R for Photobiology

A handbook

Pedro J. Aphalo
and
Andreas Albert

DRAFT
16th July 2014

Contents

Preface

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

Acknowledgements

We thank Titta Kotilainen, Stefano Catola, and ... for very useful comments and suggestions.

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 Sliney2007.

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 Caldwell1971.
GEN(G)	mathematical formulation of GEN by Green1974 .
GEN(T)	mathematical formulation of GEN by Thimijan1978.
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

Getting ready

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

All the software used for typesetting this handbook and developing the r4photobiology software suite if free and open source. All the software used 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 on a Linux server, 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 as 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 step is to use text editor which integrates to some extent with R, but still this is not ideal. 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 important in the case of errors and 'debugging'. We also use a LaTeX for typesetting. Is what we used for the handbook, and what we routinely for reporting data analyses, and that PJA also uses for all 'overhead' slides he writes. 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, in this by far the best one we have used 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

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, 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 released versions of packages are kept. Packages can be installed and updated from CRAN and similar repositories directly from within R.

In you are not familiar with R, please, go through the Appendixes A, B and C, before delving into our 'Cook Book'.

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.

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, analysts, authors and or editors to work on the same files in parallel and then merge their edits. They also allow easy transfer of the whole 'projects' between computers. Git is very popular, and Github and Bitbucket are popular hosts for repositories.

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.

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 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).

One of the main difficulties when working with spectral data is that one may need to operate on data sets measured at different wavelength values and steps sizes. The functions in the suite handle any mismatch by interpolation before applying operations or functions. 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`, and photon spectral irradiance `s.q.irrad`[1].

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.

[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	functions	basic functions and example data
•	photobiologyVIS	definitions	quantification of VIS radiation
•	photobiologyUV	definitions	quantification of UV 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	phototropins
•	photobiologygg	funtions	extensions to package `ggplot2`
•	rTUV	funs + data	TUV model interface
•	rOmniDriver	functions	control of Ocean Optics spectrometers

5.2 `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 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(photoCRAN =
                      "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(photoCRAN =
                      "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(photoCRAN =
                             "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. 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. 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 with:

```
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.

5.3 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 5.2 on the r4photo repository for an alternative to the approach given here.

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 my repository and a normal CRAN repository, then all dependencies will be automatically installed. The new 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.

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


```
install.packages(c("photobiologyAll", "photobiologygg"),
         repos = c(photoCRAN =
                     "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 my 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",
    "photobiologyVIS", "photobiologyUV",
    "photobiologyCry", "photobiologyPhy",
    "photobiologyLamps", "photobiologyLEDs",
    "photobiologySun", "photobiologygg",
    "photobiologyFilters",  "photobiologySensors")

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

The commands above install all my packages 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(photoCRAN =
                      "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 OS 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

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

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

```
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] 1.212e+13
```

The function is vectorized:

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

## [1] 3.355e+12 8.759e+12 1.212e+13
```

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

## [1] 6.388e+12 1.212e+13
```

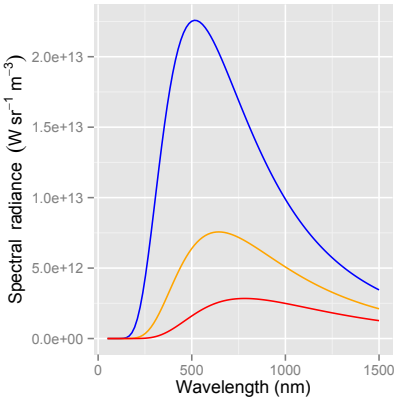
We aware that if two vectors are supplied, then the elements in each one are matched and recycled[1]:

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

## [1] 6.388e+12 1.212e+13 7.475e+12 1.278e+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)")
```

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



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

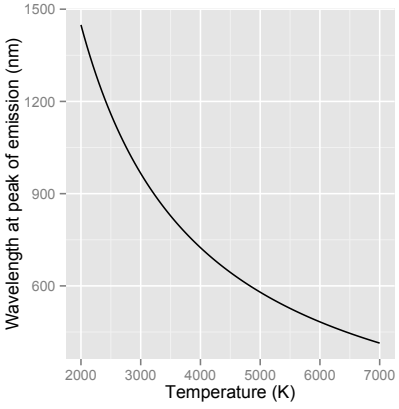
$$\lambda_{max} \cdot T = 2.898 \times 10^6 \,\mathrm{nm\,K} \tag{6.2}$$

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)")
```



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 24.94 60.17

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

##     lon   lat
## 1 25.02 60.23
```

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 -58.38 -34.6

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 14.33 hours
##
## $nightlength
## Time difference of 9.675 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 9.681 hours
```

```
##
## $nightlength
## Time difference of 14.32 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 11.58 48.14

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 8.146 hours
##
## $nightlength
## Time difference of 15.85 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 15.86 hours
##
## $nightlength
## Time difference of 8.14 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 24.94 60.17

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 12.13 hours
##
## $nightlength
## Time difference of 11.87 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 10.51 hours
##
## $nightlength
## Time difference of 13.49 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 8.846 hours
##
## $nightlength
## Time difference of 15.15 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 7.041 hours
##
## $nightlength
## Time difference of 16.96 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 29.76 62.6

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] 267.6
##
## $elevation
## [1] 25.82
##
## $diameter
## [1] 0.526
##
## $distance
## [1] 1.014
```

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] "2014-07-16 10:12:12 EEST"
##
## $azimuth
## [1] 123.8
##
## $elevation
## [1] 39.29
##
## $diameter
## [1] 0.5246
##
## $distance
## [1] 1.016
```

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,
```

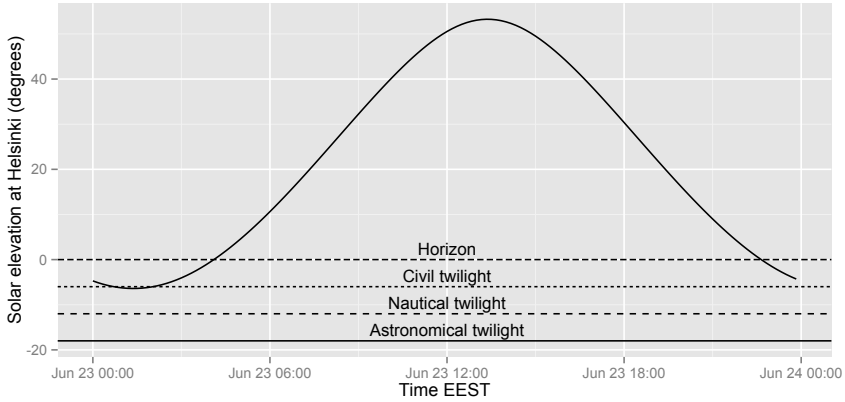
```
                              elevation = elevations,
                              location = "Helsinki",
                              lon = geo_code_He[["lon"]],
                             lat = geo_code_He[["lat"]])
```

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:

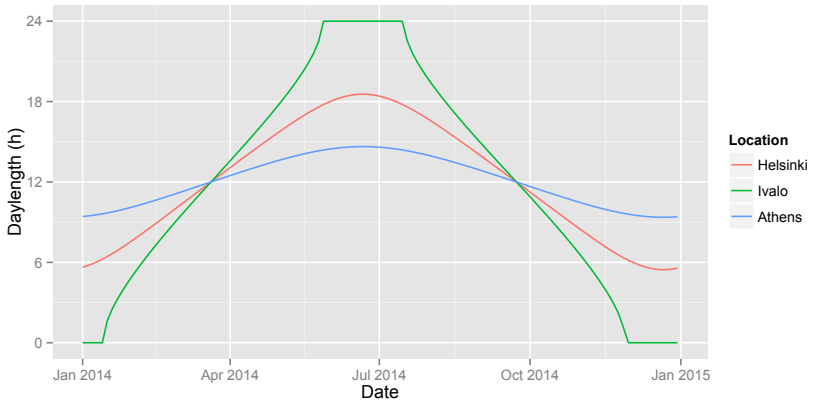
```
days <- seq(from=ymd("2014-01-01"), to=ymd("2014-12-31"),
            by="3 day")
```

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")
```



CHAPTER 8

Basic operations on spectra

Abstract

In this chapter we describe the use of a few basic functions, which can be useful when no predefined functions are available for a given operation.

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(photobiologyFilters)
library(photobiologyLEDs)
```

8.2 Introduction

How are example spectra stored?

The suite uses to some extent object-oriented programming. Objects are implemented using "S3" classes. 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" and "source.spct" are derived. Having this class structure allows us to create special methods and operators, which use the same names than the generic ones but take into account the special properties of spectra. Each spectrum object can hold only one spectrum.

