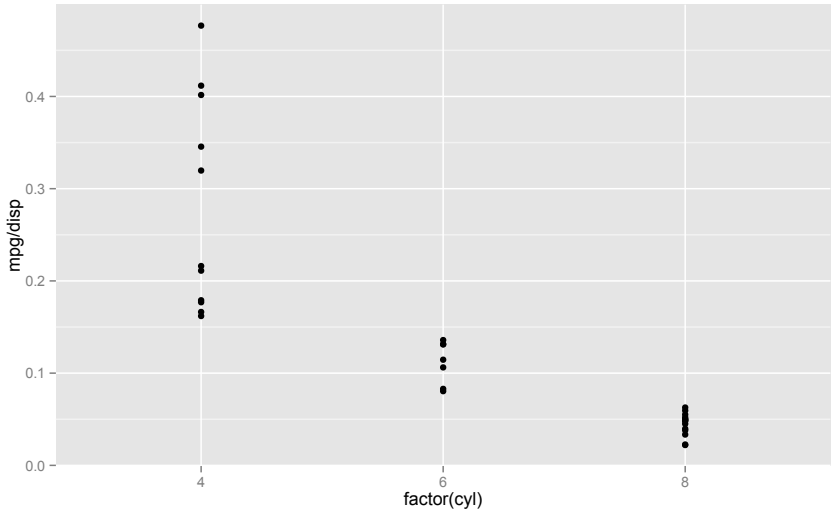
We can convert `cyl` in a factor 'on-the-fly' to force the use of a discrete color scale.

```
ggplot(mtcars, aes(x=disp, y=mpg, colour=factor(cyl))) +
  geom_point()
```



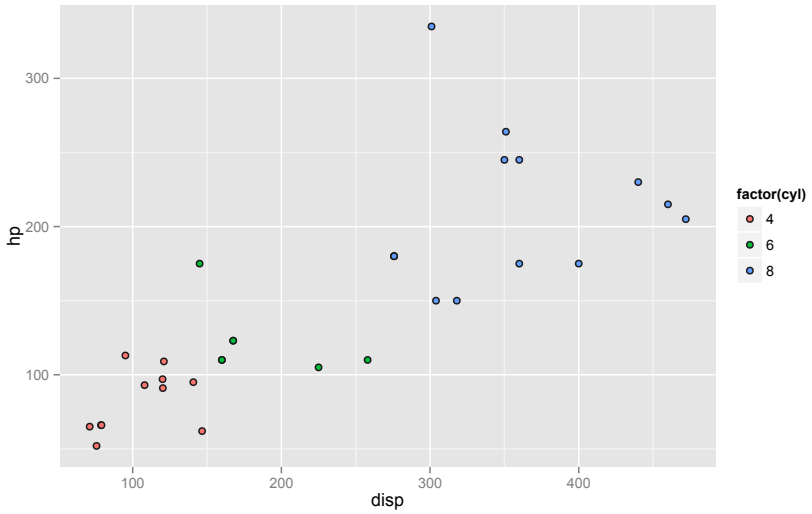
Data assigned to an aesthetic can be the 'result of a calculation'.

```
ggplot(mtcars, aes(x=factor(cyl), y=mpg / disp)) +
  geom_point()
```



Within `aes` the aesthetics are interpreted as being a function of the values in the data. If given outside `aes` they are interpreted as constants, which apply to one geom if given within the call to `geom_xxx` but outside `aes` or to the whole plot if given within `ggplot` but outside `aes`. The aesthetics and data given as `ggplot`'s arguments become the defaults for all the geoms, but geoms also take aesthetics and data as arguments, which then override the defaults.

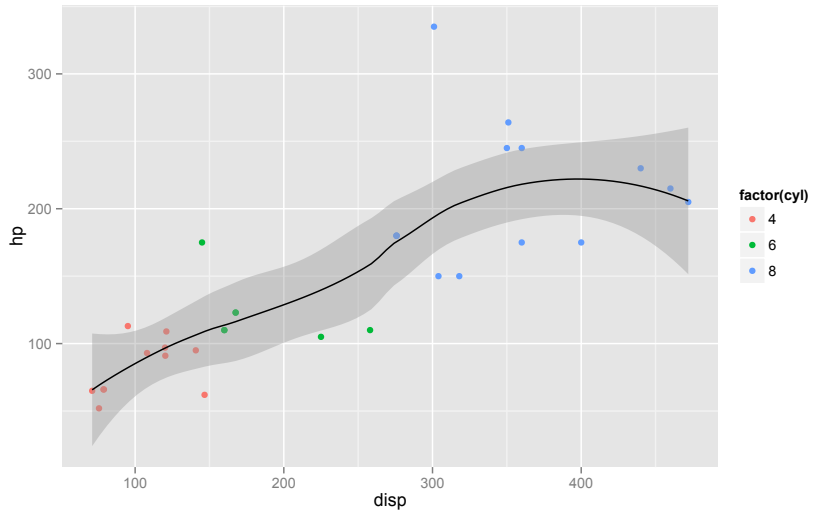
```
ggplot(mtcars, aes(x=disp, y=hp, fill=factor(cyl))) +
  geom_point(shape=21, colour="grey10")
```



In the next example we override the `color` aesthetic in `geom_smooth`[1], causing all the data to be fitted together

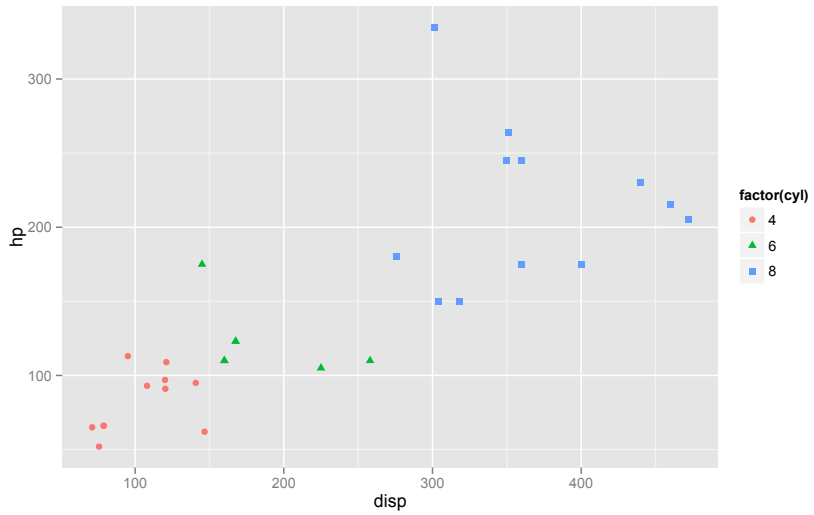
[1] Smoothing and curve fitted is discussed in more detail in section ??.

```
ggplot(mtcars, aes(x=disp, y=hp, colour=factor(cyl))) +
  geom_point() +
  geom_smooth(colour="black")

## geom_smooth:  method="auto" and size of largest group is
<1000, so using loess.  Use 'method = x' to change the smoothing
method.
```



We can assign the same variable to more than one aesthetic, and the combined key will be produced automatically.

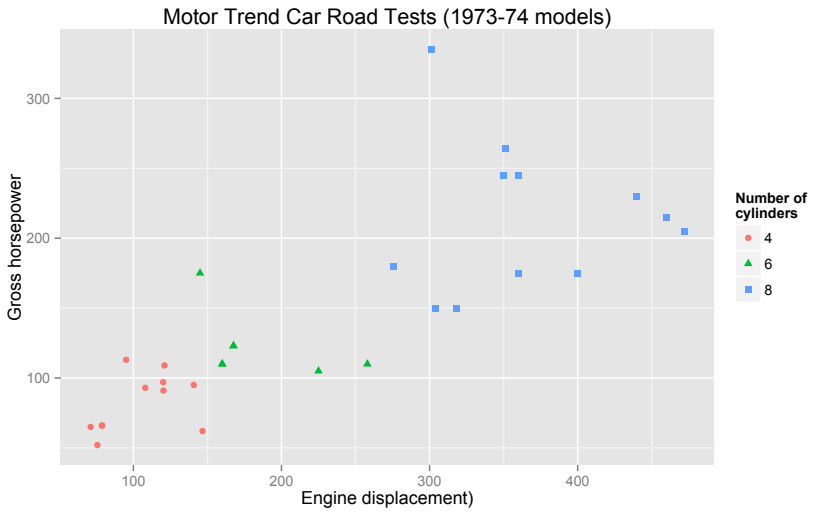
```
ggplot(mtcars, aes(x=disp, y=hp, colour=factor(cyl), shape=factor(cyl))) +
  geom_point()
```



We can change the labels for the different aesthetics, and give a title (\n means 'new line' and can be used to continue a label in the next line). In this

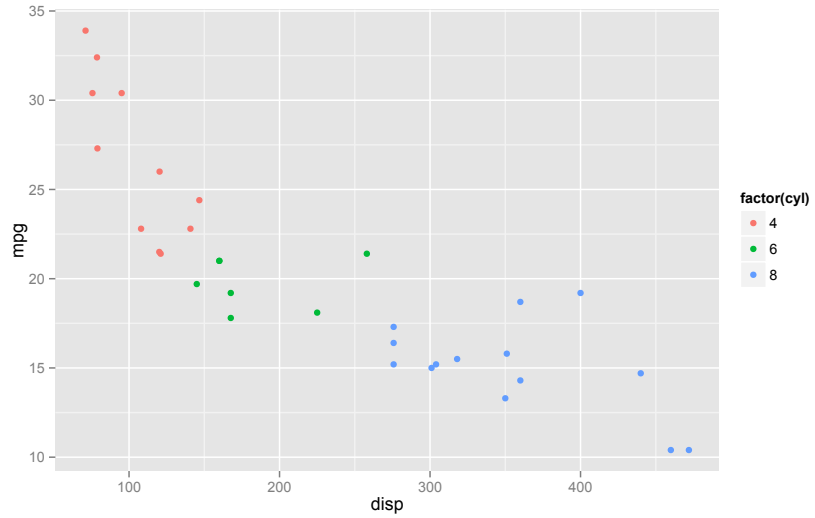
case, if two aesthetics are linked to the same variable, the labels supplied should be identical, otherwise two separate keys will be produced.

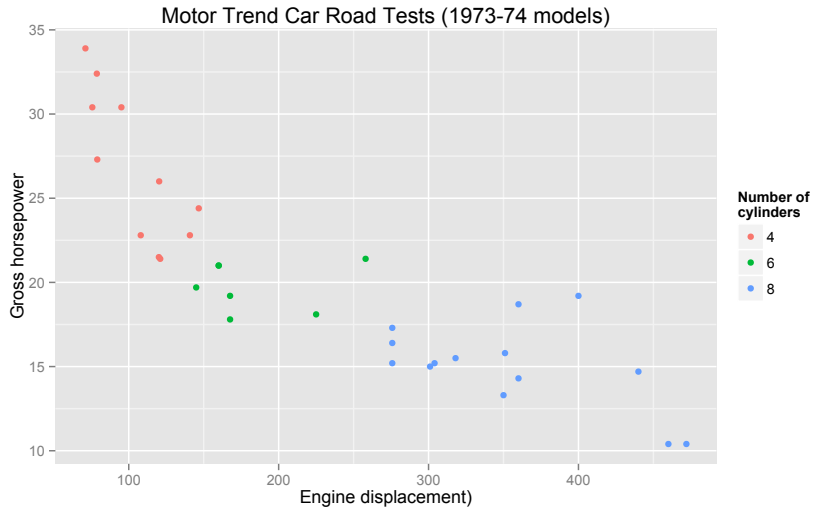
```
ggplot(mtcars, aes(x=disp, y=hp, colour=factor(cyl), shape=factor(cyl))) +
  geom_point() +
    labs(x="Engine displacement)",
       y="Gross horsepower",
       colour="Number of\ncylinders",
       shape="Number of\ncylinders",
       title="Motor Trend Car Road Tests (1973-74 models)")
```



We can assign a ggplot object or a part of it to a variable, and then assemble a new plot from the different pieces.

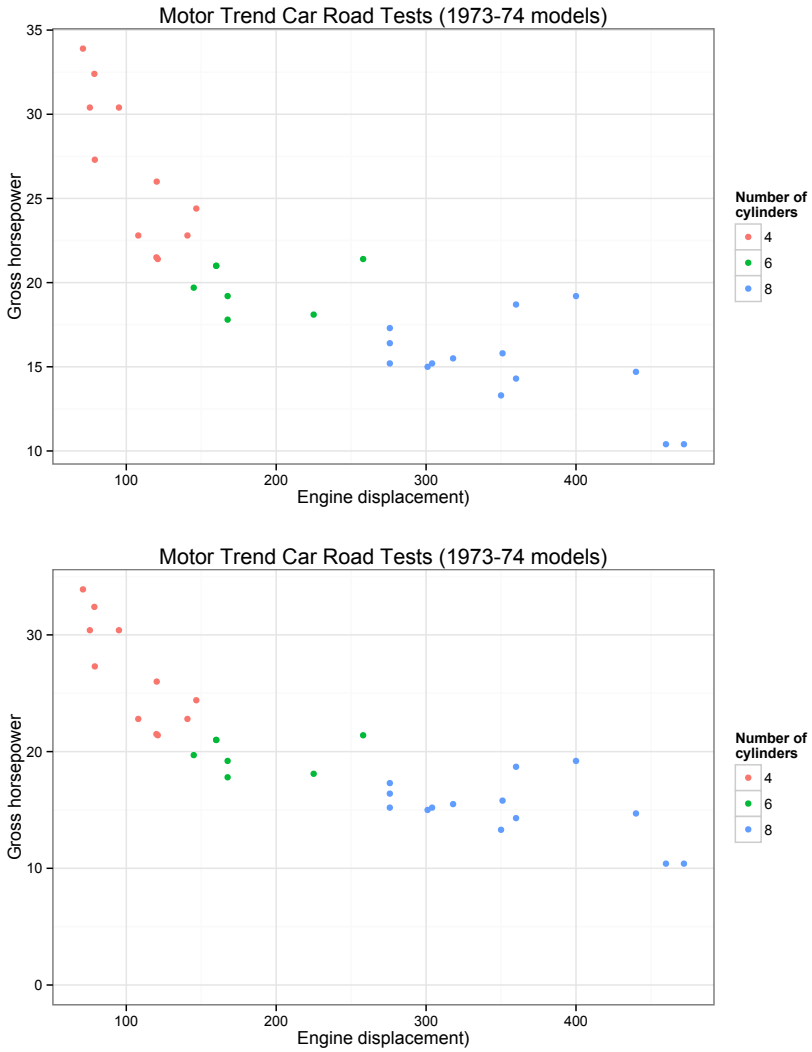
```
myplot <- ggplot(mtcars, aes(x=disp, y=mpg, colour=factor(cyl))) +
  geom_point()
mylabs <- labs(x="Engine displacement)",
       y="Gross horsepower",
       colour="Number of\ncylinders",
       shape="Number of\ncylinders",
       title="Motor Trend Car Road Tests (1973-74 models)")
myplot
myplot + mylabs
```





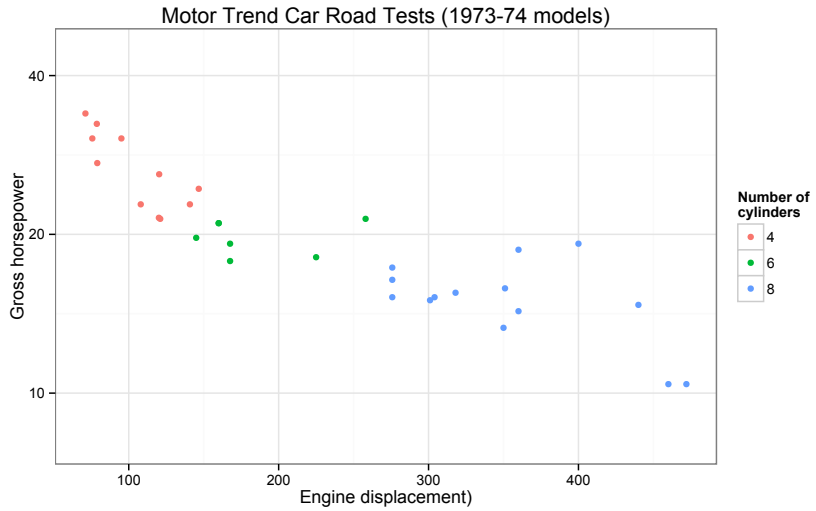
And now we can assemble them into plots.

```
myplot + mylabs + theme_bw()
myplot + mylabs + theme_bw() + ylim(0, NA)
```



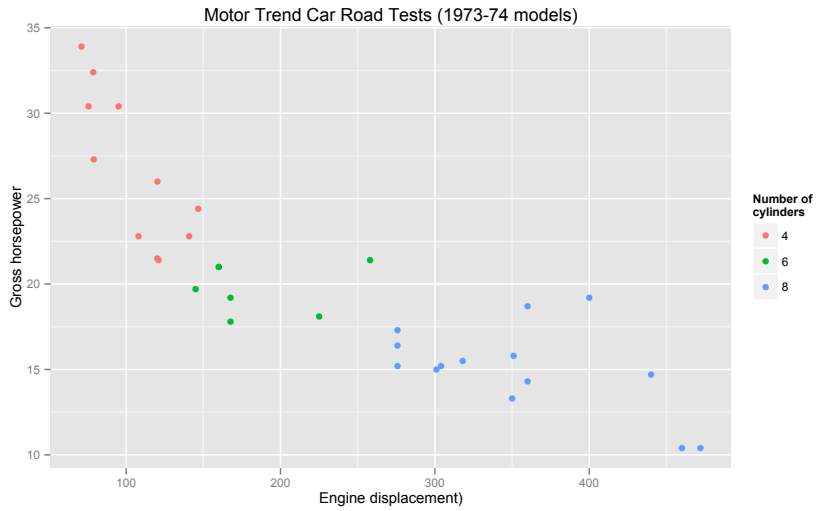
We can also save intermediate results.