Objects of class "source.spct" have three components `w.length`, `s.e.irrad` and `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 have a "comment" attribute with a textual description.

Objects of class "filter.spct" have three components `w.length`, `Tfr` and `Tpc`. 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`. Objects have a "comment" attribute with a textual description.

What operators are available for operations between spectra?

Operators are defined for these objects. These are an easy and simple way of doing calculations, but are rather inflexible and performance is expected to be slower than with functions with additional parameters. The operators are defined so that an operation between two "filter.spct" objects yields another "filter.spct" object, and operation between a "filter.spct" object and a "source.spct", or between two "source.spct" objects yields a "source.spct" object. 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.

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

The same four 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 components such as spectral irradiance and transmittance.

What functions are available for operations between spectra?

Logarithms (`log, log10`), square root (`sqrt`) and exponentiation (`exp`) are defined for spectra. These functions are not applied on `w.length`, but instead on the other components such as spectral irradiance and spectral transmittance.

What 'summary' functions are available for spectra?

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

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[1].

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

## Error:  object 'gg400.dt' not found

head(filtered_sun.spct)

## Error:  object 'filtered_sun.spct' not found
```

The GG440 data is for internal transmittance. Let's assume a filter with 9% reflection across all wavelengths (a coarse approximation):

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

## Error:  object 'gg400.dt' not found

head(filtered_uncoated_sun.spct)

## Error:  object 'filtered_uncoated_sun.spct' not found
```

Calculations related to filters will be explained in detail in chapter 15. This is just an example of how the operators work.

8.3 Task: uniform scaling of a spectrum

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

```
head(sun.spct)

##    w.length s.e.irrad s.q.irrad
## 1:      293 2.610e-06 6.392e-12
## 2:      294 6.142e-06 1.510e-11
## 3:      295 2.176e-05 5.366e-11
## 4:      296 6.780e-05 1.678e-10
## 5:      297 1.533e-04 3.807e-10
## 6:      298 3.670e-04 9.141e-10

head(sun.spct * 2)

##    w.length s.e.irrad s.q.irrad
## 1:      293 5.219e-06 1.278e-11
## 2:      294 1.228e-05 3.019e-11
## 3:      295 4.352e-05 1.073e-10
## 4:      296 1.356e-04 3.355e-10
## 5:      297 3.067e-04 7.614e-10
## 6:      298 7.339e-04 1.828e-09
```

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.

[1] Here and in many other examples `head` is used to limit the amount of output to a few lines.

8.4 Task: simple operations between two spectra

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

## Error:  object 'ug1.dt' not found

head(filtered_sun.spct)

## Error:  object 'filtered_sun.spct' not found
```

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

8.5 Task: other operations between two spectra

If data for two spectra are available for the same wavelength values, then we can still use the built in R mat operators. These operators are vectorized, which means that an addition between two vectors adds the elements at each position. A non-nonsensical example follows:

```
head(with(sun.data, s.e.irrad^2 / w.length))

## [1] 2.324e-14 1.283e-13 1.605e-12 1.553e-11
## [5] 7.918e-11 4.519e-10
```

Operations using built-in R operators cannot be done if the wavelengths in two spectral data sets do not match. In this situation is where functions 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). The functions are `sum_spectra`, `subt_spectra`, `prod_spectra`, `div_spectra`, and `oper_spectra`. Here is a very simple hypothetical example:

```
out1.dt <- sum_spectra(spc1$w.length, spc2$w.length,
                       spc1$s.e.irrad, spc2$s.e.irrad)
```

This is what happens under the hood when we execute code like:

```
out2.spct <- spc1 + spc2
```

with the difference that in the first case only spectral energy irradiance is calculated, while in the second both spectral energy irradiance and spectral photon irradiance are returned. `out1.data` is a "data.table", while the second will be an 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.

The function `oper_spectra` takes the operator to use as an argument:

```
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.

8.6 Task: trimming a spectrum

This is basically a subsetting operation, but the function `trim_tails` adds a few 'bells and whistles'. The trimming is based on wavelengths, by default the cut points are inserted by interpolation, so that the spectrum returned includes the limits given as arguments. By default the trimming is done by deleting both spectral irradiance and wavelength values outside the range delimited by the limits, but through parameter `fill` the values outside the limits can be replaced any value desired (most commonly NAor 0.) It is possible to supply only one, or both of `low.limit` and `high.limit`, depending on the desired trimming.

```
head(with(sun.data,
          trim_tails(w.length, s.e.irrad,
                     low.limit=300)))
```

```
##   w.length  s.irrad
## 1      300 0.001265
## 2      301 0.002624
## 3      302 0.003923
## 4      303 0.008974
## 5      304 0.011656
## 6      305 0.017991
```

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

```
##   w.length  s.irrad
## 1      300 0.001265
## 2      301 0.002624
## 3      302 0.003923
## 4      303 0.008974
## 5      304 0.011656
## 6      305 0.017991
```

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

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

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

```
##   w.length s.irrad
## 1      293       0
## 2      294       0
## 3      295       0
## 4      296       0
## 5      297       0
## 6      298       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.

```
tail(with(sun.data,
          trim_tails(w.length, s.e.irrad,
                     low.limit=300, high.limit=1000)))

## Warning:  Ignoring high.limit as it is too high.

##     w.length s.irrad
## 496      795  0.4147
## 497      796  0.4081
## 498      797  0.4141
## 499      798  0.4236
## 500      799  0.4186
## 501      800  0.4069

tail(with(sun.data,
          trim_tails(w.length, s.e.irrad,
                     low.limit=300, high.limit=1000, fill=NA)))

##     w.length s.irrad
## 703      995      NA
## 704      996      NA
## 705      997      NA
## 706      998      NA
## 707      999      NA
## 708     1000      NA

tail(with(sun.data,
          trim_tails(w.length, s.e.irrad,
                     low.limit=300, high.limit=1000, fill=0.0)))

##     w.length s.irrad
## 703      995       0
## 704      996       0
## 705      997       0
## 706      998       0
## 707      999       0
## 708     1000       0
```

8.7 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[2], 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] 919.5
```

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

The functions are vectorized, so they can be applied to whole spectra, to convert W m^{-2} nm^{-1} to mol m^{-2} s^{-1} nm^{-1}:

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

##  [1] 2.610e-06 6.142e-06 2.176e-05 6.780e-05
##  [5] 1.533e-04 3.670e-04 7.845e-04 1.265e-03
##  [9] 2.624e-03 3.923e-03

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

##  [1] 6.392e-12 1.510e-11 5.366e-11 1.678e-10
##  [5] 3.807e-10 9.141e-10 1.961e-09 3.171e-09
##  [9] 6.602e-09 9.903e-09
```

8.8 Task: conversion from photon to energy base

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

In **aphalo2012** it is written: "Example 1: red light at 600 nm has about 200 kJ mol^{-1}, therefore, 1 µmol photons has 0.2 J. Example 2: UV-B radiation at

[2]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.

300 nm has about 400 $kJ\,mol^{-1}$, 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 is we multiply by 10^{-3} to obtain $kJ\,mol^{-1}$:

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

## [1] 199.4

as_energy(300, 1) * 1e-3

## [1] 398.8
```

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

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

##  [1] 6.392e-12 1.510e-11 5.366e-11 1.678e-10
##  [5] 3.807e-10 9.141e-10 1.961e-09 3.171e-09
##  [9] 6.602e-09 9.903e-09

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

##  [1] 2.610e-06 6.142e-06 2.176e-05 6.780e-05
##  [5] 1.533e-04 3.670e-04 7.845e-04 1.265e-03
##  [9] 2.624e-03 3.923e-03
```

8.9 Task: interpolating a spectrum

The function `interpolate_spectrum` is used internally for interpolating 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.

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`:

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

##  [1]        NA        NA        NA 2.610e-06
##  [5] 6.142e-06 2.176e-05 6.780e-05 1.533e-04
##  [9] 3.670e-04 7.845e-04 1.265e-03

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

##  [1] 0.000e+00 0.000e+00 0.000e+00 2.610e-06
##  [5] 6.142e-06 2.176e-05 6.780e-05 1.533e-04
##  [9] 3.670e-04 7.845e-04 1.265e-03
```

This function, in its current implementation, always returns interpolated values, even when the density of wavelengths in the output is less than that in the input. A future version will *likely* include a parameter for changing this behaviour to averaging or smoothing.

8.10 Internal-use functions

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.

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(photobiologyVIS)
library(photobiologyUV)
```

9.2 Introduction

9.3 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 $\lambda_1 = 400\mathrm{nm}$ and $\lambda_1 = 700\mathrm{nm}$. In this example we will use example data for sunlight to calculate $E_{400\,\mathrm{nm}<\lambda<700\,\mathrm{nm}}$:

```
with(sun.data,
     energy_irradiance(w.length, s.e.irrad,
                       new_waveband(400, 700)))

## range.400.700
##         196.7
```

Function PAR() is predefined in package photobiologyVIS as a convenience function, so the code above can be replaced by:

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

##   PAR
## 196.7
```

If no waveband is supplied as argument, then the whole range of wavelengths in the spectral data is used for the integration, and the 'name' attribute is generated accordingly:

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

## range.293.800
##         269.1
```

If a waveband that does not fully overlap with the data is supplied as argument, then spectral irradiance for wavelengths outside the range is assumed to be zero:

```
with(sun.data,
     energy_irradiance(w.length, s.e.irrad,
                       new_waveband(700,1000)))

## range.700.1000
##           44.1
```

If a waveband that does not overlap with the data is supplied as argument, then spectral irradiance for wavelengths outside the range is assumed to be zero:

```
with(sun.data,
     energy_irradiance(w.length, s.e.irrad,
                       new_waveband(100,200)))

## range.100.200
##             0
```

9.4 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. In this example we will use example data for sunlight.

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

##       PAR
## 0.0008938
```

If we want to have Q_{PAR} (PPFD) expressed in the usual units of $\mu\mathrm{mol\,m^{-2}\,s^{-1}}$, we need to multiply the result above by 10^6:

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

##   PAR
## 893.8
```

`PAR()` is predefined in package photobiologyVIS as a convenience function, see section **??** for an example with arbitrary values for λ_1 and λ_2.

9.5 Task: calculate energy and photon irradiances from spectral photon irradiance

In the case of the calculation of energy irradiance from spectral photon irradiance the calculation is:

$$E_{\lambda_1<\lambda<\lambda_2} = \int_{\lambda_1}^{\lambda_2} Q(\lambda) \frac{\lambda}{h' \cdot c} \mathrm{d}\,\lambda \tag{9.3}$$

And the code[1]:

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

##       PAR
## 0.0008938
```

The calculation of photon irradiance from spectral photon irradiance, is a simple integration, analogous to that in equation 9.1, and the code is:

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

##       PAR
## 4.158e-09
```

[1]The dataframe `sun.data` contains both spectral energy irradiance vales in 'column' `s.e.irrad` and spectral photon irradiance in 'column' `s.q.irrad`

9.6 Task: irradiances for more than one waveband

It is possible to calculate the irradiances for several wavebands with a single function call by supplying a `list` of `wavebands` as argument:

```
with(sun.data,
     photon_irradiance(w.length, s.e.irrad,
                       list(Red(), Green(), Blue()))) * 1e6

##   Red.ISO Green.ISO  Blue.ISO
##     452.2     220.2     149.0

Q.RGB <- with(sun.data,
     photon_irradiance(w.length, s.e.irrad,
                       list(Red(), Green(), Blue()))) * 1e6
signif(Q.RGB, 3)

##   Red.ISO Green.ISO  Blue.ISO
##       452       220       149

Q.RGB[1]

## Red.ISO
##   452.2

Q.RGB["Green.ISO"]

## Green.ISO
##     220.2
```

A named list can be used to override the use as names for the output of the waveband names:

```
with(sun.data,
     photon_irradiance(w.length, s.e.irrad,
                       list(R=Red(), G=Green(), B=Blue()))) * 1e6

##     R     G     B
## 452.2 220.2 149.0
```

Even when using a single waveband:

```
with(sun.data,
     photon_irradiance(w.length, s.e.irrad,
                       list(UVB=UVB()))) * 1e6

##   UVB
## 1.527
```

9.7 Task: use simple wavebands

Please, consult the packages' documentation for a list of predefined functions for creating wavebands. 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 49

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` function for `wavebands`. Functions available are `min`, `max`, `range`, `center_wl`, `labels`, and `color`.

```
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

labels(red)

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

color(red)

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

Here we demonstrate the use of an argument to choose a certain definition:

```
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
```

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

```
with(sun.data,
     photon_irradiance(w.length, s.e.irrad, UVB("ISO"))) * 1e6

## UVB.ISO
##   1.527

with(sun.data,
     photon_irradiance(w.length, s.e.irrad, UVB("none"))) * 1e6

## UVB.none
##    3.282
```

9.8 Task: define simple wavebands

Here we briefly introduce `new_waveband`, and only in chapter **??** we describe its use in full detail, including the use of spectral weighting functions (SWFs).

Defining a new `waveband` based on extreme wavelengths expressed in nm.

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

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

with(sun.data,
     photon_irradiance(w.length, s.e.irrad, wb1)) * 1e6

## range.500.600
##         314.1

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

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

with(sun.data,
     photon_irradiance(w.length, s.e.irrad, wb2)) * 1e6

## my.colour
##     314.1
```

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 read 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[2]:

$$\zeta = \frac{Q_{655\text{nm}<\lambda<665\text{nm}}}{Q_{725\text{nm}<\lambda<735\text{nm}}} \tag{9.4}$$

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

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

## [1] 1.251
```

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

or using the predefined convenience function `R_FR_ratio`:

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

## [1] 1.251
```

Using defaults for waveband definitions:

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

## [1] 0.00299
```

9.10 Task: energy ratios

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

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

## [1] 1.384
```

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:

$$\bar{q}' = \frac{Q_{\lambda_1<\lambda<\lambda_2}}{E_{\lambda_1<\lambda<\lambda_2}} \tag{9.5}$$

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

## [1] 4.544e-06
```

For obtaining the same quotient in $\mathrm{\mu mol\,J^{-1}}$ we just need to multiply by 10^6. We can use such a multiplier to convert E [$\mathrm{W\,m^{-2}}$] into Q [$\mathrm{\mu mol\,m^{-2}\,s^{-1}}$] (as $\mathrm{W = J\,s^{-1}}$), or as a divisor to convert Q [$\mathrm{\mu mol\,m^{-2}\,s^{-1}}$] into E [$\mathrm{W\,m^{-2}}$], *for a given light source and waveband*:

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

## [1] 4.544
```

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. We can use the 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
##         69.63         68.53         58.54
```

Here we demonstrate that the sum of the four 'split' irradiances add to the total for the range of wavelengths covered:

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

## [1] 196.7

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

##   PAR
## 196.7
```

It also 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
##        0.3540        0.3484        0.2976
```

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
##         35.40         34.84         29.76
```

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
##         35.40         34.84         29.76
```

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

## range.400.500 range.500.600
##          50.4          49.6
```

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
##         196.7
```

9.13 Task: calculate the contribution of different regions of a spectrum to photon irradiance

The function `split_photon_irradiance` takes the same arguments as the equivalent function for photon irradiance, consequently only one code example is provided here (see section 9.12 for more details):

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

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

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(photobiologyVIS)
library(photobiologyUV)
```

10.2 Introduction

Weighted irradiance is usually reported in weighted energy units, but it is possible to also 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 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 use a biological spectral weighting function (BSWF). The focus of this chapter is on the differences between calculations for weighted irradiances compared to those for unweighted irradiances described in chapter 9. In particular it is important that you read sections 9.3, 9.4, **??**, and 9.6 before reading the present chapter.

10.3 Task: choosing the normalization wavelength

Function `GEN.G()` is predefined in package photobiologyUV as a convenience function for Green's formulation of Caldwell's generalized plant action spectrum (GPAS) **Green198x**

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

## GEN.G.300
##    0.1034
```

The code above uses the default normalization wavelength of 300 nm. Any arbitrary wavelength (nm), within the range of the waveband can be provided as an argument.

```
range(GEN.G())

## [1] 250.0 313.3

with(sun.data,
     energy_irradiance(w.length, s.e.irrad, GEN.G(280)))

## GEN.G.280
##   0.02402
```

10.4 Task: use weighted wavebands

Please, consult the packages' documentation for a list of predefined functions for creating weighted wavebands. Here we will present just a few examples of their use. We usually think of weighted irradiances as being defined by the weighting function, however, in many cases different normalizations are in use, and the result of any calculation depends very strongly on the wavelength used for normalization. For this reason, the functions provided accept an argument that can be used to select the normalization wavelength. In general, the default, is to use the most frequently used normalization.