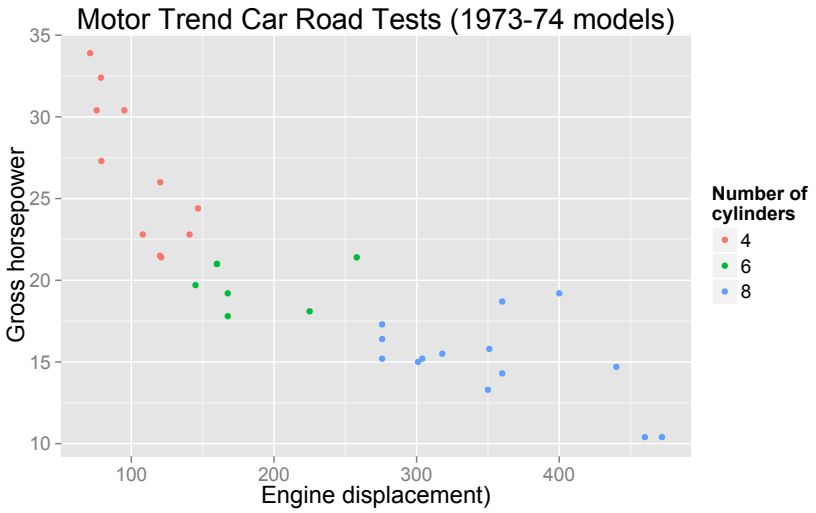
```
mylogplot <- myplot + scale_y_log10(breaks=c(10,20,40), limits=c(8,45))
mylogplot + mylabs + theme_bw()
```



The are a few predefined themes, even the default `theme_grey` can come in handy because the first parameter to themes is the point size used as reference to calculate all other font sizes. You can see in the two examples bellow, that the size of all text elements changes proportionally.

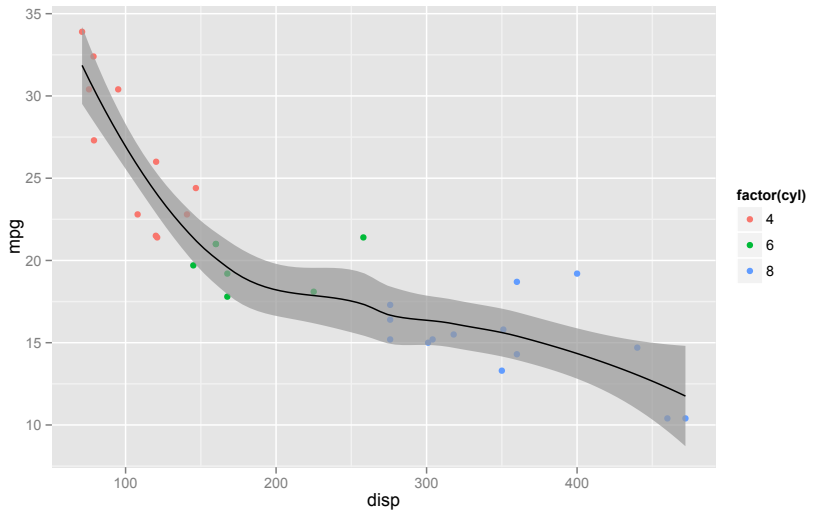
```
myplot + mylabs + theme_grey(10)
myplot + mylabs + theme_grey(16)
```





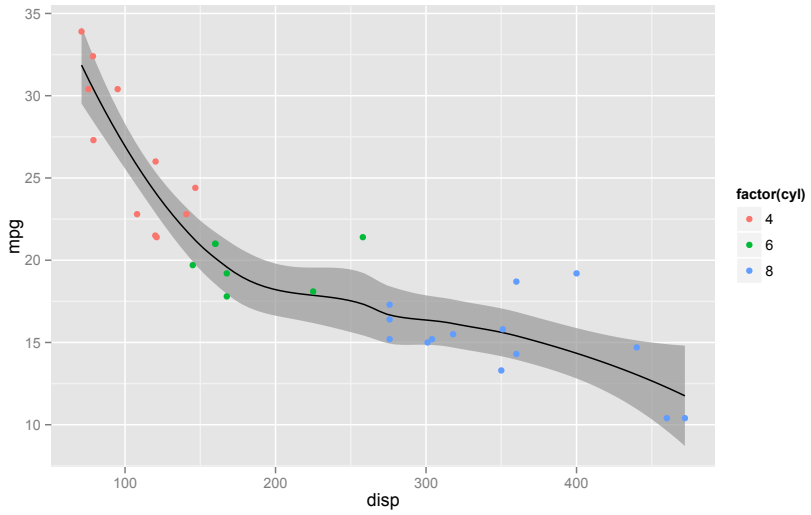
Be aware that the different geoms and elements can be added in almost any order to a ggplot object, but they will be plotted in the order that they are added. We use the `alpha` aesthetic to make the confidence band less transparent so that the example is easier to see in print.

```
ggplot(mtcars, aes(x=disp, y=mpg, colour=factor(cyl))) +
  geom_point() + geom_smooth(colour="black", alpha=0.7)

## geom_smooth:  method="auto" and size of largest group is
<1000, so using loess.  Use 'method = x' to change the smoothing
method.
```



The plot looks different if the order of the geoms is swapped. The data points overlapping the confidence band are more clearly visible in this second example because they are above the shaded area instead of bellow it.

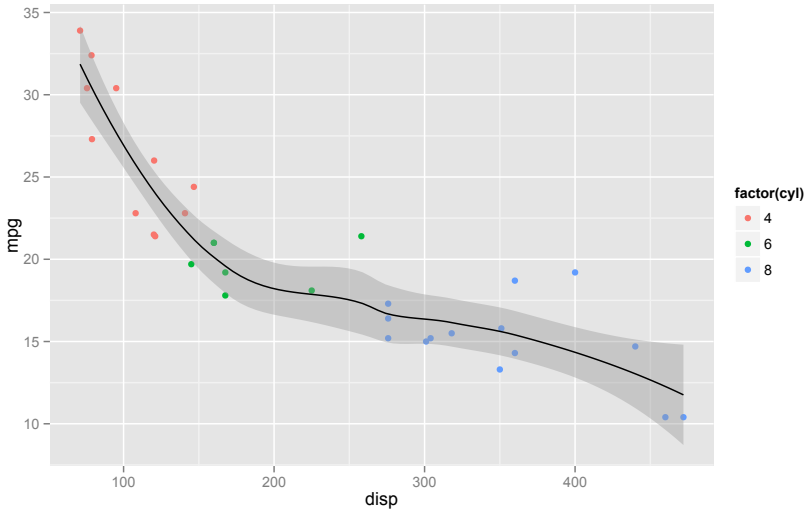
```
ggplot(mtcars, aes(x=disp, y=mpg, colour=factor(cyl))) +
   geom_smooth(colour="black", alpha=0.7) + geom_point()

## geom_smooth:  method="auto" and size of largest group is
<1000, so using loess.  Use 'method = x' to change the smoothing
method.
```



D.4 Adding fitted curves, including splines

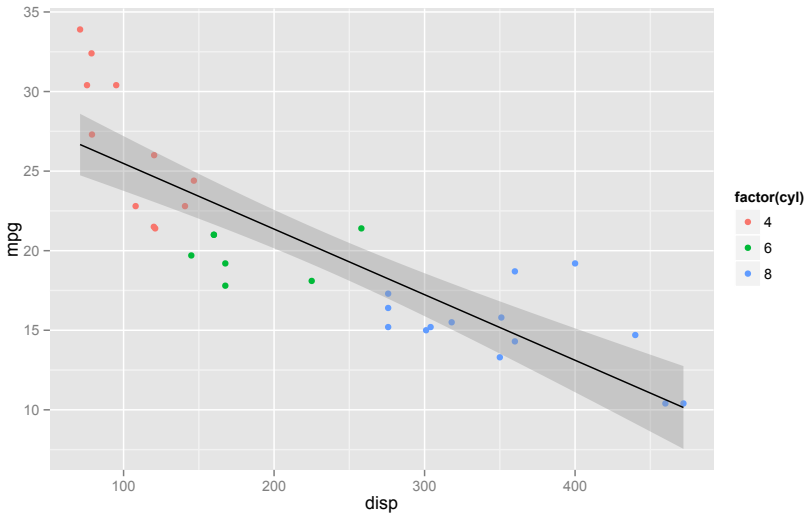
We will now show an example of use of `stat_smooth` using the default spline smoothing.

```
myplot + stat_smooth(colour="black")

## geom_smooth:  method="auto" and size of largest group is
<1000, so using loess.  Use 'method = x' to change the smoothing
method.
```



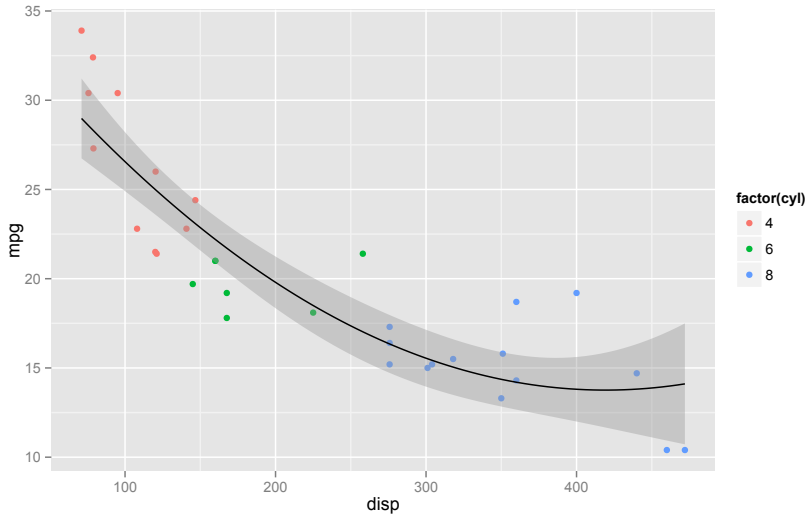
Instead of using the default spline, we can use a linear model fit. In this example we use a linear model, fitted by lm, as smoother:

```
myplot + stat_smooth(method="lm", colour="black")
```



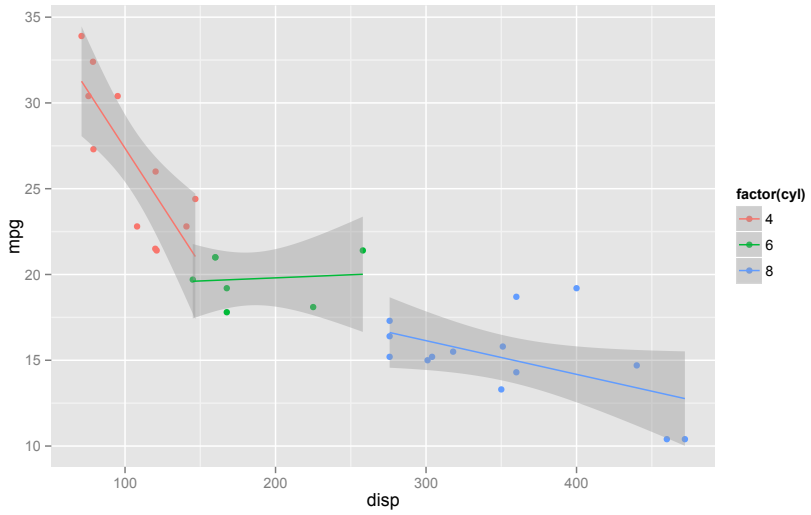
Instead of using the default linear regression as smoother, we can use a linear model fit. In this example we use a polynomial of order 2 fitted by lm.

```
myplot + stat_smooth(method="lm", formula=y~poly(x,2), colour="black")
```



If we do not use `colour="black"` then the colour aesthetics supplied to `ggplot` is used, and splits the data into three groups to which the model is fitted separately.

```
myplot + stat_smooth(method="lm")
```



It is possible to use other types of models, including GAM and GLM, as smoothers, but we will not give examples of the use these more advanced models.