In a few cases different mathematical formulations are available for the same spectrum, and the differences among them can be quite large. In such cases separate functions are provided for each of them (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.G(300)
```

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

GEN.G(280)

## GEN.G.280
## low (nm) 250
## high (nm) 313
## weighted SWF
## normalized at 280 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 seen above, there is a specialized `print` function for `wavebands`. Functions available are `min`, `max`, `range`, `midpoint`, `labels`, and `color`.

```
min(cie)

## [1] 250

max(cie)

## [1] 400

range(cie)

## [1] 250 400

midpoint(cie)

## [1] 325

normalization(cie)

## [1] 298

labels(cie)

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

color(cie)

## CIE98 CMF  CIE98 CC
## "#02000F" "#1A00DD"
```

10.5 Task: define wavebands

In section ?? we briefly introduced `new_waveband`, and here we describe its use in full detail, including the use of spectral weighting functions (SWFs).

Defining a new weighted `waveband`. We start with a simple 'toy' example:

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

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

with(sun.data,
     energy_irradiance(w.length, s.e.irrad, toy.wb))

##   TOY
## 241.7

with(sun.data,
     photon_irradiance(w.length, s.e.irrad, toy.wb))

##      TOY
## 0.001111
```

10.6 Introduction

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(photobiologyFilters)
```

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:

```
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:

```
A2T <- function(x) {10^-x}
```

They could be directly applied to spectra but doing this still gives "filter.spc" objects as a result, with the data labelled "T", when it has changed into absorbance. 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.dt <- copy(gg400.dt)
## Error:  object 'gg400.dt' not found
my.gg400.dt[ , A := T2A(Tfr)]
## Error:  object 'my.gg400.dt' not found
head(gg400.dt)
## Error:  object 'gg400.dt' not found
```

11.4 Task: spectral absorbance from spectral transmittance

11.5 Task: spectral transmittance from spectral absorbance

11.6 Task: absorbance from spectral asorbance and spectral irradiance

11.7 Task: reflected spectrum from spectral reflectance and spectral irradiance

11.8 Task: transmitted irradiance

11.9 Task: reflected radiance

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

11.11 Task: combined spectral transmittance of two or more filters

Ignoring reflectance

Considering reflectance

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

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

##     550 nm
## "#00FF00"

w_length2rgb(630) # red

##     630 nm
## "#FF0000"

w_length2rgb(380) # UVA

##     380 nm
## "#000000"

w_length2rgb(750) # far red

##     750 nm
## "#000000"

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

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

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)

## Warning:  Using only extreme wavelength values.

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

w_length_range2rgb(sun.data$w.length)

## Warning:  Using only extreme wavelength values.

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

w_length_range2rgb(550)

## Warning:  Calculating RGB values for monochromatic light.

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

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.data))

## [1] "#544F4B"

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

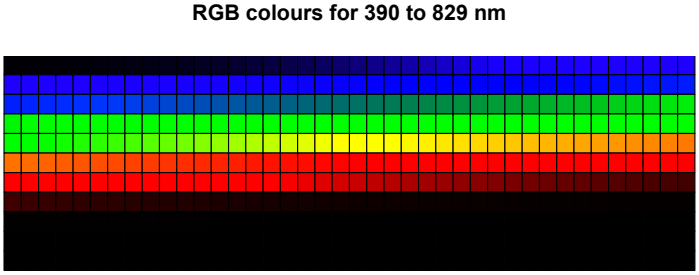
```
## [1] "#59534F"

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

## [1] "#B63C37"

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

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

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)])
}
```


RGB colours for 390 to 829 nm

CHAPTER 13

Photoreceptors

Abstract

In this chapter we explain how to .

13.1 Task:

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(photobiologyVIS)
library(photobiologyUV)
library(ggplot2)
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(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(photobiologyVIS)
library(photobiologyUV)
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

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.

16.1 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(photobiologyVIS)
library(photobiologyUV)
library(ggplot2)
library(ggtern)
library(photobiologygg)
library(gridExtra)

## Loading required package:  grid
```

16.2 Introduction to plotting spectra

We show in this chapter examples of how one can plot spectra. All the examples are done with package `ggplot2`, sometimes using in addition other packages. `ggplot2` provides the most recent type 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

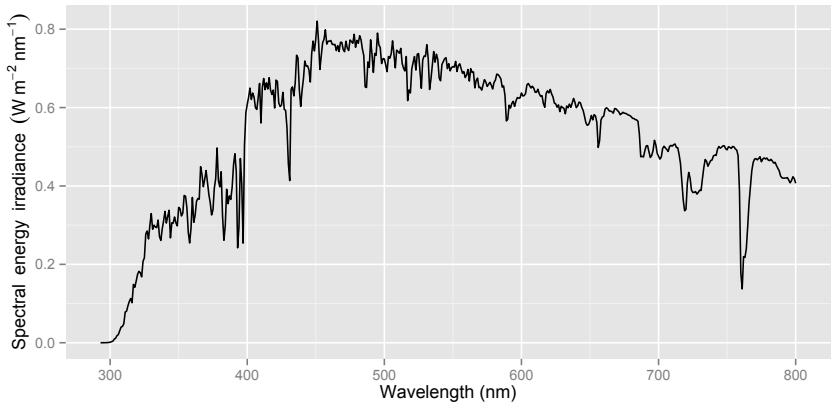
alternatives. Several of the functions used in this chapter are extensions to package `ggplot2`[1]

If you are not familiar with `ggplot2` plotting, please read Appendix C on page 157 before reading the present chapter.

16.3 Task: plotting spectra with `ggplot2`

We create a simple line plot, assign it a variable called `fig_sun.e` and then on the next line `print` it. We use `labs` to set nice axis labels.

```
fig_sun.e <-
  ggplot(data=sun.data, aes(x=w.length, y=s.e.irrad)) +
  geom_line()  +
  labs(
    y = expression(Spectral~~energy~~irradiance~~(W~m^{-2}~nm^{-1})),
    x = "Wavelength (nm)")
fig_sun.e
```

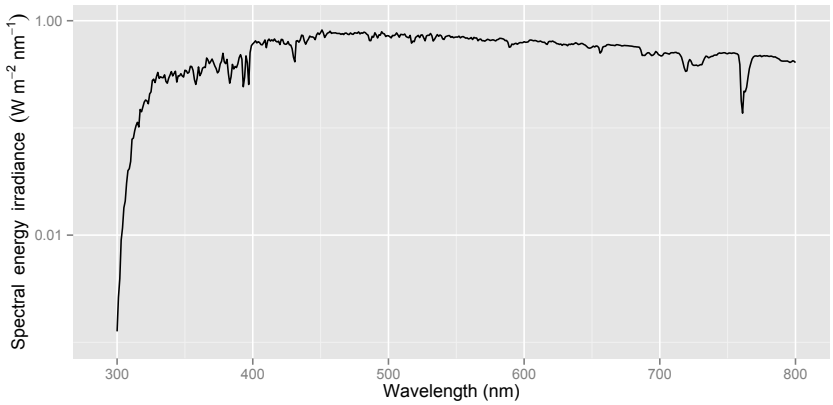


16.4 Task: using a log scale

Here without need to recreate the figure, we add a logarithmic scale for the y-axis and print on the fly the result. In this case we override the automatic limits of the scale.

```
fig_sun.e + scale_y_log10(limits=c(1e-3, 1e0))

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

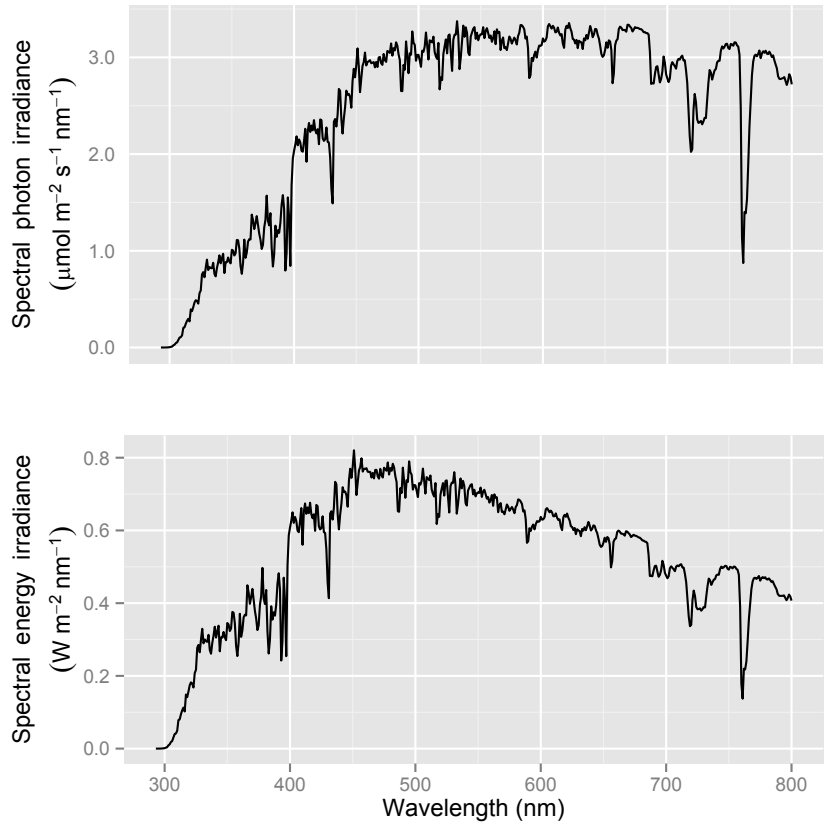
[1] `ggplot2` is feature-frozen. Consequently it is a good basis for adding application specific functionality through separate packages. `ggplot2` uses the *grammar of graphics* for describing the plots. This grammar, because it is consistent, tends to be easier to understand, and makes it easier to design new functionality that uses extensions based on the same 'language grammar' as used by the original package.



16.5 Task: compare energy and photon spectral units

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.