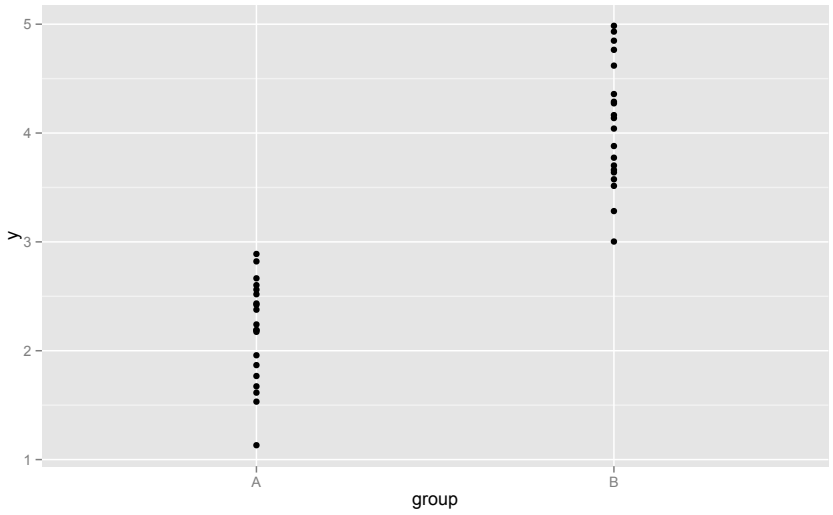
D.5 Adding statistical "summaries"

It is also possible to summarize data on-the-fly when plotting, but before showing this we will generate some normally distributed artificial data:

```
fake.data <- data.frame(
  y = c(rnorm(20, mean=2, sd=0.5), rnorm(20, mean=4, sd=0.7)),
  group = factor(c(rep("A", 20), rep("B", 20)))
  )
```

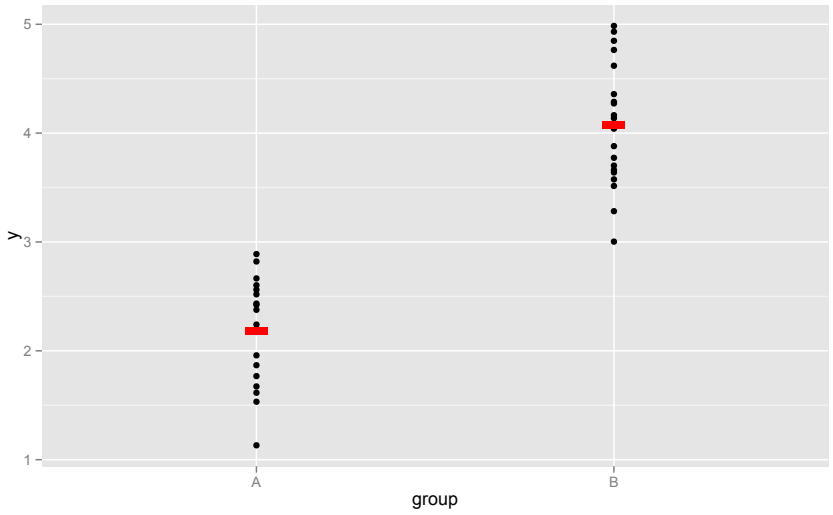
Now we use these data to plot means and confidence intervals by group:

```
fig2 <- ggplot(data=fake.data, aes(y=y, x=group)) + geom_point()
fig2
```

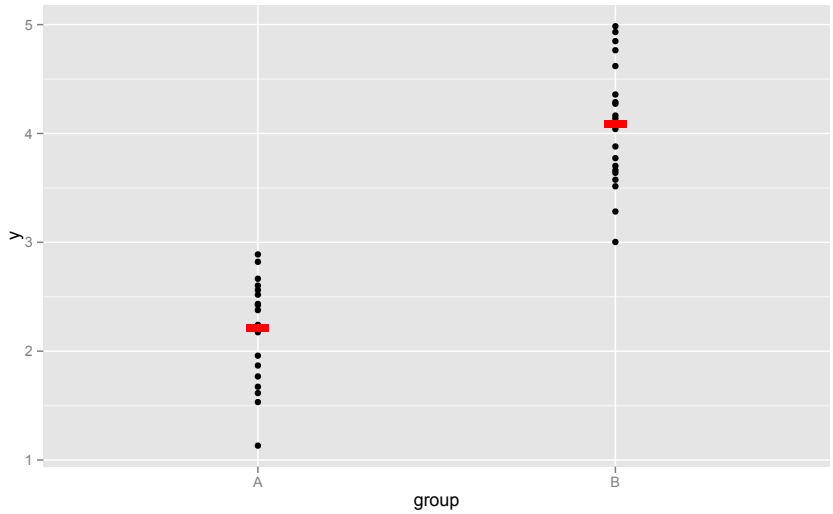


We have saved the base figure in `fig2`, so now we can play with different summaries. We first add just the mean. In this case we need to add as argument to `stat_summary` the geom to use, as the default one expects data for plotting error bars, in later examples, this is not needed.

```
fig2 + stat_summary(fun.y = "mean", geom="point",
                    colour="red", shape="-", size=20)
```

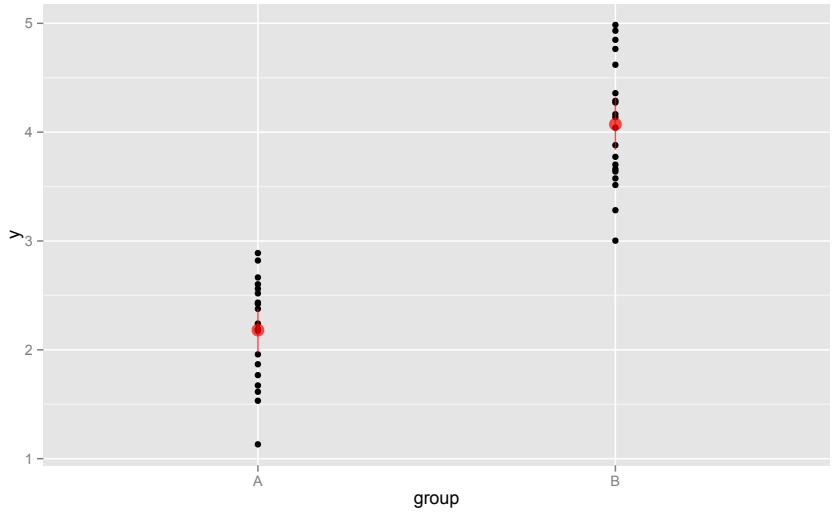


```
fig2 + stat_summary(fun.y = "median", geom="point",
                    colour="red", shape="-", size=20)
```



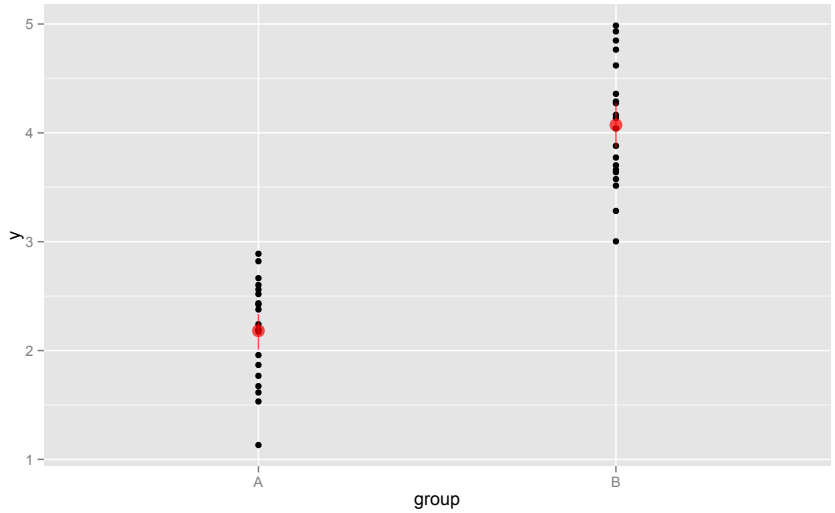
We can add the means and $p = 0.95$ confidence intervals not assuming normality (using the actual distribution of the data by bootstrapping):

```
fig2 + stat_summary(fun.data = "mean_cl_boot",
                    colour="red", size=1, alpha=0.7)
```



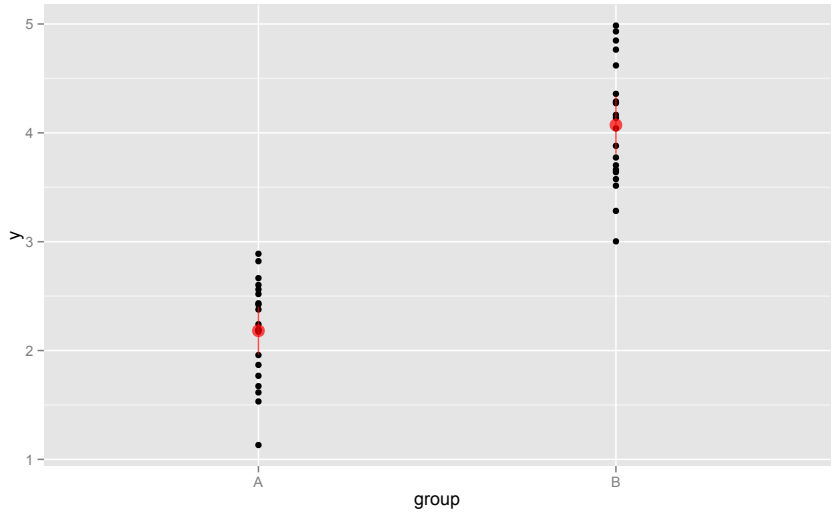
We can instead add the means and $p = 0.90$ confidence intervals, by supplying a value to parameter `conf.int`:

```
fig2 + stat_summary(fun.data = "mean_cl_boot", conf.int=0.90,
                    colour="red", size=1, alpha=0.7)
```



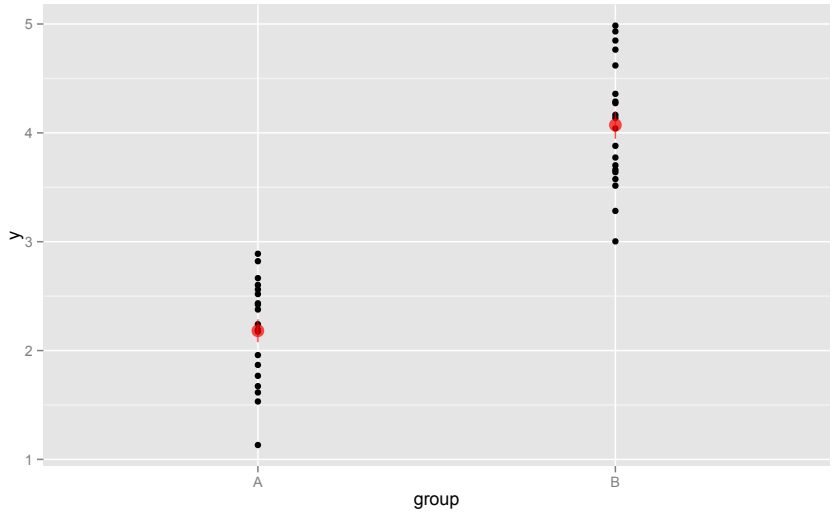
We can add the mean and $p = 0.95$ confidence intervals assuming normality (using the t distribution):

```
fig2 + stat_summary(fun.data = "mean_cl_normal",
                    colour="red", size=1, alpha=0.7)
```



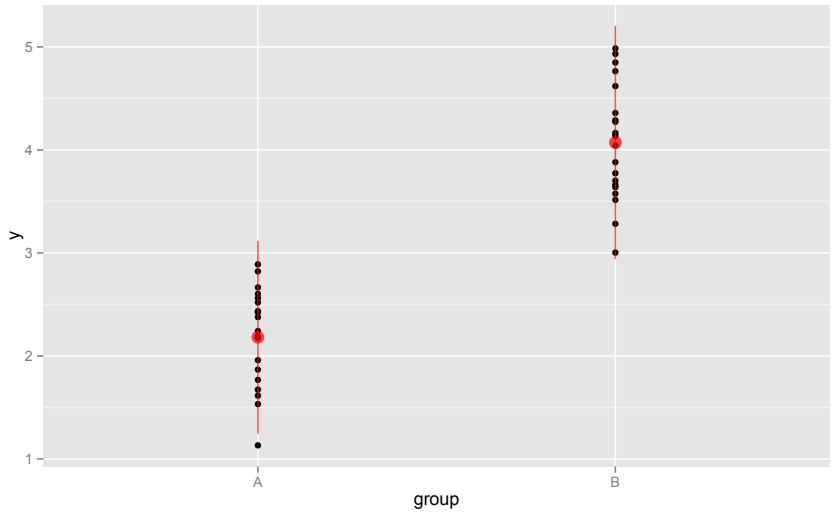
In this case the multiplier `mult` is by default is calculated from the t distribution according to degrees of freedom, but if we force the multiplier to 1, then we get error bars corresponding to ±s.e. (standard errors).

```
fig2 + stat_summary(fun.data = "mean_cl_normal", mult=1,
                    colour="red", size=1, alpha=0.7)
```



Finally we can plot error bars showing ±s.d. (standard deviation). The default value for `mult` is 2, giving error bars ±2 s.d., we use 1 as multiplier instead.

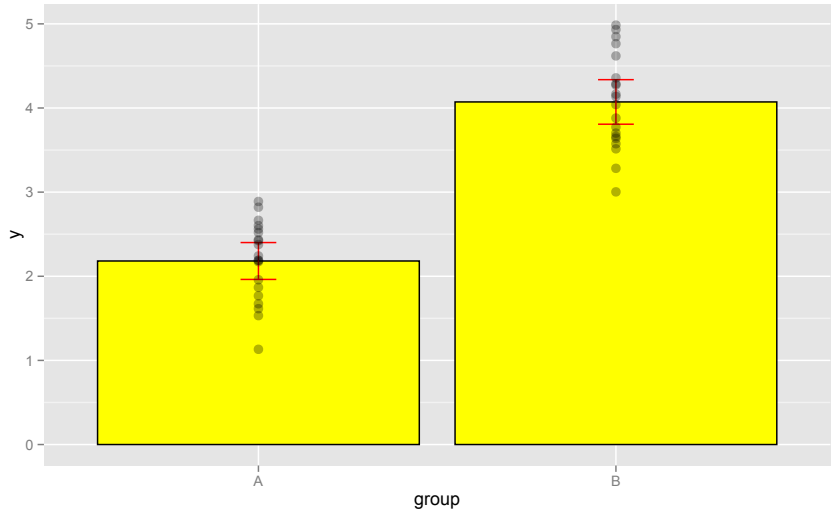
```
fig2 + stat_summary(fun.data = "mean_sdl",
                    colour="red", size=1, alpha=0.7)
```



We do not show it here, but instead of using these functions (from package `Hmisc`) it is possible to define one's own functions.

Finally we plot the means in a bar plot, with the observations superimposed and $p = 0.95$ C.I. (the order in which the geoms are added is important: by having `geom_point` last it is plotted on top of the bars. In this case we set fill, colour and alpha (transparency) to constants, but in more complex data sets they can be assigned to factors in the data set.

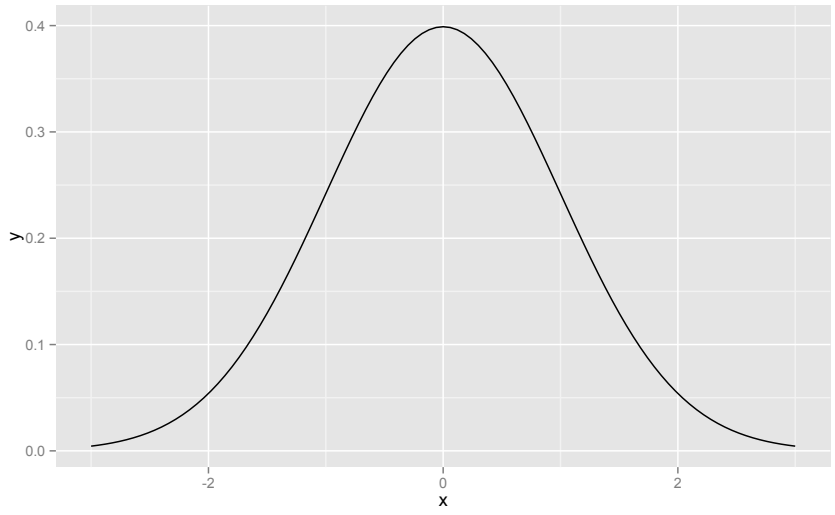
```
ggplot(data=fake.data, aes(y=y, x=group)) +
  stat_summary(fun.y = "mean", geom = "bar",
               fill="yellow", colour="black") +
  stat_summary(fun.data = "mean_cl_normal",
               geom = "errorbar",
               width=0.1, size=1, colour="red") +
  geom_point(size=3, alpha=0.3)
```



D.6 Plotting functions

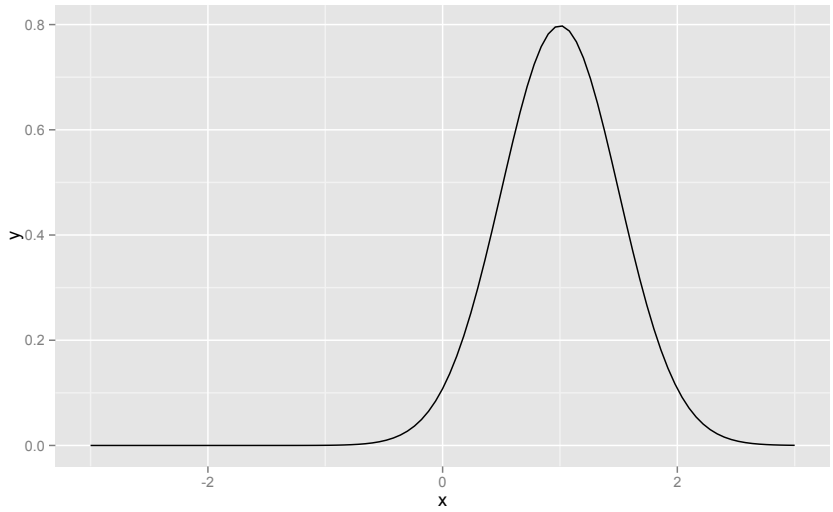
We can also directly plot functions, without need to generate data beforehand:

```
ggplot(data.frame(x=-3:3), aes(x=x)) +
  stat_function(fun=dnorm)
```