```
theme_stack_opts <-
  list(theme(axis.text.x = element_blank(),
             axis.ticks = element_blank(),
             axis.title.x = element_blank()))
num_one_dec <- function(x, ...)
  format(x, nsmall=1, trim=FALSE, width=4, ...)
fig_sun.q <-
  ggplot(data=sun.data, aes(x=w.length, y=s.q.irrad * 1e6)) +
  geom_line() +
  scale_y_continuous(labels = num_one_dec) +
  labs(y = expression(atop(Spectral~~photon~~irradiance,
                           (mu*mol~m^{-2}~s^{-1}~nm^{-1})))) +
  theme_stack_opts
fig_sun.e1 <- fig_sun.e +
  labs(y = expression(atop(Spectral~~energy~~irradiance,
                           (W~m^{-2}~nm^{-1}))),
       x = "Wavelength (nm)")
grid.arrange(fig_sun.q, fig_sun.e1, nrow=2)
```



To make sure that the widths of booth plots are the same, we need to make sure that the tick labels in both plots have the same format. For this we define a formatting function `num_one_dec` and then use it a the scale definition. We also add `atop` to the expression to set the spectral irradiance units on a second line in the axis label.

16.6 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)[2]. 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

[2]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.

y to spectral irradiance, while `label` is a character string with the wavelngth, possibly formatted.

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

##     x      y label
## 1 378 0.4970   378
## 2 416 0.6762   416
## 3 451 0.8205   451
## 4 478 0.7870   478
## 5 495 0.7900   495
## 6 531 0.7603   531

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

##     x      y label
## 1 416 0.6762   416
## 2 451 0.8205   451
## 3 478 0.7870   478
## 4 495 0.7900   495
## 5 531 0.7603   531
## 6 582 0.6854   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.data,
          get_peaks(w.length, s.e.irrad, span=21)))

##     x      y label
## 1 354 0.3759   354
## 2 366 0.4492   366
## 3 378 0.4970   378
## 4 416 0.6762   416
## 5 436 0.7337   436
## 6 451 0.8205   451

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

##     x      y label
## 1 451 0.8205   451
## 2 495 0.7900   495
## 3 747 0.5026   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.

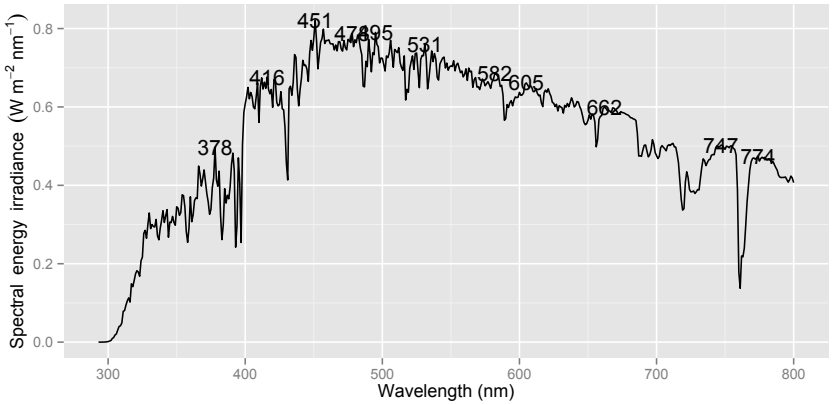
```
head(with(sun.data,
          get_valleys(w.length, s.e.irrad, span=51)))

##     x      y label
## 1 358 0.2545   358
## 2 393 0.2422   393
## 3 431 0.4137   431
## 4 487 0.6512   487
## 5 517 0.6177   517
## 6 589 0.5659   589

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

##     x      y label
## 1 431 0.4137   431
## 2 487 0.6512   487
## 3 517 0.6177   517
## 4 589 0.5659   589
## 5 656 0.4983   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 16.7.

16.7 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 16.6. We reuse once more `fig_sun.e` saved in section 16.3.

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

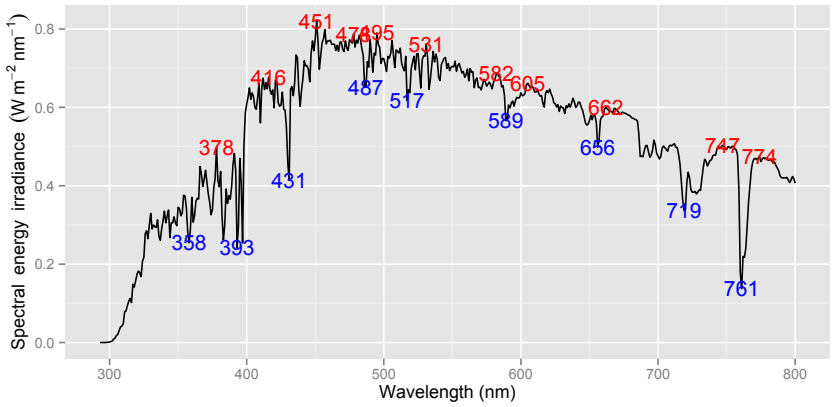


Now we play with `ggplot2` to show different ways of plotting the peaks and valleys. It behaves as a `ggplot2 stat_xxxx` function accepting a

`geom` argument and all the aesthetics valid for the chosen geom. By default `geom_text` is used.

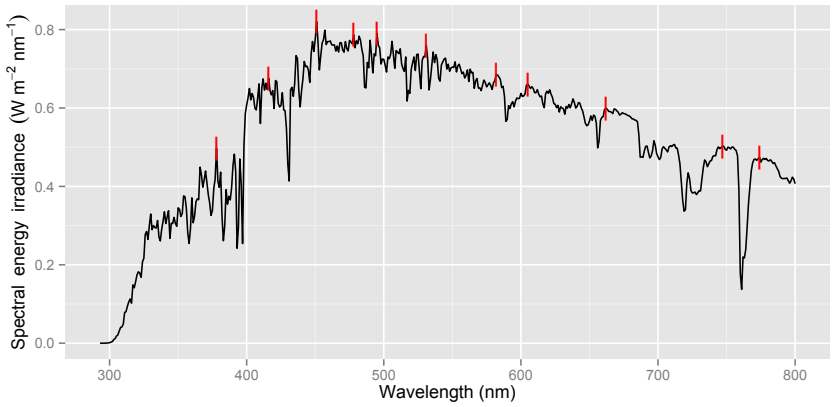
We can change aesthetics, for example the colour:

```
fig_sun.e + 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.

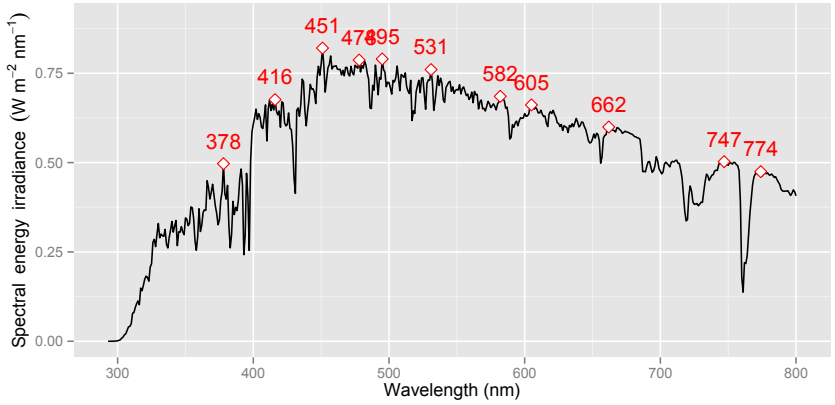
```
fig_sun.e +
  stat_peaks(colour="red", geom="point",
             shape="|", size=6, span=31)
```



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[3] In addition we expand the y-axis scale so that all labels fall within the plotting area.

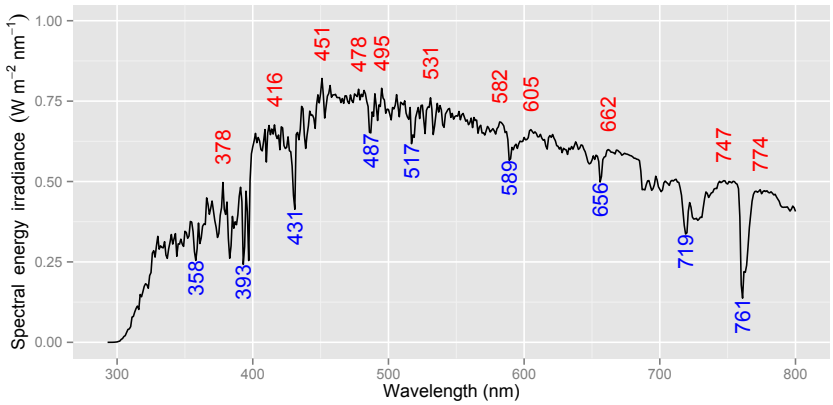
```
fig_sun.e +
  stat_peaks(colour="red", geom="point", shape=23,
             fill="white", size=3, span=31) +
  stat_peaks(colour="red", vjust=-1, span=31) +
  expand_limits(y=0.9)
```



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.e +
  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)
```