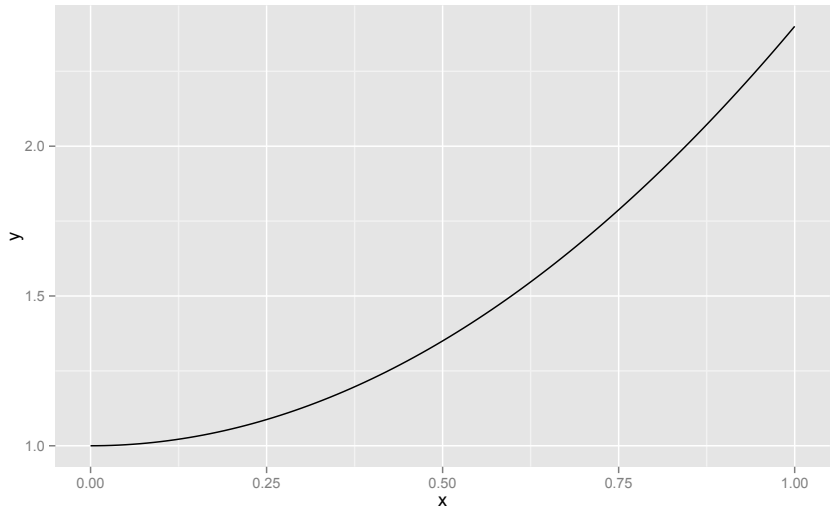
We can even pass additional arguments to a function:

```
ggplot(data.frame(x=-3:3), aes(x=x)) +
  stat_function(fun = dnorm, args = list(mean = 1, sd = .5))
```



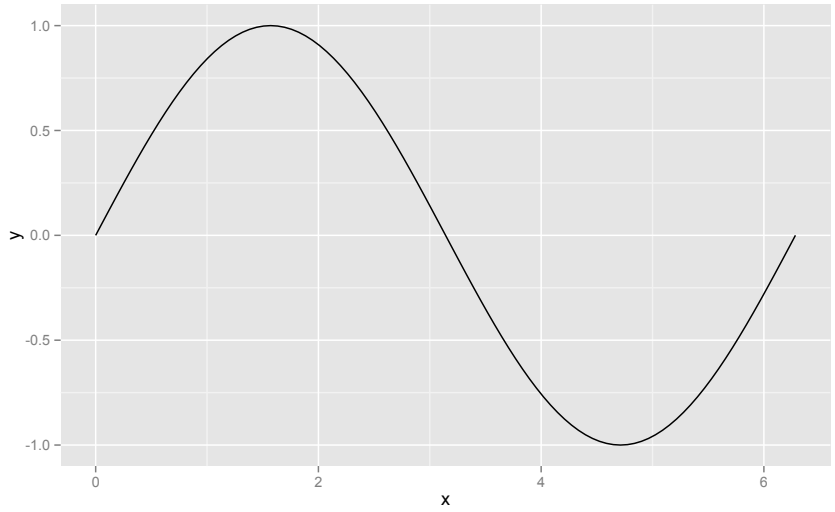
Of course, user-defined functions (not shown), and anonymous functions can also be used:

```
ggplot(data.frame(x=0:1), aes(x=x)) +
  stat_function(fun = function(x, a, b){a + b * x^2},
                args = list(a = 1, b = 1.4))
```



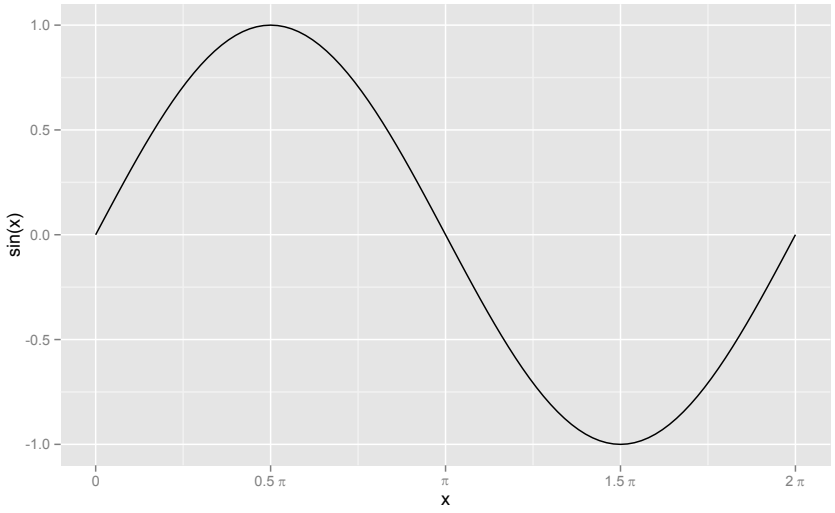
Here is another example of a predefined function, but in this case the default scale is not the best:

```
ggplot(data.frame(x=c(0, 2 * pi)), aes(x=x)) +
  stat_function(fun=sin)
```



In this case we need to change the x-axis scale to better suit the sin function and the use of radians as angular units[2].

```
ggplot(data.frame(x=c(0, 2 * pi)), aes(x=x)) +
  stat_function(fun=sin) +
  scale_x_continuous(
    breaks=c(0, 0.5, 1, 1.5, 2) * pi,
    labels=c("0", expression(0.5~pi), expression(pi),
             expression(1.5~pi), expression(2~pi))) +
  labs(y="sin(x)")
```

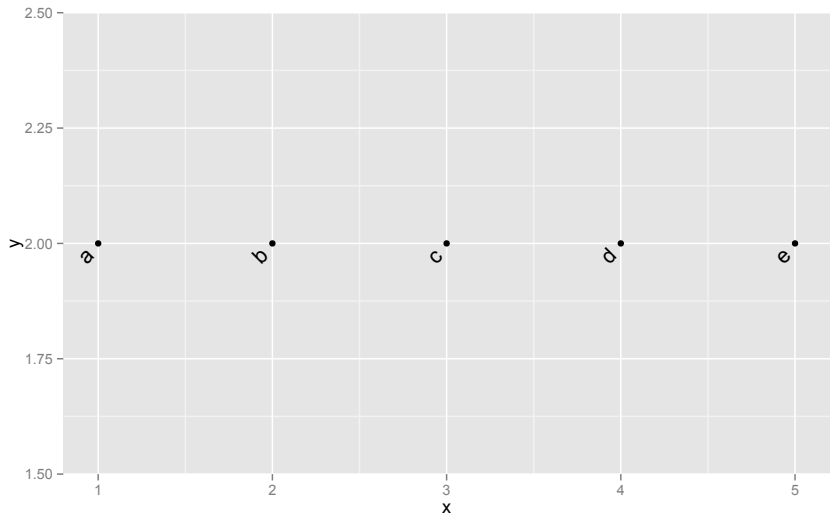


[2]The use of `expression` is explained in detail in section **??**, an the use of `scales` in section **??**.

D.7 Plotting text

One can use `geom_text` to add text labels to observations. The aesthetic `label` gives text and the usual aesthetics `x` and `y` the location of the labels. As one would expect the `colour` aesthetic can be also used for text. In addition `angle` and `vjust` and `hjust` can be used to rotate the label, and adjust its position. The default value of zero for both `hjust` and `vjust` centres the label. The centre of the text is at the supplied `x` and `y` coordinates. 'Vertical' and 'horizontal' for justification refer to the text, not the plot. This is important when `angle` is different from zero. Negative justification values, shift the label left or down, and positive values right or up. A value of 1 or -1 sets the text so that its edge is at the supplied coordinate. Values outside the range $-1 \ldots 1$ sift the text even further away.

```
my.data <-
  data.frame(x=1:5, y=rep(2, 5), label=paste(letters[1:5], " "))
ggplot(my.data, aes(x,y,label=label)) +
  geom_text(angle=45, hjust=1) + geom_point()
```



In this example we use `paste` (which uses recycling here) to add a space at the end of each label. Justification values outside the range $-1 \ldots 1$ are allowed, but are relative to the with of the label. As the default font used in this case has variable with characters, the justification would be inconsistent (e.g. try the code above but using `hjust` set to 3 instead of to 1 without pasting a space character to the labels.)

D.8 Scales

Scales map data onto aesthetics. There are different types of scales depending on the characteristics of the data being mapped: scales can be continuous or discrete. And of course, there are scales for different attributes of the plotted object, such as `colour`, `size`, position (`x, y, z`), `alpha` or transparency, `angle`, justification, etc. This means that many properties of, for example,

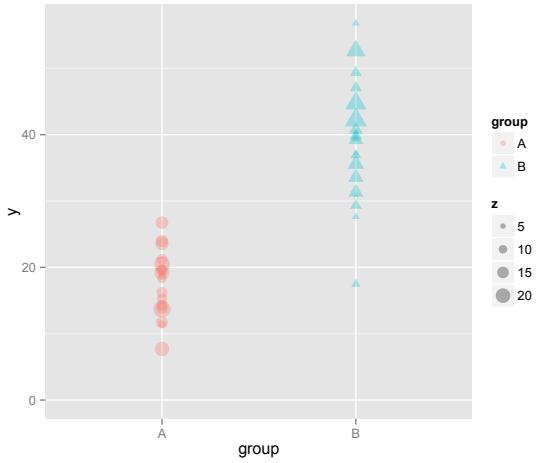
the symbols used in a plot can be either set by a constant, or mapped to data. The most elemental mapping is `identity`, which means that the data is taken at its face value. In a numerical scale, say `scale_x_continuous`, this means that for example a '5' in the data is plotted at a position in the plot corresponding to the value '5' along the x-axis. A simple mapping could be a log10 transformation, that we can easily achieve with the pre-defined `scale_x_log10` in which case the position on the x-axis will be based on the logarithm of the original data. A continuous data variable can, if we think it useful for describing our data, be mapped to continuous scale either using an identity mapping or transformation, which for example could be useful if we want to map the value of a variable to the area of the symbol rather than its diameter.

Discrete scales work in a similar way. We can use `scale_colour_identity` and have in our data a variable with values that are valid colour names like "red" or "blue". However we can also assign the `colour` aesthetic to a factor with levels like "control", and "treatment", an these levels will be mapped to colours from the default palette, unless we chose a different one, or even use `scale_colour_manual` to assign whatever colour we want to each level to be mapped. The same is true for other discrete scales like symbol `shape` and `linetype`. Be aware that for example for colour, and 'numbers' there are both discrete and continuous scales available.

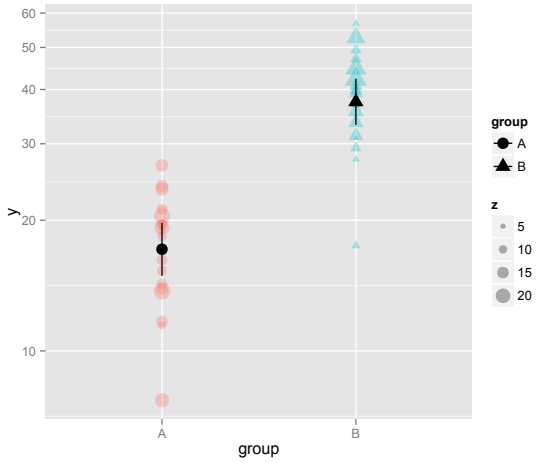
Advanced scale manipulation requires the package `scales` to be loaded. Some simple examples follow.

```
fake2.data <- data.frame(
  y = c(rnorm(20, mean=20, sd=5), rnorm(20, mean=40, sd=10)),
  group = factor(c(rep("A", 20), rep("B", 20))),
  z = rnorm(40, mean=12, sd=6)
  )
```

```
fig2 <-
  ggplot(data=fake2.data,
         aes(y=y, x=group, shape=group, colour=group, size=z)) +
  geom_point(alpha=0.3) + ylim(0, NA)
fig2
```




```
fig2 +
  scale_y_log10(breaks=c(10,20,30,40,50,60)) +
       stat_summary(fun.data = "mean_cl_normal",
                    colour="black", size=1, alpha=1)

## Scale for 'y' is already present.  Adding another scale for
'y', which will replace the existing scale.
```

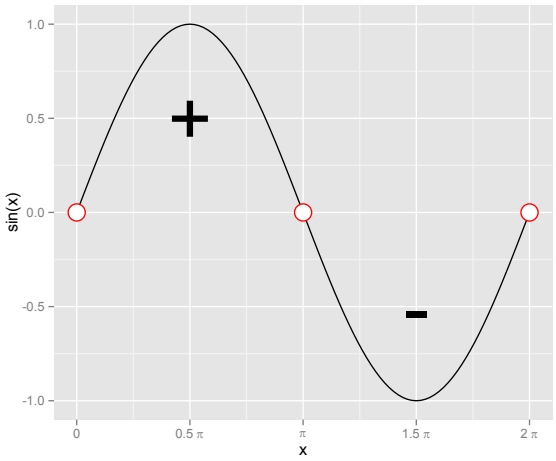


D.9 Adding annotations

Annotations use the data coordinates of the plot, but do not 'inherit' data or aesthetics from the ggplot.

```
ggplot(data.frame(x=c(0, 2 * pi)), aes(x=x)) +
  stat_function(fun=sin) +
  scale_x_continuous(
    breaks=c(0, 0.5, 1, 1.5, 2) * pi,
    labels=c("0", expression(0.5~pi), expression(pi),
             expression(1.5~pi), expression(2~pi))) +
```

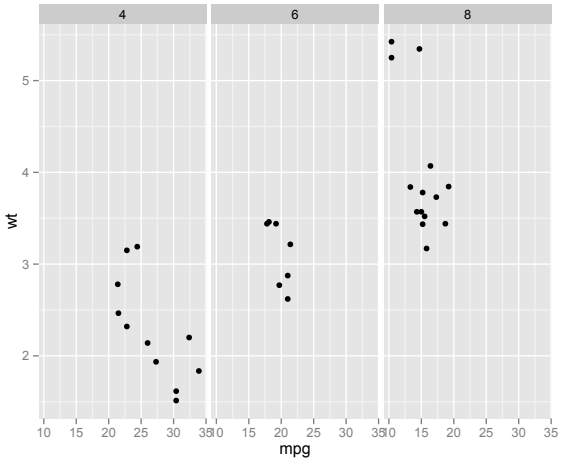
```
  labs(y="sin(x)") +
  annotate(geom="text",
           label=c("+", "-"),
           x=c(0.5, 1.5) * pi, y=c(0.5, -0.5),
           size=20) +
  annotate(geom="point",
           colour="red",
           shape=21,
           fill="white",
           x=c(0, 1, 2) * pi, y=0,
           size=6)
```



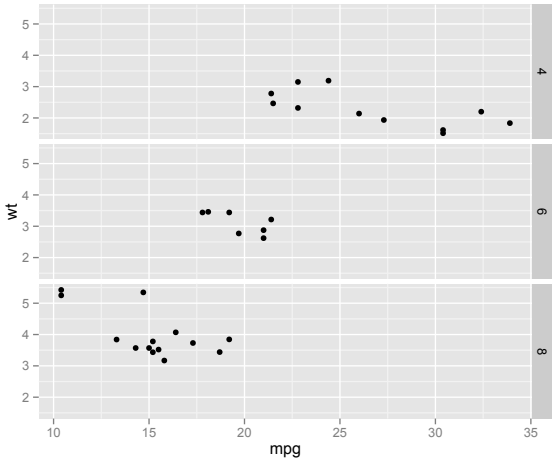
D.10 Using facets

Sets of coordinated plots are a very useful tool for visualizing data. These became popular through the `trellis` graphs in S, and the `lattice` package in R. The basic idea is to have row and/or columns of plots with common scales, all plots showing values for the same response variable. This is useful when there are multiple classification factors in a data set. Similarly looking plots but with free scales or with the same scale but a 'floating' intercept are sometimes also useful. In `ggplot2` there are two possible types of facets: facets organized in a grid, and facets on along a single 'axis' but wrapped into several rows. In the examples below we use `geom_point` but faceting can be used with any geom, and even with maps and ternary plots.

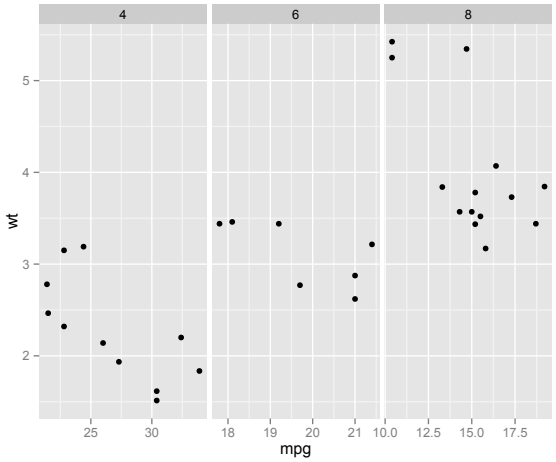
```
p <- ggplot(mtcars, aes(mpg, wt)) + geom_point()
# With one variable
p + facet_grid(. ~ cyl)
```



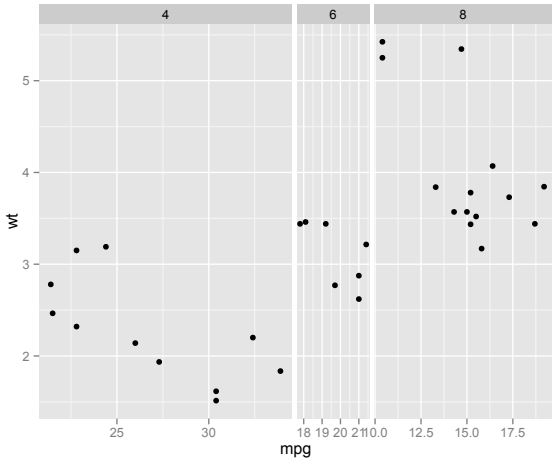

```
p + facet_grid(cyl ~ .)
```



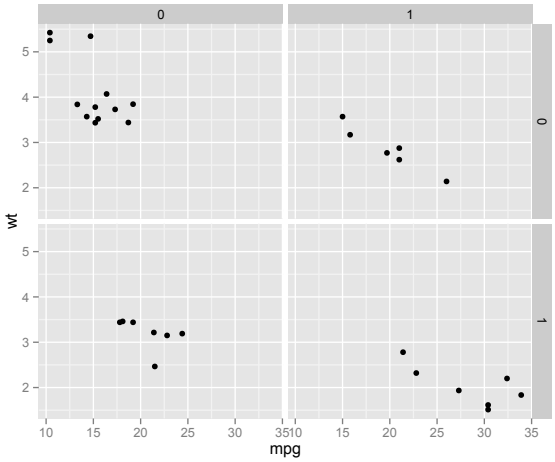

```
p + facet_grid(. ~ cyl, scales = "free")
```



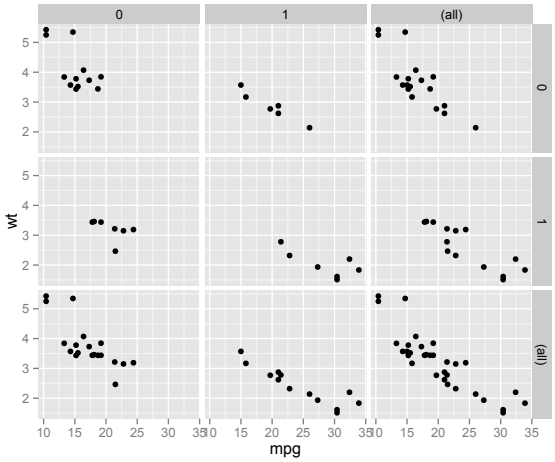

```
p + facet_grid(. ~ cyl, scales = "free", space = "free")
```



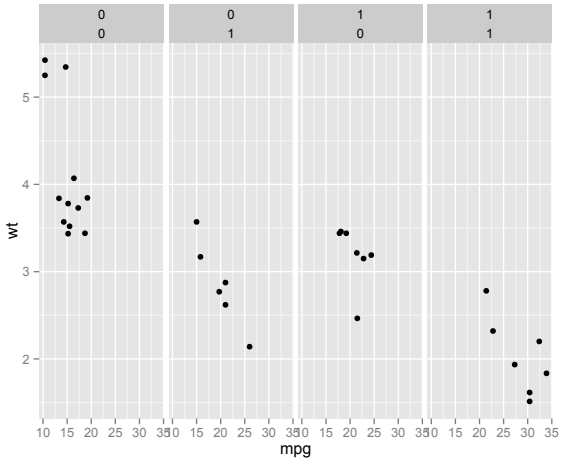

```
p + facet_grid(vs ~ am)
```



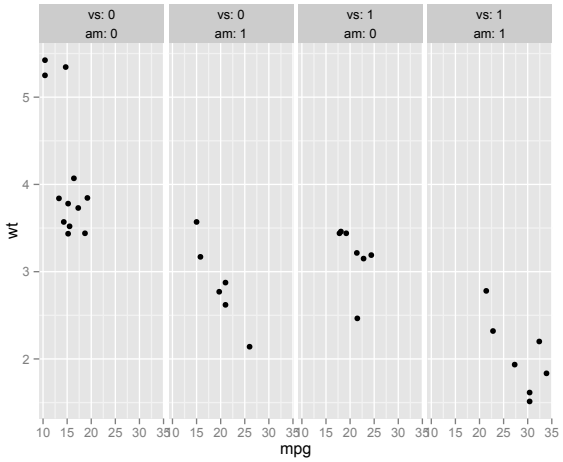

```
p + facet_grid(vs ~ am, margins=TRUE)
```



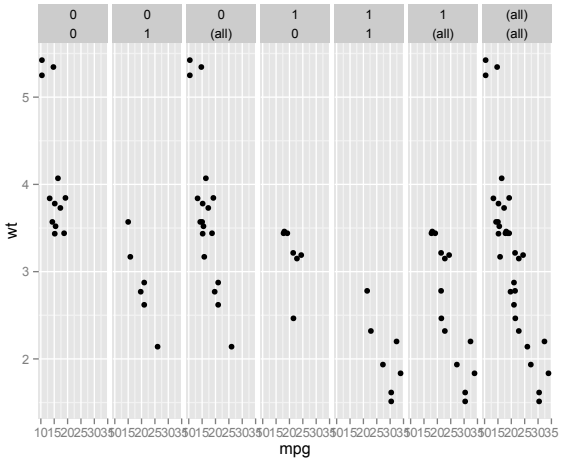

```
p + facet_grid(. ~ vs + am)
```



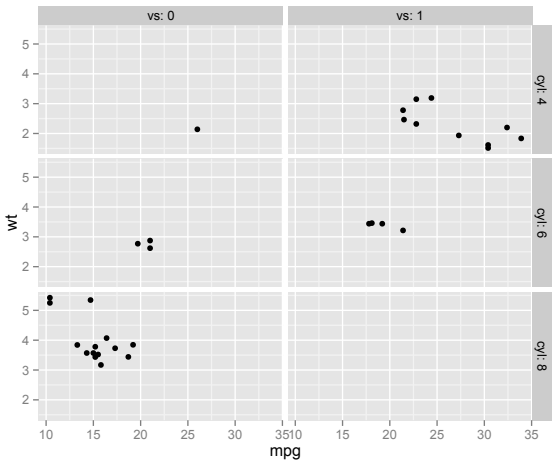

```
p + facet_grid(. ~ vs + am, labeller = label_both)
```



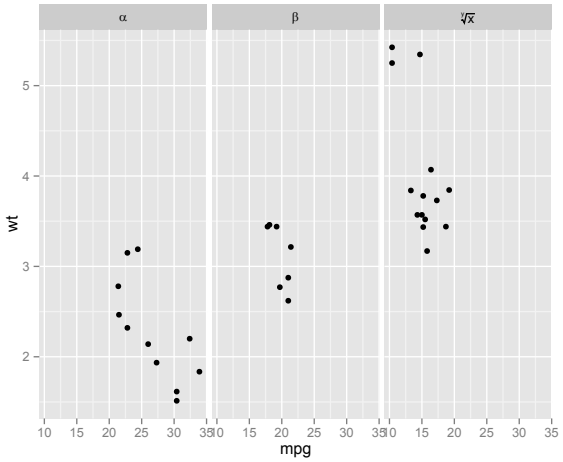

```
p + facet_grid(. ~ vs + am, margins=TRUE)
```



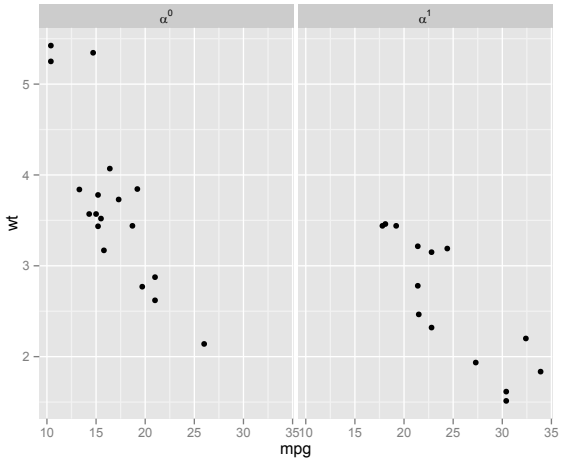

```
p + facet_grid(cyl ~ vs, labeller = label_both)
```



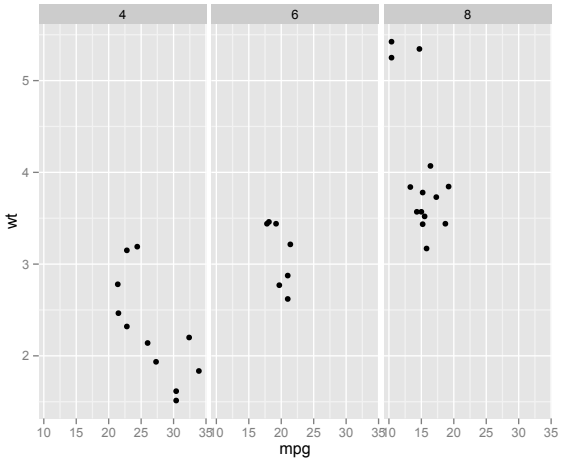

```
mtcars$cyl12 <- factor(mtcars$cyl, labels = c("alpha", "beta", "sqrt(x, y)"))
p1 <- ggplot(mtcars, aes(mpg, wt)) + geom_point()
p1 + facet_grid(. ~ cyl12, labeller = label_parsed)
```



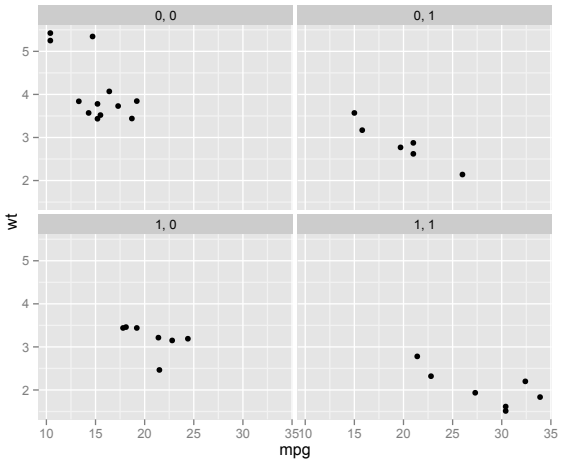

```
p + facet_grid(. ~ vs, labeller = label_bquote(alpha ^ .(x)))
```




```
p + facet_wrap(~ cyl)
```




```
p + facet_wrap(~ vs + am, ncol=2)
```



At the current time, `facet_wrap` does not accept labellers, so neither expressions nor including the name of the variable in the labels can be done automatically.

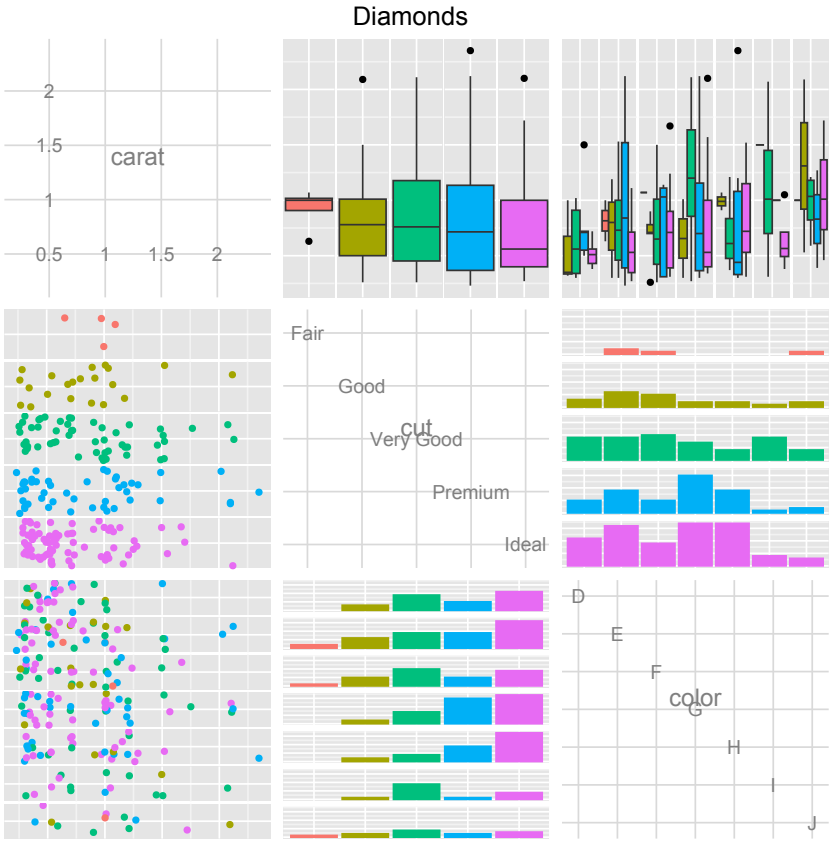
D.11 Plot matrices

In this type of plot a set of several variables are plotted against each other, forming all possible pairs. There is a function `plotmatrix` in ggplot2, but it is deprecated. Function `ggpairs` from package GGally provides this type of plots as an extension to ggplot.

```
library(GGally)
```

```
# Use sample of the diamonds data
data(diamonds, package="ggplot2")
```

```
diamonds.samp <- diamonds[sample(1:dim(diamonds)[1],200),]
# Custom Example
pm <- ggpairs(
diamonds.samp[,1:3],
upper = list(continuous = "density", combo = "box"),
lower = list(continuous = "points", combo = "dot"),
color = "cut",
title = "Diamonds"
)
pm
```



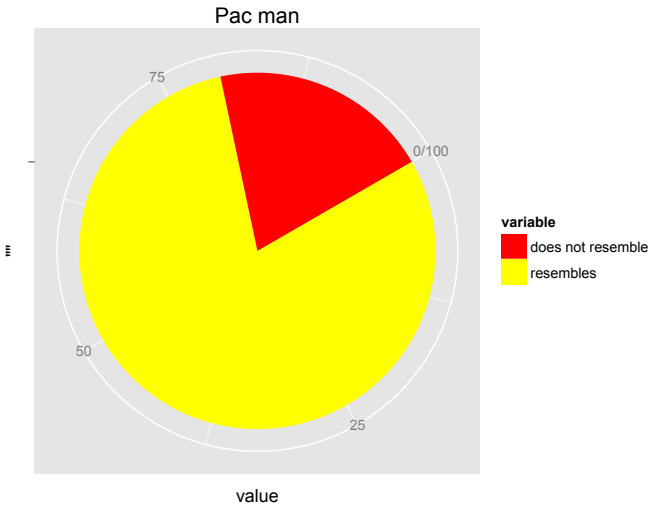

```
try(detach(package:GGally))
```

D.12 Circular plots

A funny example stolen from the ggplot2 website at `http://docs.ggplot2.org/current/coord_polar.html`.