[3]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.



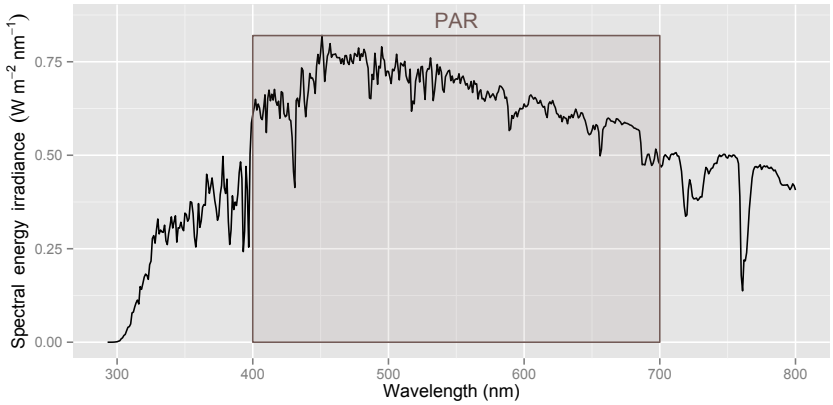
See section ?? in chapter 14 for an example these stats together with facets.

16.8 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`.

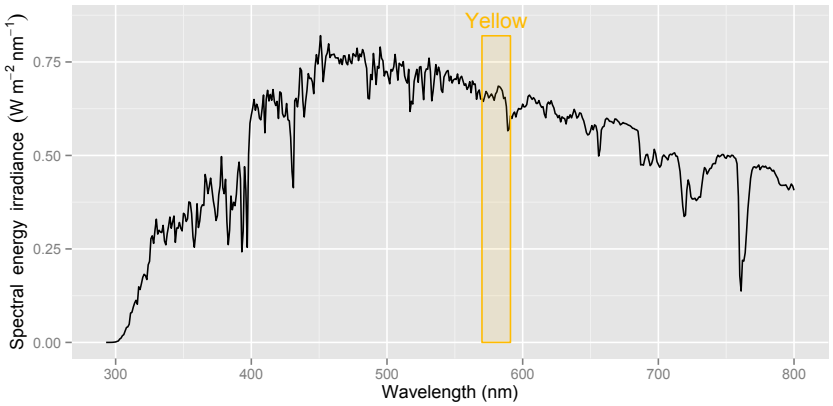
Here is an example for PAR using defaults, and with arguments supplied only for parameters with no defaults. The example does the annotation using two different 'geoms', `"rect"` for marking the region, and `"text"` for the labels.

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



This example annotates a narrow waveband.

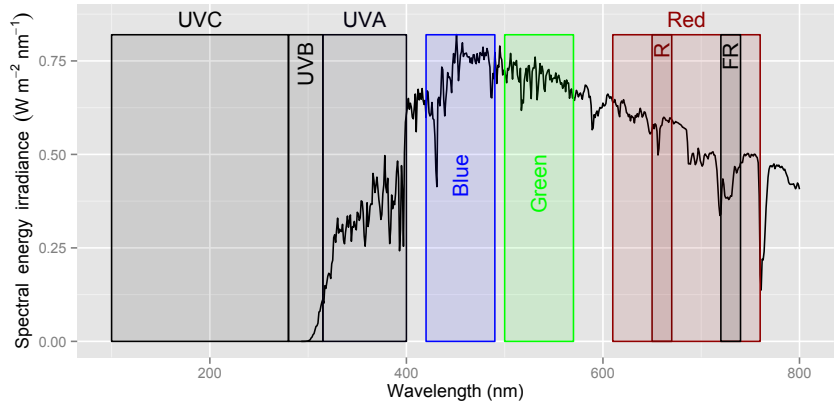
```
figvl <- fig_sun.e + annotate_waveband(Yellow(), "rect", ymax=0.82) +
                     annotate_waveband(Yellow(), "text", y=0.86)
figvl
```



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.

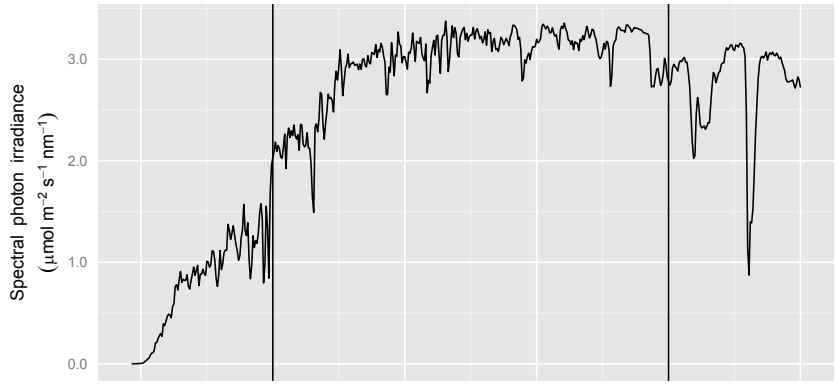
```
figv2 <- fig_sun.e +
  annotate_waveband(UVC(), "rect",
                    ymax=0.82) +
  annotate_waveband(UVC(), "text",
                    y=0.86) +
  annotate_waveband(UVB(), "rect",
                    ymax=0.82) +
  annotate_waveband(UVB(), "text",
                    y=0.80, angle=90, hjust=1)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



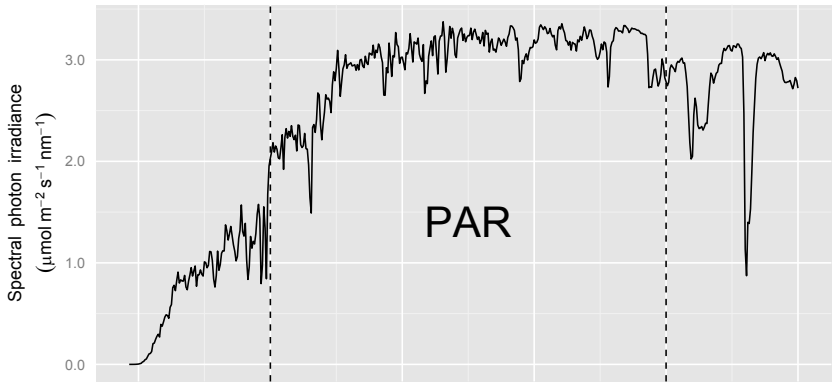
A simple example using `geom_vline`:

```
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
```

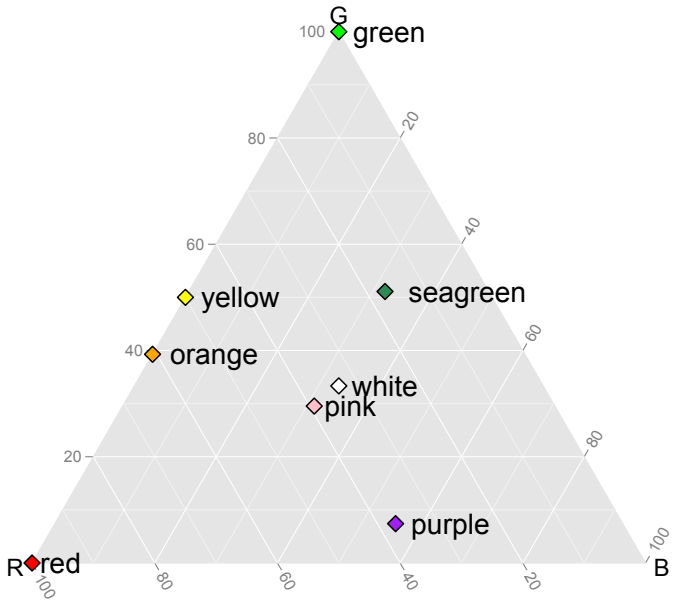


16.9 Task: plotting colours in Maxwell's triangle

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
```



16.10 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 λ 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.