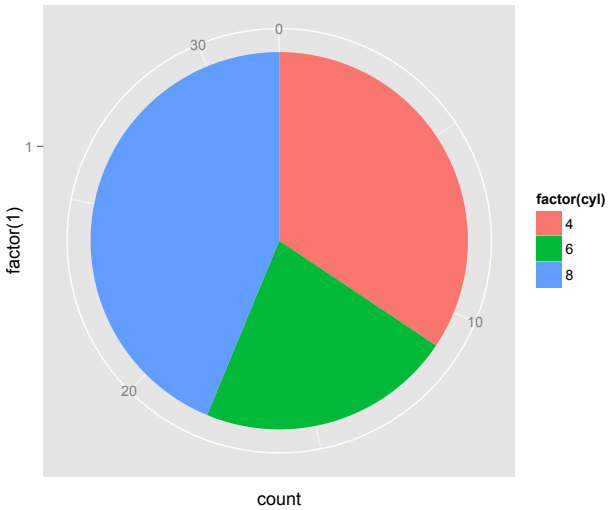
```
# Hadley's favourite pie chart
df <- data.frame(
  variable = c("resembles", "does not resemble"),
  value = c(80, 20)
)
```

```
ggplot(df, aes(x = "", y = value, fill = variable)) +
  geom_bar(width = 1, stat = "identity") +
  scale_fill_manual(values = c("red", "yellow")) +
  coord_polar("y", start = pi / 3) +
  labs(title = "Pac man")
```



Something just a bit more useful, also stolen from the same page:

```
# A pie chart = stacked bar chart + polar coordinates
pie <- ggplot(mtcars, aes(x = factor(1), fill = factor(cyl))) +
 geom_bar(width = 1)
pie + coord_polar(theta = "y")
```

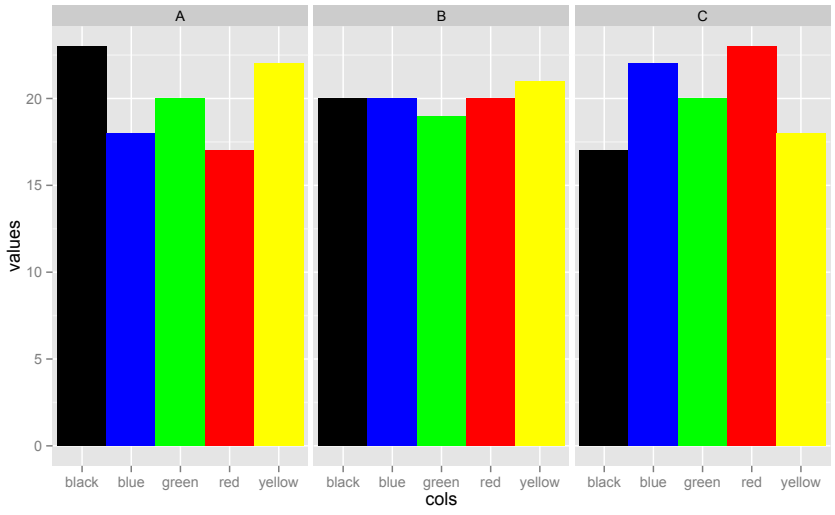


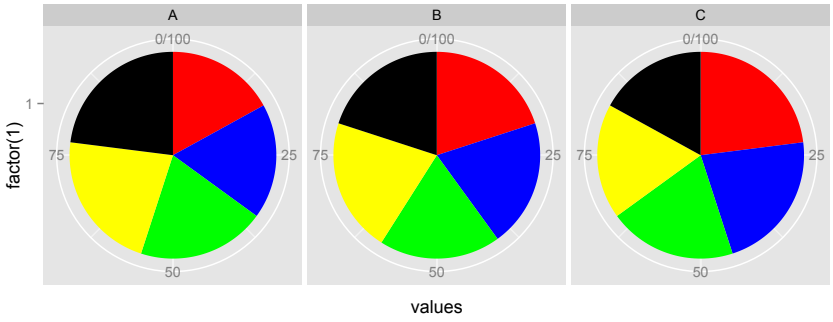
D.13 Pie charts vs. bar plots example

There is an example figure widely used in Wikipedia to show how much easier it is to 'read' bar plots than pie charts (`http://commons.wikimedia.org/wiki/File:Piecharts.svg?uselang=en-gb`).

Here is my `ggplot2` version of the same figure, using much simpler code and obtaining almost the same result.

```
example.data <-
  data.frame(values = c(17, 18, 20, 22, 23,
                        20, 20, 19, 21, 20,
                        23, 22, 20, 18, 17),
             examples= rep(c("A", "B", "C"), c(5,5,5)),
             cols = rep(c("red", "blue", "green", "yellow", "black"), 3)
  )

ggplot(example.data, aes(x=cols, y=values, fill=cols)) +
  geom_bar(width = 1, stat="identity") +
  facet_grid(.~examples) +
  scale_fill_identity()
ggplot(example.data, aes(x=factor(1), y=values, fill=cols)) +
  geom_bar(width = 1, stat="identity") +
  facet_grid(.~examples) +
  scale_fill_identity() +
  coord_polar(theta="y")
```





D.14 A classical example about regression

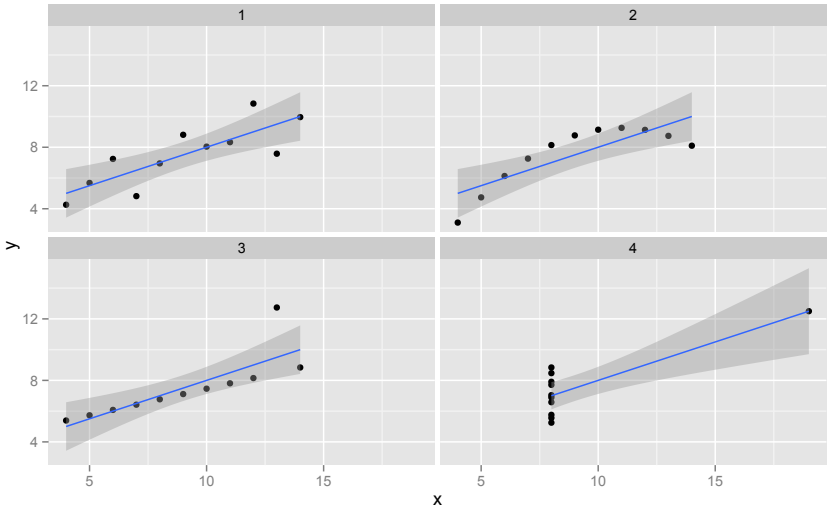
This is another figure from Wikipedia `http://commons.wikimedia.org/wiki/File:Anscombe.svg?uselang=en-gb`. The original code (not run):

```
svg("anscombe.svg", width=10.5, height=7)
par(las=1)

##-- some "magic" to do the 4 regressions in a loop:
ff <- y ~ x
for(i in 1:4) {
  ff[2:3] <- lapply(paste(c("y","x"), i, sep=""), as.name)
  ## or   ff2 <- as.name(paste("y", i, sep=""))
  ##      ff3 <- as.name(paste("x", i, sep=""))
  assign(paste("lm.",i,sep=""), lmi <- lm(ff, data= anscombe))
}

## Now, do what you should have done in the first place: PLOTS
op <- par(mfrow=c(2,2), mar=1.5+c(4,3.5,0,1), oma=c(0,0,0,0),
          lab=c(6,6,7), cex.lab=1.5, cex.axis=1.3, mgp=c(3,1,0))
for(i in 1:4) {
  ff[2:3] <- lapply(paste(c("y","x"), i, sep=""), as.name)
  plot(ff, data =anscombe, col="red", pch=21, bg = "orange", cex = 2.5,
       xlim=c(3,19), ylim=c(3,13),
       xlab=eval(substitute(expression(x[i]), list(i=i))),
       ylab=eval(substitute(expression(y[i]), list(i=i))))
  abline(get(paste("lm.",i,sep="")), col="blue")
}

dev.off()
```

My version using `ggplot2`:

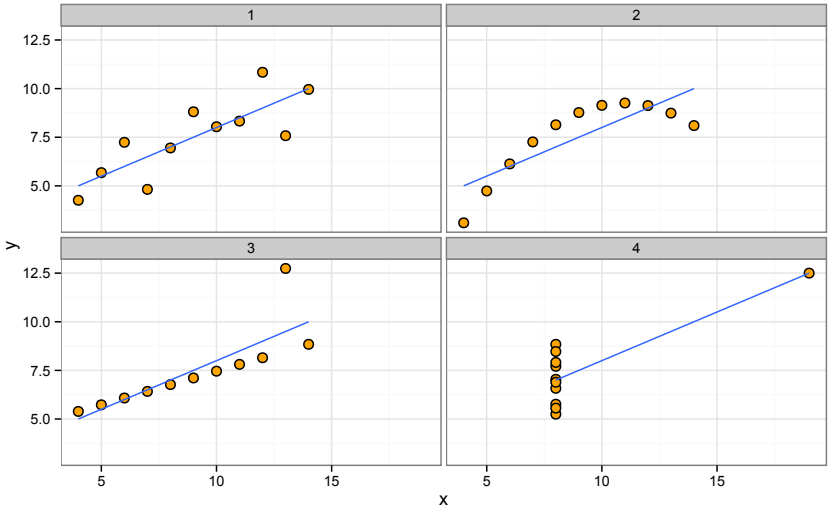
```
# we rearrange the data
my.mat <- matrix(as.matrix(anscombe), ncol=2)
```

```
my.anscombe <- data.frame(x = my.mat[ , 1], y = my.mat[ , 2], case=factor(rep(1:4, r
# we draw the figure
ggplot(my.anscombe, aes(x,y)) +
  geom_point() +
  geom_smooth(method="lm") +
  facet_wrap(~case, ncol=2)
```



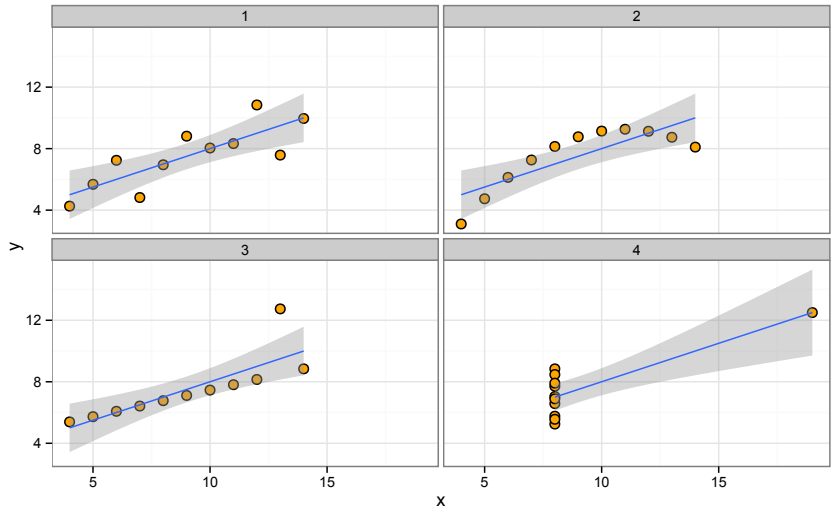
It is not much more difficult to make it look similar to the original

```
ggplot(my.anscombe, aes(x,y)) +
  geom_point(shape=21, fill="orange", size=3) +
  geom_smooth(method="lm", se=FALSE) +
  facet_wrap(~case, ncol=2) +
  theme_bw()
```



Although I think that the confidence bands make the point of the example much clearer

```
ggplot(my.anscombe, aes(x,y)) +
  geom_point(shape=21, fill="orange", size=3) +
  geom_smooth(method="lm") +
  facet_wrap(~case, ncol=2) +
  theme_bw()
```



This classical example form Anscombe **xxx** demonstrates four very different data sets that yield exactly the same results when a linear regression model is fit to them, including $R^2 = 0.666$. It is usually presented as a warning about the need to check model fits beyond looking at R^2 and other parameter's estimates.

D.15 Ternary plots

Being an extension to `ggplot2` the main difference is that a ternary plot can be created using `coord_tern` and that the three aesthetics `x, y, z` are required. By default the values of the variables mapped to these aesthetics are re-expressed as percentages or fractions. We present here only a few examples, and we encourage the readers to check the package's web site at `http://www.ggtern.com`.

For the first example we first generate some random data values from the uniform distribution:

```
# create some artificial data
my.trn1.data <- data.frame(x=runif(50), y=runif(50), z=runif(50))
```

A ternary plot is just a plot with a different system of coordinates, and can be obtained using `coord_tern`:

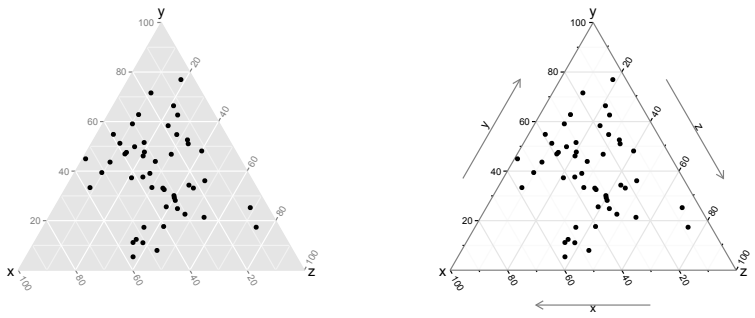
```
fig.trn <- ggplot(my.trn1.data, aes(x,y,z)) +
  coord_tern(L="x",T="y",R="z")
```

One can achieve a similar result by using `ggtern` instead of `ggplot`:

```
fig.trn <- ggtern(my.trn1.data, aes(x,y,z))
```

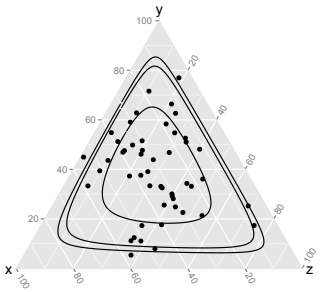
As with any other plot based on `ggplot2` one builds the plot by adding 'layers'. Themes are also supported.

```
fig.trn +
  geom_point()
fig.trn +
  geom_point() +
  theme_bw()
```



It is possible to also draw confidence regions:

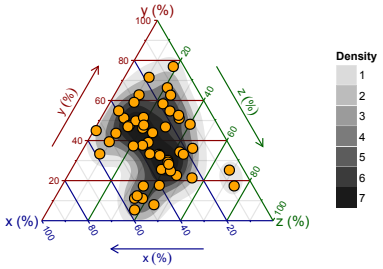
```
fig.trn +
  geom_point() +
  geom_confidence()
```



Or density estimates. In this last version of the plot I adjust a few other aesthetics and refine the appearance of the plot:

```
fig.trn +
  stat_density2d(fullrange=T,n=200,
                        geom="polygon", fill="grey10",
                        aes(alpha =..level..)) +
  geom_point(shape=21, fill="orange", size=4) +
```

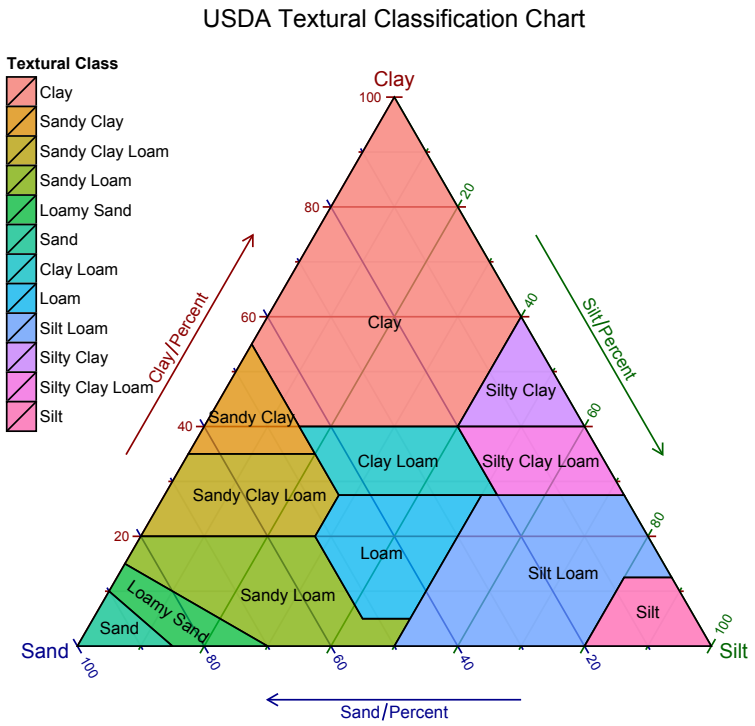
```
  labs(x="x (%)", y="y (%)", z="z (%)", alpha="Density") +
  theme_rgbw()
```



As a final example we reproduce an elaborate ternary plot from `http://www.ggtern.com/2014/01/15/usda-textural-soil-classification/`, the website of the package.

```
# Load the Data. (Available in ggtern 1.0.3.0 next version)
data(USDA)

# Put tile labels at the midpoint of each tile.
USDA.LAB = ddply(USDA, 'Label', function(df) {
    apply(df[, 1:3], 2, mean)
})

# Tweak
USDA.LAB$Angle = 0
USDA.LAB$Angle[which(USDA.LAB$Label == 'Loamy Sand')] = -35
```

```
# Construct the plot.
ggplot(data = USDA, aes(y=Clay, x=Sand, z=Silt,
                        color = Label,
                        fill = Label)) +
  coord_tern(L="x",T="y",R="z") +
  geom_polygon(alpha = 0.75, size = 0.5, color = 'black') +
  geom_text(data = USDA.LAB,
            aes(label = Label, angle = Angle),
            color = 'black',
            size = 3.5) +
  theme_rgbw() +
  theme_showsecondary() +
  theme_showarrows() +
  custom_percent("Percent") +
  theme(legend.justification = c(0, 1),
        legend.position      = c(0, 1),
        axis.tern.padding    = unit(0.15, 'npc')) +
  labs(title = 'USDA Textural Classification Chart',
       fill  = 'Textural Class',
       color = 'Textural Class')
```




```
try(detach(package:ggtern))
```

D.16 Plotting data onto maps

```
library(ggmap)
library(rgdal)

## Loading required package:  sp
## rgdal:  version:  0.9-1, (SVN revision 518)
## Geospatial Data Abstraction Library extensions to R
successfully loaded
## Loaded GDAL runtime:  GDAL 1.11.0, released 2014/04/16
## Path to GDAL shared files:  C:/Users/aphalo/Documents/R/win-library/3.1/rgdal/gda
## GDAL does not use iconv for recoding strings.
## Loaded PROJ.4 runtime:  Rel.  4.8.0, 6 March 2012,
[PJ_VERSION: 480]
## Path to PROJ.4 shared files:  C:/Users/aphalo/Documents/R/win-library/3.1/rgdal/p
```

Another extension to package `ggplot2` is package `ggmap`. Package `ggmap` makes it possible to plot data using normal `ggplot2` syntax on top of a map. Maps can be easily retrieved from the internet through different services. Some of these services require the user to register and obtain a key

for access. As Google Maps do not require such a key for normal resolution maps, we use this service in the examples.

The first step is to fetch the desired map. One can fetch the maps base on any valid Google Maps search term, or by giving the coordinates at the center of the map. Although `zoom` defaults to "auto", frequently the best result is obtained by providing this argument. Valid values for `zoom` are integers in the range 1 to 20.

> We will fetch maps from Google Maps. We have disabled the messages, to avoid repeated messages about Google's terms of use.
>
> **Google Maps API Terms of Service:** `http://developers.google.com/maps/terms`
>
> **Information from URL:** `http://maps.googleapis.com/maps/api/geocode/json?address=Europe&sensor=false`
>
> **Map from URL:** `http://maps.googleapis.com/maps/api/staticmap?center=Europe&zoom=3&size=%20640x640&scale=%202&maptype=terrain&sensor=false`

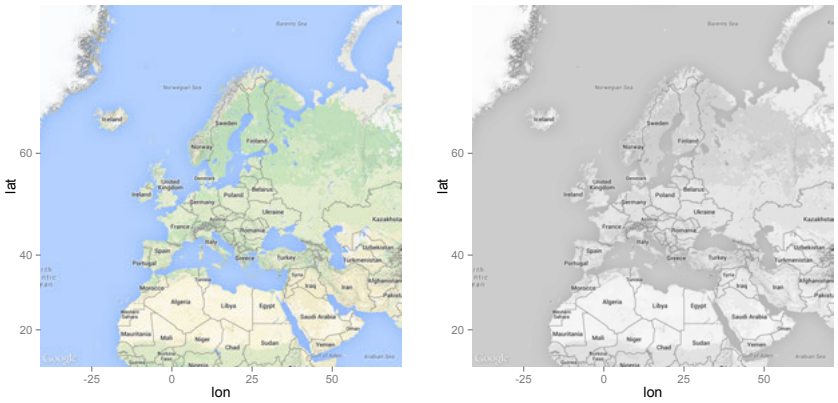
We start by fetching and plotting a map of Europe of type `satellite`. We use the default extent `panel`, and also the extent `device` and `normal`. The `normal` plot includes axes showing the coordinates, while `device` does not show them, while `panel` shows axes but the map fits tightly into the drawing area:

```
Europe1 <- get_map("Europe", zoom=3, maptype="satellite")
ggmap(Europe1)

ggmap(Europe1, extent = "device")

ggmap(Europe1, extent = "normal")
```

To demonstrate the option to fetch a map in black and white instead of the default colour version, we use a map of Europe of type `terrain`.

```
Europe2 <- get_map("Europe", zoom=3,
                   maptype="terrain")
```

```
ggmap(Europe2)

Europe3 <- get_map("Europe", zoom=3,
                   maptype="terrain",
                   color="bw")
ggmap(Europe3)
```



To demonstrate the difference between type `roadmap` and the default type `terrain`, we use the map of Finland. Note that we search for "Oulu" instead of "Finland" as Google Maps takes the position of the label "Finland" as the center of the map, and clips the northern part. By means of `zoom` we override the default automatic zooming onto the city of Oulu.

```
Finland1 <- get_map("Oulu", zoom=5, maptype="terrain")
ggmap(Finland1)

Finland2 <- get_map("Oulu", zoom=5, maptype="roadmap")
ggmap(Finland2)
```

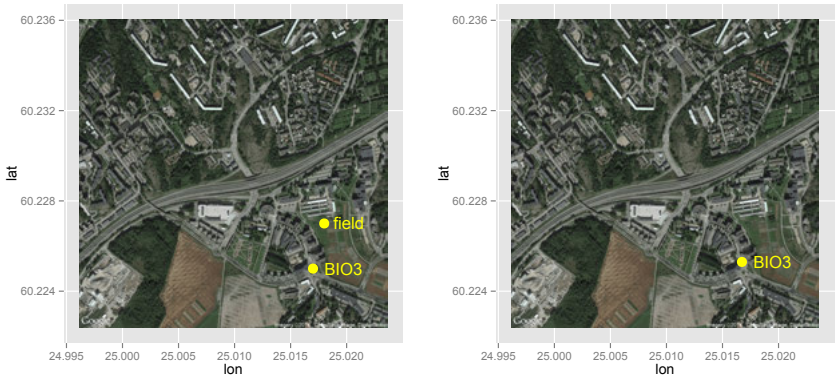
We can even search for a street address, and in this case with high zoom value, we can see the building where one of us works:

```
BIO3 <- get_map("Viikinkaari 1, 00790 Helsinki",
                zoom=18,
                maptype="satellite")
```

```
ggmap(BIO3)
```

We will now show a simple example of plotting data on a map, first by explicitly giving the coordinates, and in the second example we show how to fetch from Google Maps coordinate values that can be then plotted. We use function `geocode`. In one example we use `geom_point` and `geom_text`, while in the second example we use `annotate`, but either approach could have been used for both plots:

```
viikki <- get_map("Viikki",
                  zoom=15,
                  maptype="satellite")

our_location <- data.frame(lat=c(60.225, 60.227),
                           lon=c(25.017, 25.018),
                           label=c("BIO3", "field"))
ggmap(viikki, extent = "normal") +
  geom_point(data=our_location, aes(y=lat, x=lon),
             size=4, colour="yellow") +
  geom_text(data=our_location, aes(y=lat, x=lon, label=label),
            hjust=-0.3, colour="yellow")

our_geocode <- geocode("Viikinkaari 1, 00790 Helsinki")
ggmap(viikki, extent = "normal") +
  annotate(geom="point",
           y=our_geocode[ 1, "lat"], x=our_geocode[ 1, "lon"],
           size=4, colour="yellow") +
  annotate(geom="text",
           y=our_geocode[ 1, "lat"], x=our_geocode[ 1, "lon"],
           label="BIO3", hjust=-0.3, colour="yellow")
```



Using `get_map` from package `ggmap` for drawing a world map is not possible at the time of writing. In addition a worked out example of how to plot shape files, and how to download them from a repository is suitable as our final example. We also show how to change the map projection. The example is adapted from a blog post at `http://rpsychologist.com/working-with-shapefiles-projections-and-world-maps-in-ggplot`.

We start by downloading the map data archive files from `http://www.naturalearthdata.com` which is available in different layers. We only use three of the available layers: 'physical' which describes the coastlines and a grid and bounding box, and 'cultural' which gives country borders. We save them in a folder with name 'maps', which is expected to already exist. After downloading each file, we unzip it.

```
oldwd <- setwd("./maps")

url_path <-
#   "http://www.naturalearthdata.com/download/110m/"
  "http://www.naturalearthdata.com/http//www.naturalearthdata.com/download/110m/"

download.file(paste(url_path,
                    "physical/ne_110m_land.zip",
                    sep=""), "ne_110m_land.zip")
unzip("ne_110m_land.zip")

download.file(paste(url_path,
                    "cultural/ne_110m_admin_0_countries.zip",
                    sep=""), "ne_110m_admin_0_countries.zip")
unzip("ne_110m_admin_0_countries.zip")

download.file(paste(url_path,
                    "physical/ne_110m_graticules_all.zip",
                    sep=""), "ne_110m_graticules_all.zip")
unzip("ne_110m_graticules_all.zip")

setwd(oldwd)
```

We list the layers that we have downloaded.

```
ogrListLayers(dsn="./maps")
```

```
## [1] "ne_110m_admin_0_countries"
## [2] "ne_110m_graticules_1"
## [3] "ne_110m_graticules_10"
## [4] "ne_110m_graticules_15"
## [5] "ne_110m_graticules_20"
## [6] "ne_110m_graticules_30"
## [7] "ne_110m_graticules_5"
## [8] "ne_110m_land"
## [9] "ne_110m_wgs84_bounding_box"
## attr(,"driver")
## [1] "ESRI Shapefile"
## attr(,"nlayers")
## [1] 9
```

Next we read the layer for the coastline, and use `fortify` to convert it into a data frame. We also create a second version of the data using the Robinson projection.

```
wmap <- readOGR(dsn="./maps", layer="ne_110m_land")

## OGR data source with driver: ESRI Shapefile
## Source: "./maps", layer: "ne_110m_land"
## with 127 features and 2 fields
## Feature type: wkbPolygon with 2 dimensions

wmap.data <- fortify(wmap)

## Regions defined for each Polygons

wmap_robin <- spTransform(wmap, CRS("+proj=robin"))
wmap_robin.data <- fortify(wmap_robin)

## Regions defined for each Polygons
```

We do the same for country borders,

```
countries <- readOGR("./maps", layer="ne_110m_admin_0_countries")

## OGR data source with driver: ESRI Shapefile
## Source: "./maps", layer: "ne_110m_admin_0_countries"
## with 177 features and 63 fields
## Feature type: wkbPolygon with 2 dimensions

countries.data <- fortify(countries)

## Regions defined for each Polygons

countries_robin <- spTransform(countries, CRS("+init=ESRI:54030"))
countries_robin.data <- fortify(countries_robin)

## Regions defined for each Polygons
```

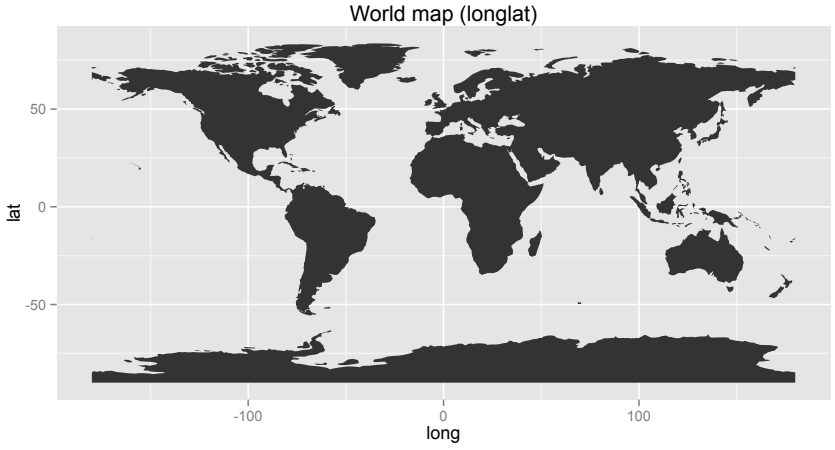
and for the graticule at 15° intervals, and the bounding box.

```
grat <- readOGR("./maps", layer="ne_110m_graticules_15")

## OGR data source with driver: ESRI Shapefile
## Source: "./maps", layer: "ne_110m_graticules_15"
```

```
## with 35 features and 5 fields
## Feature type: wkbLineString with 2 dimensions

grat.data <- fortify(grat)
grat_robin <- spTransform(grat, CRS("+proj=robin"))
grat_robin.data <- fortify(grat_robin)

bbox <- readOGR("./maps", layer="ne_110m_wgs84_bounding_box")

## OGR data source with driver: ESRI Shapefile
## Source: "./maps", layer: "ne_110m_wgs84_bounding_box"
## with 1 features and 2 fields
## Feature type: wkbPolygon with 2 dimensions

bbox.data <- fortify(bbox)

## Regions defined for each Polygons

bbox_robin <- spTransform(bbox, CRS("+proj=robin"))
bbox_robin.data <- fortify(bbox_robin)

## Regions defined for each Polygons
```

Now we plot the world map of the coastlines, on a longitude and latitude scale, as a `ggplot` using `geom_polygon`.