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:

CHAPTER 20

Optimizing performance

Abstract

In this chapter we explain how to make your photobiology calculations execute as fast as possible. The code has been profiled and the performance bottlenecks removed in most cases the implementing some functions in C++. Furthermore copying of spectra is minimized by using package `data.table` as the base class of all objects where spectral data is stored. However, it is possible to improve performance even more by changing some defaults and writing efficient user code. This is what is discussed in the present chapter, and should not be of concern unless several thousands of spectra need to be processed.

20.1 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(photobiologyVIS)
library(photobiologyUV)
library(microbenchmark)
```

Although not a recommended practice, just to keep the examples shorter, we `attach` a data set for the solar spectrum:

```
attach(sun.data)

## The following objects are masked _by_ .GlobalEnv:
##
##     s.e.irrad, s.q.irrad
```

20.2 Introduction

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

There is also a lot you can achieve by carefully writing the code in your own scripts. The packages themselves are fairly well optimized for speed. In your own code try to avoid unnecessary copying of big objects. The r4photobiology suite makes extensive use of the `data.table` package, using it also in your own code could help. Try to avoid use of explicit loops by replacing them with vectorized operations, and when sequentially building vectors in a loop, preallocate an object big enough before entering the loop.

Being R an interpreted language, there is rather little automatic code optimization taking place, so you may find that even simple things like moving invariant calculations out of loops, and avoiding repeated calculations of the same value by storing the value in a variable can improve performance.

This type of 'good style' optimizations have been done throughout the suite's code, and more specific problem identified by profiling and and dealt with case by case. Of course, to achieve maximum overall performance, to should follow the same approach with your own code.

20.3 Task: avoiding repeated validation

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

20.4 Task: caching of multipliers

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

20.5 Task: benchmarking

You can use the package `microbenchmark` to time the code and find the parts that slow it down. I have used it, and also I have used profiling to optimize the code for speed. The examples below show how choosing different values from the defaults can speed up calculations when the same calculations are done repeatedly on spectra measured at exactly the same wavelengths, something which is usual when analyzing spectra measured with the same instrument. The choice of defaults is based on what is best when processing a moderate number of spectra, say less than a few hundreds, as opposed to many thousands.

```
library(microbenchmark)
```

Convenience functions

The convenience functions are slightly slower than the generic `irradiance` function.

```
res1 <- microbenchmark(
  photon_irradiance(w.length, s.e.irrad, PAR(), use.cache=TRUE),
  times=100L, control=list(warmup = 10L))
res2 <- microbenchmark(
  irradiance(w.length, s.e.irrad, PAR(), unit.out="photon",
             use.cache=TRUE),
  times=100L, control=list(warmup = 10L))
```

Using the generic reduces the median execution time from 0.104 ms to 0.104 ms, by 4.8×10^{-4}% if using the cache.

Using cached multipliers

Using the cache when repeatedly applying the same waveband has a large impact on the execution time.

```
res1 <- microbenchmark(
  photon_irradiance(w.length, s.e.irrad, PAR()),
  times=100L, control=list(warmup = 10L))
res2 <- microbenchmark(
  photon_irradiance(w.length, s.e.irrad, PAR(), use.cache=TRUE),
  times=100L, control=list(warmup = 10L))
```

When using an unweighted waveband the cache reduces the median execution time from 0.181 ms to 0.169 ms, by 6.9%.

When using BSWFs the speed up by use of the cache is more important, and dependent on the complexity of the equation used in the calculation.

```
res1 <- microbenchmark(
  energy_irradiance(w.length, s.e.irrad, CIE()),
  times=100L, control=list(warmup = 10L))
res2 <- microbenchmark(
  energy_irradiance(w.length, s.e.irrad, CIE(), use.cache=TRUE),
  times=100L, control=list(warmup = 10L))
```

When using a weighted waveband, in this example, `CIE()`, the cache reduces the median execution time from 0.322 ms to 0.124 ms, by 61%.

Disabling checks

Disabling the checking of the spectrum halves once again the execution time for unweighted wavebands.

```
res1 <- microbenchmark(
  photon_irradiance(w.length, s.e.irrad, PAR(), use.cache=TRUE),
  times=100L, control=list(warmup = 10L))
res2 <- microbenchmark(
  photon_irradiance(w.length, s.e.irrad, PAR(), use.cache=TRUE,
                    check.spectrum=FALSE),
  times=100L, control=list(warmup = 10L))
```

When using an unweighted waveband, in this example, `PAR()`, the disabling the data validation checking reduces the median execution time from 0.105 ms to 0.076 ms, by 27%.

Using stored wavebands

Saving a waveband object and reusing it, can give an additional speed up when all other optimizations are also used.

```
myPAR <- PAR()
res1 <- microbenchmark(
  photon_irradiance(w.length, s.e.irrad, PAR(), use.cache=TRUE,
                    check.spectrum=FALSE),
  times=100L, control=list(warmup = 10L))
res2 <- microbenchmark(
  photon_irradiance(w.length, s.e.irrad, myPAR, use.cache=TRUE,
                    check.spectrum=FALSE),
  times=100L, control=list(warmup = 10L))
```

When using an unweighted waveband, in this example, `PAR()`, using a saved waveband object reduces the median execution time from 0.0787 ms to 0.0688 ms, by 13%.

Saving a waveband object that uses weighting and reusing it, gives an additional speed up when all other optimizations are also used.

```
myCIE <- CIE()
res1 <- microbenchmark(
  photon_irradiance(w.length, s.e.irrad, CIE(), use.cache=TRUE,
                    check.spectrum=FALSE),
  times=100L, control=list(warmup = 10L))
res2 <- microbenchmark(
  photon_irradiance(w.length, s.e.irrad, myCIE, use.cache=TRUE,
                    check.spectrum=FALSE),
  times=100L, control=list(warmup = 10L))
```

When using a weighted waveband, in this example, `CIE()`, using a saved waveband object reduces the median execution time from 0.0848 ms to 0.0699 ms, by 17%.

Inserting hinges

Inserting 'hinges' to reduce integration errors slows down the computations considerably. If the spectral data is measured with a small wavelength step, the errors are rather small. By default the use of 'hinges' is automatically decided based on the average wavelength step in the spectral data. The 'cost' of using hinges depends on the waveband definition, as BSWFs with discontinuities in the slope require several hinges, while unweighted one requires at most two, one at each boundary.

```
res1 <- microbenchmark(
  energy_irradiance(w.length, s.e.irrad, PAR(), use.cache=TRUE),
  times=100L, control=list(warmup = 10L))
res2 <- microbenchmark(
  energy_irradiance(w.length, s.e.irrad, PAR(), use.cache=TRUE,
                    use.hinges=TRUE),
  times=100L, control=list(warmup = 10L))
```

When using an uweighted waveband, in this example, `PAR()`, enabling use of hinges increases the median execution time from 0.103 ms to 0.421 ms, by a factor of 4.068.

Inserting 'hinges' to reduce integration errors slows down the computations a lot. If the spectral data is measured with a small wavelength step, the errors are rather small. By default the use of 'hinges' is automatically decided based on the average wavelength step in the spectral data.

```
res1 <- microbenchmark(
  energy_irradiance(w.length, s.e.irrad, CIE(), use.cache=TRUE),
  times=100L, control=list(warmup = 10L))
res2 <- microbenchmark(
  energy_irradiance(w.length, s.e.irrad, CIE(), use.cache=TRUE,
                    use.hinges=TRUE),
  times=100L, control=list(warmup = 10L))
```

When using an weighted waveband, in this example, `CIE()`, enabling use of hinges increases the median execution time from 0.112 ms to 0.421 ms, by a factor of 3.7589.

20.6 Overall speed-up achievable

GEN.G

If we consider a slow computation, using a BSWF with a complex equation like GEN.G, we can check the best case improvement in throughput that can be —on a given hardware and software system.