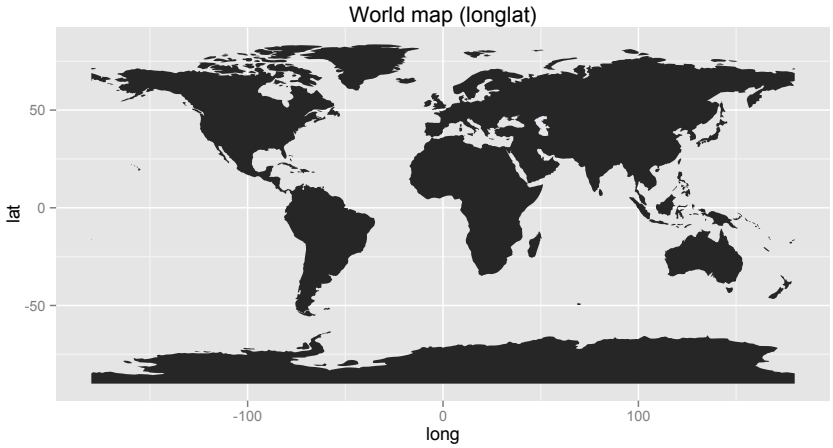
```
ggplot(wmap.data, aes(long,lat, group=group)) +
  geom_polygon() +
  labs(title="World map (longlat)") +
  coord_equal()
```



There is one noticeable problem in the map shown above: the Caspian sea is missing. We need to use aesthetic `fill` and a manual scale to correct this.

```
ggplot(wmap.data, aes(long,lat, group=group, fill=hole)) +
  geom_polygon() +
  labs(title="World map (longlat)") +
  scale_fill_manual(values=c("#262626", "#e6e8ed"),
```

```
                      guide="none") +
  coord_equal()
```



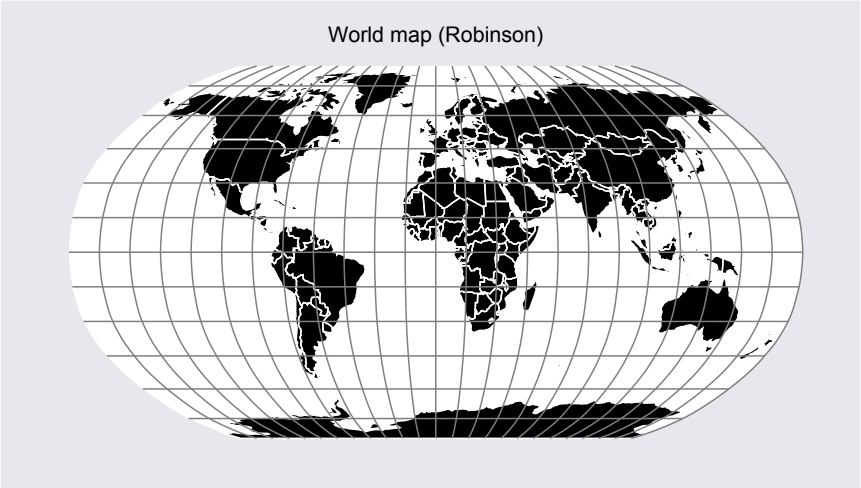
When plotting a map using a projection, many default elements of the `ggplot` theme need to be removed, as the data is no longer in units of degrees of latitude and longitude and axes and their labels are no longer meaningful.

```
theme_map_opts <-
  list(theme(panel.grid.minor = element_blank(),
             panel.grid.major = element_blank(),
             panel.background = element_blank(),
             plot.background = element_rect(fill="#e6e8ed"),
             panel.border = element_blank(),
             axis.line = element_blank(),
             axis.text.x = element_blank(),
             axis.text.y = element_blank(),
             axis.ticks = element_blank(),
             axis.title.x = element_blank(),
             axis.title.y = element_blank()))
```

Finally we plot all the layers using the Robinson projection. This is still a `ggplot` and consequently one can plot data on top of the map, being aware of the transformation of the scale needed to make the data location match locations in a map using a certain projection.

```
ggplot(bbox_robin.data, aes(long,lat, group=group)) +
  geom_polygon(fill="white") +
  geom_polygon(data=countries_robin.data,
               aes(long,lat, group=group,
                   fill=hole)) +
  geom_path(data=countries_robin.data,
            aes(long,lat, group=group, fill=hole),
            color="white",
            size=0.3) +
  geom_path(data=grat_robin.data,
            aes(long, lat, group=group, fill=NULL),
            linetype="dashed",
```

```
            color="grey50") +
  labs(title="World map (Robinson)") +
  coord_equal() +
  theme_map_opts +
  scale_fill_manual(values=c("black", "white"),
                    guide="none")
```




```
try(detach(package:ggmap))
try(detach(package:rgdal))
```

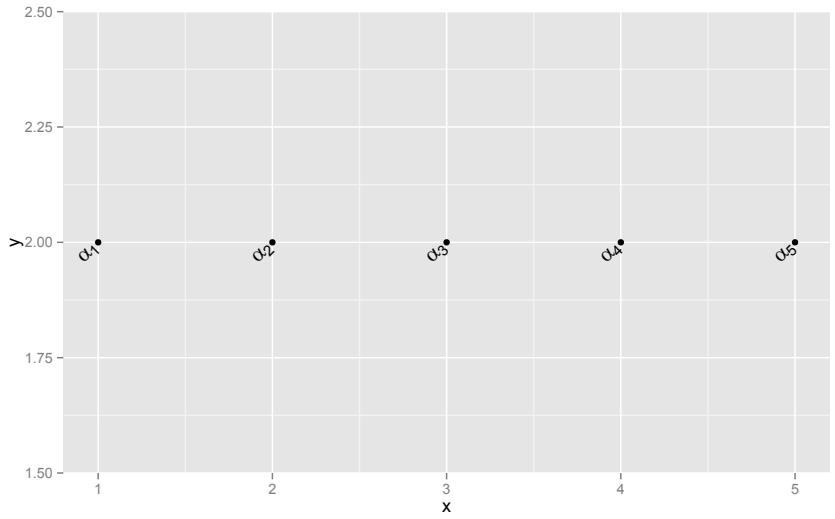
D.17 Advanced topics

D.18 Using `plotmath` expressions

Expressions are very useful but rather tricky to use because the syntax is unusual. In `ggplot` one can either use expressions explicitly, or supply them as character string labels, and tell `ggplot` to parse them. For titles, axis-labels, etc. (anything that is defined with `labs`) the expressions have to entered explicitly, or saved as such into a variable, and the variable supplied as argument. When plotting expressions using `geom_text` expression arguments should be supplied as character strings and the optional argument `parse=TRUE` used to tell the geom to interpret the labels as expressions. We will go through a few useful examples.

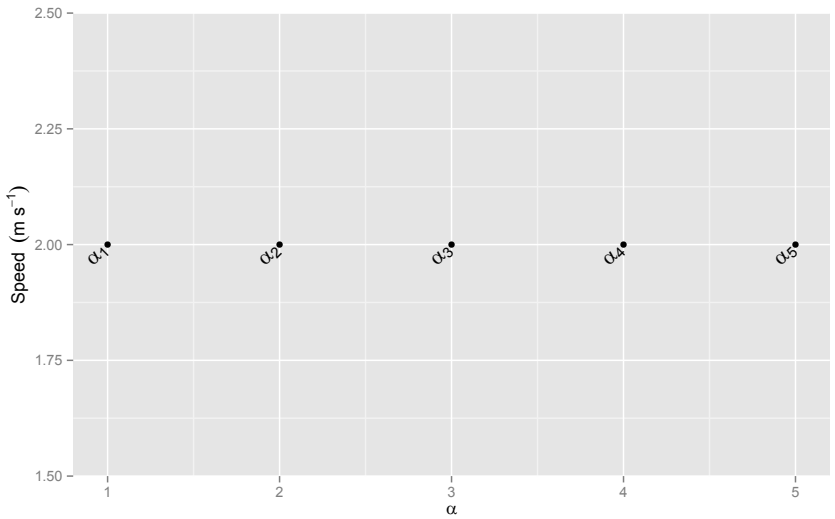
We will revisit the example from the previous section, but now using subscripted Greek α for labels. In this example we use as subscripts numeric values from another variable in the same dataframe.

```
my.data$greek.label <- paste("alpha[", my.data$x, "]", sep="")
(fig <- ggplot(my.data, aes(x,y,label=greek.label)) +
   geom_text(angle=45, hjust=1.2, parse=TRUE) + geom_point())
```



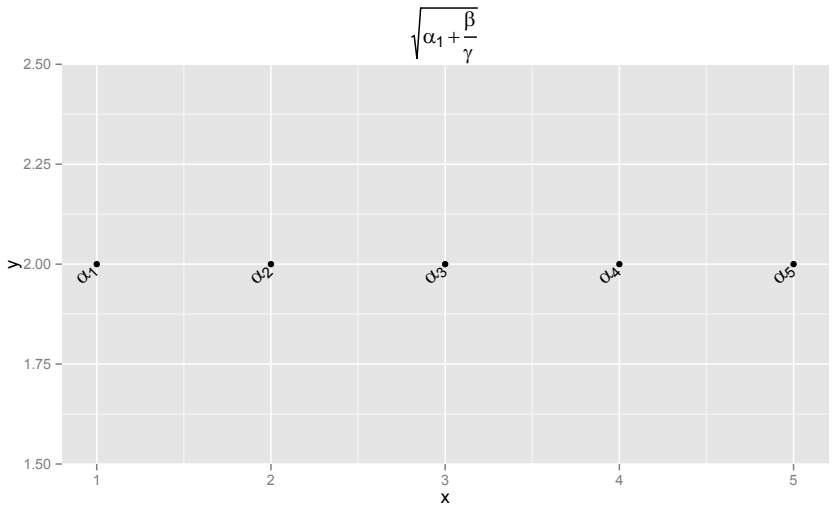
Setting an axis label with superscripts. The easiest way to deal with spaces is to use ‘ ’ or ‘ ’. One can connect pieces that would otherwise cause errors using ‘*’. If we

```
fig + labs(x=expression(alpha), y=expression(Speed~~(m~s^{-1})))
```



It is possible to store expressions in variables.

```
my.title <- expression(sqrt(alpha[1] + frac(beta, gamma)))
fig + labs(title=my.title)
```



Annotations are plotted ignoring the default aesthetics, but still make use of geoms, so labels for annotations also have to be supplied as character strings and parsed.

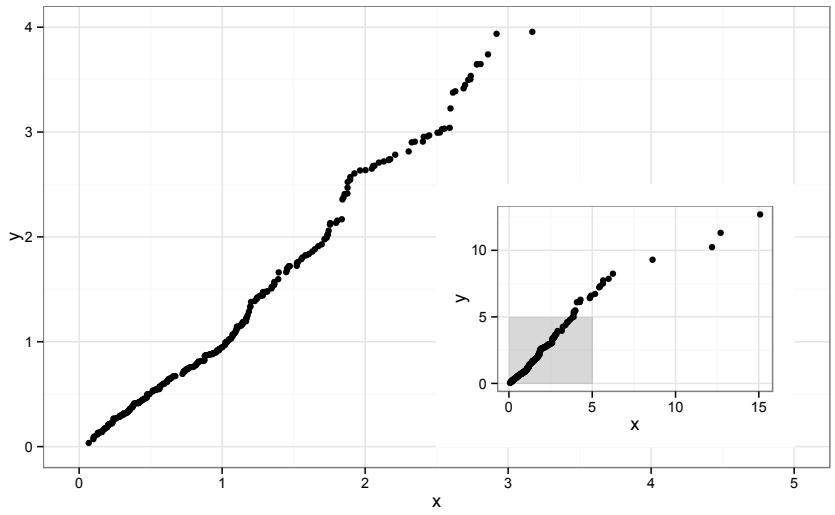
```
fig + ylim(1,3) +
  annotate("text", label="sqrt(alpha[1] + frac(beta, gamma))",
           y=2.5, x=3, size=8, colour="red", parse=TRUE)
```

We discuss how to use expressions as facet labels in section **??**.

D.18.1 Inset plots using same data

Example from `http://stackoverflow.com/questions/20708012/embedding-a-subplot-in-ggplot-ggsubplot`, authored by Baptiste Auguié `http://baptiste.github.io/`.

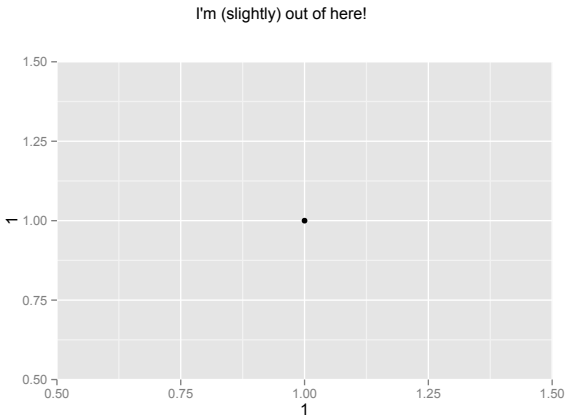
```
d = data.frame(x = sort(rlnorm(300)),
               y = sort(rlnorm(300)),
               grp = 1)

main <- ggplot(d, aes(x, y)) +
  geom_point() + theme_bw()

sub <- main +
  geom_rect(data=d[1,],
            xmin=0, ymin=0, xmax=5, ymax=5,
            fill="grey50", alpha=0.3)
sub$layers <- rev(sub$layers) # draw rect below

main +
  annotation_custom(ggplotGrob(sub),
                    xmin=2.5, xmax=5,
                    ymin=0, ymax=2.5) +
  scale_x_continuous(limits=c(0, 5)) +
  scale_y_continuous(limits=c(0, 4))

## Warning:  Removed 28 rows containing missing values
(geom_point).
```



D.18.2 Adding elements using `grid`

`ggplot2` creates the plots using package `grid`, consequently it is possible to manipulate `ggplot` objects using `grid` functions. Here we present a very simple example. For more information on using `grid` together with `ggplot2` please see **Murriel2009**

```
print(qplot(1,1), vp=viewport(height=0.8))
grid.text(0.5, unit(1,"npc") - unit(1,"line"),
          label="I'm (slightly) out of here!")
```



D.19 Generating output files

It is possible, when using RStudio, to directly export the displayed plot to a file. However, if the file will have to be generated again at a later time, or a series of plots need to be produced with consistent format, it is best to include the commands to export the plot in the script.

In R, files are created by printing to different devices. Printing is directed to a currently open device. Some devices produce screen output, others files. Devices depend on drivers. There are both devices that or part of R, and devices that can be added through packages.

A very simple example of PDF output (width and height in inches):

```
fig1 <- ggplot(data.frame(x=-3:3), aes(x=x)) +
  stat_function(fun=dnorm)
pdf(file="fig1.pdf", width=8, height=6)
print(fig1)
dev.off()
```

Encapsulated Postscript output (width and height in inches):

```
postscript(file="fig1.eps", width=8, height=6)
print(fig1)
dev.off()
```

There are Graphics devices for BMP, JPEG, PNG and TIFF format bitmap files. In this case the default units for width and height is pixels. For example we can generate TIFF output:

```
tiff(file="fig1.tiff", width=1000, height=800)
print(fig1)
dev.off()
```

```
try(detach(package:scales))
try(detach(package:plyr))
try(detach(package:Hmisc))
try(detach(package:ggplot2))
try(detach(package:grid))
```

Further reading about R

E.1 Temporary list

APPENDIX F

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] splines   methods   tools     stats
## [5] graphics  grDevices utils     datasets
```

```
## [9] base
##
## other attached packages:
## [1] sp_1.0-16       Formula_1.1-2
## [3] survival_2.37-7 lattice_0.20-29
## [5] splus2R_1.2-0   proto_0.3-10
## [7] lubridate_1.3.3 stringr_0.6.2
## [9] knitr_1.8
##
## loaded via a namespace (and not attached):
##  [1] acepack_1.3-3.3
##  [2] chron_2.3-45
##  [3] cluster_1.15.3
##  [4] colorspace_1.2-4
##  [5] data.table_1.9.4
##  [6] digest_0.6.4
##  [7] evaluate_0.5.5
##  [8] foreign_0.8-61
##  [9] formatR_1.0
## [10] GGally_0.4.8
## [11] ggmap_2.3
## [12] ggplot2_1.0.0
## [13] ggtern_1.0.3.2
## [14] grid_3.1.2
## [15] gridExtra_0.9.1
## [16] gtable_0.1.2
## [17] highr_0.4
## [18] Hmisc_3.14-6
## [19] labeling_0.3
## [20] latticeExtra_0.6-26
## [21] mapproj_1.2-2
## [22] maps_2.3-9
## [23] MASS_7.3-35
## [24] memoise_0.2.1
## [25] munsell_0.4.2
## [26] nnet_7.3-8
## [27] photobiology_0.4.8
## [28] photobiologyFilters_0.1.13
## [29] photobiologygg_0.1.14
## [30] photobiologyLEDs_0.1.4
## [31] photobiologyWavebands_0.1.0
## [32] plyr_1.8.1
## [33] png_0.1-7
## [34] RColorBrewer_1.0-5
## [35] Rcpp_0.11.3
## [36] reshape_0.8.5
## [37] reshape2_1.4
## [38] rgdal_0.9-1
## [39] RgoogleMaps_1.2.0.6
## [40] rjson_0.2.15
## [41] RJSONIO_1.3-0
## [42] rpart_4.1-8
## [43] scales_0.2.4
```