```
# slowest
res1 <- microbenchmark(
  photon_irradiance(w.length, s.e.irrad, GEN.G(),
                    use.cache=FALSE,
                    use.hinges=TRUE,
                    check.spectrum=TRUE),
                      times=100L, control=list(warmup = 10L))
```

```
# default
res2 <- microbenchmark(
  photon_irradiance(w.length, s.e.irrad, GEN.G()),
                        times=100L, control=list(warmup = 10L))

# fastest
gen.g <- GEN.G()
res3 <- microbenchmark(
  irradiance(w.length, s.e.irrad, gen.g,
             use.cache=TRUE,
             use.hinges=FALSE,
             check.spectrum=FALSE,
             unit.out="photon"),
                        times=100L, control=list(warmup = 10L))
```

When using a weighted waveband, in this example, `GEN.G()`, enabling all checks and optimizations for precision, and disabling all optimizations for speed yields a median execution time of 0.596 ms, accepting all defaults yields a median execution time 0.279 ms, and disabling all checks, optimizations for precision and enabling all optimizations for speed yields a median execution time of 0.0646, in relation to the slowest one, execution times are 100, 47, and 11%.

Finally we compare the returned values for the irradiance, to see the impact on them of optimizing for speed.

```
# slowest
photon_irradiance(w.length, s.e.irrad, GEN.G(),
                  use.cache=FALSE,
                  use.hinges=TRUE,
                  check.spectrum=TRUE)

## GEN.G.300
## 2.579e-07

# default
photon_irradiance(w.length, s.e.irrad, GEN.G())

## GEN.G.300
## 2.592e-07

# fastest
gen.g <- GEN.G()
irradiance(w.length, s.e.irrad, gen.g,
           use.cache=TRUE,
           use.hinges=FALSE,
           check.spectrum=FALSE,
           unit.out="photon")

## GEN.G.300
## 2.592e-07
```

These results are based on spectral data at 1 nm interval, for more densely measured data the effect of not using hinges becomes even smaller. In contrast, with data measured at wider wavelength steps, the errors will be larger. They also depend on the specific BSWF being used.

CIE

If we consider a slow computation, using a BSWF with a complex equation like CIE, we can check the best case improvement in throughput that can be —on a given hardware and software system.

```
# slowest
res1 <- microbenchmark(
  photon_irradiance(w.length, s.e.irrad, CIE(),
                    use.cache=FALSE,
                    use.hinges=TRUE,
                    check.spectrum=TRUE),
                      times=100L, control=list(warmup = 10L))
# default
res2 <- microbenchmark(
  photon_irradiance(w.length, s.e.irrad, CIE()),
                       times=100L, control=list(warmup = 10L))

# fastest
cie <- CIE()
res3 <- microbenchmark(
  irradiance(w.length, s.e.irrad, cie,
             use.cache=TRUE,
             use.hinges=FALSE,
             check.spectrum=FALSE,
             unit.out="photon"),
                       times=100L, control=list(warmup = 10L))
```

When using a weighted waveband, in this example, `CIE()`, enabling all checks and optimizations for precision, and disabling all optimizations for speed yields a median execution time of 0.648 ms, accepting all defaults yields a median execution time 0.325 ms, and disabling all checks, optimizations for precision and enabling all optimizations for speed yields a median execution time of 0.0646, in relation to the slowest one, execution times are 100, 50, and 10%.

Finally we compare the returned values for the irradiance, to see the impact on them of optimizing for speed.

```
# slowest
photon_irradiance(w.length, s.e.irrad, CIE(),
                  use.cache=FALSE,
                  use.hinges=TRUE,
                  check.spectrum=TRUE)

## CIE98.298
## 2.038e-07

# default
photon_irradiance(w.length, s.e.irrad, CIE())

## CIE98.298
## 2.037e-07

# fastest
CIE <- CIE()
irradiance(w.length, s.e.irrad, CIE,
           use.cache=TRUE,
```

```
            use.hinges=FALSE,
            check.spectrum=FALSE,
            unit.out="photon")

## CIE98.298
## 2.037e-07
```

These results are based on spectral data at 1 nm interval, for more densely measured data the effect of not using hinges becomes even smaller. In contrast, with data measured at wider wavelength steps, the errors will be larger. They also depend on the specific BSWF being used.

Using `split_irradiance`

Using the cache also helps with `split_irradiance`.

```
res1 <- microbenchmark(
  split_photon_irradiance(w.length, s.e.irrad,
                          c(400, 500, 600, 700)),
  times=100L, control=list(warmup = 10L))
res2 <- microbenchmark(
  split_photon_irradiance(w.length, s.e.irrad,
                          c(400, 500, 600, 700),
                          use.cache=TRUE),
  times=100L, control=list(warmup = 10L))
```

When using `split_irradiance`, the cache reduces the median execution time from 0.622 ms to 0.425 ms, by 32%.

Using hinges slows down calculations:

```
res1 <- microbenchmark(
  split_photon_irradiance(w.length, s.e.irrad,
                          c(400, 500, 600, 700),
                          use.cache=TRUE),
  times=100L, control=list(warmup = 10L))
res2 <- microbenchmark(
  split_photon_irradiance(w.length, s.e.irrad,
                          c(400, 500, 600, 700),
                          use.cache=TRUE,
                          use.hinges=TRUE),
  times=100L, control=list(warmup = 10L))
```

When using `split_irradiance`, enabling use of hinges increases the median execution time from 0.426 ms to 0.77 ms, by a factor of 1.8056. There is less overhead than if calculating the same three wavebands separately, as all hinges are inserted in a single operation.

Disabling checking of the spectrum reduces the execution time, but proportionally not as much as for the `irradiance` functions, as the spectrum is checked only once independently of the number of bands into which it is split.

```
res1 <- microbenchmark(
  split_photon_irradiance(w.length, s.e.irrad,
                          c(400, 500, 600, 700),
                          use.cache=TRUE),
```

```
  times=100L, control=list(warmup = 10L))
res2 <- microbenchmark(
  split_photon_irradiance(w.length, s.e.irrad,
                          c(400, 500, 600, 700),
                          use.cache=TRUE,
                          check.spectrum=FALSE),
  times=100L, control=list(warmup = 10L))
```

When using `split_irradiance`, disabling the data validation check reduces the median execution time from 0.423 ms to 0.398 ms, by 5.9%.

As all the execution times are in milliseconds, all the optimizations discussed above are totally irrelevant unless you are planning to repeat similar calculations on thousands of spectra. They apply only to the machine, OS and version of R and packages used when building this typeset output.

20.7 Profiling

Profiling is basically fine-grained benchmarking. It provides information about in which part of your code the program spends most time when executing. Once you know this, you can try to just make those critical sections execute faster. Speed-ups can be obtained either by rewriting these parts in a compiled language like C or C++, or by use of a more efficient calculation algorithm. A detailed discussion is outside the scope of this handbook, so only a brief example will be shown here.

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

Part III

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] 3.142

print(pi, digits=22)

## [1] 3.141592653589793115998

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

## [1] 1.225e-16

log(100)

## [1] 4.605

log10(100)

## [1] 2

log2(8)

## [1] 3

exp(1)

## [1] 2.718
```

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:  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)
z

## Error: object 'z' not found

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] 0.3333
```

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 112.

```
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] 1.225e-16

sin(2 * pi)

## [1] -2.449e-16
```

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

## [1] 2.22e-16

.Machine$double.neg.eps

## [1] 1.11e-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"
h + 2

## Error:  non-numeric argument to binary operator
```

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 Type conversions

The least intuitive ones are those related to logical values. All others are as one would expect.

```
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

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.7 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.8 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] 5.916

range(x)

## [1]  1 20

max(x)
```

```
## [1] 20

min(x)

## [1] 1

length(x)

## [1] 20
```

A.9 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 UV`Mean` 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] 1.797

SEM(a)

## [1] 1.797

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] 1.797

SEM(a)

## [1] 1.797

SEM(a.na)

## [1] 1.797
```

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] 1.797

SEM(a.na)

## [1] NA

SEM(a.na, TRUE)
```

```
## [1] 1.797

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

## [1] 1.797

SEM(TRUE, a.na)

## Warning:  the condition has length > 1 and only the first
element will be used

## [1] NA

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

## [1] 1.797
```

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

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.

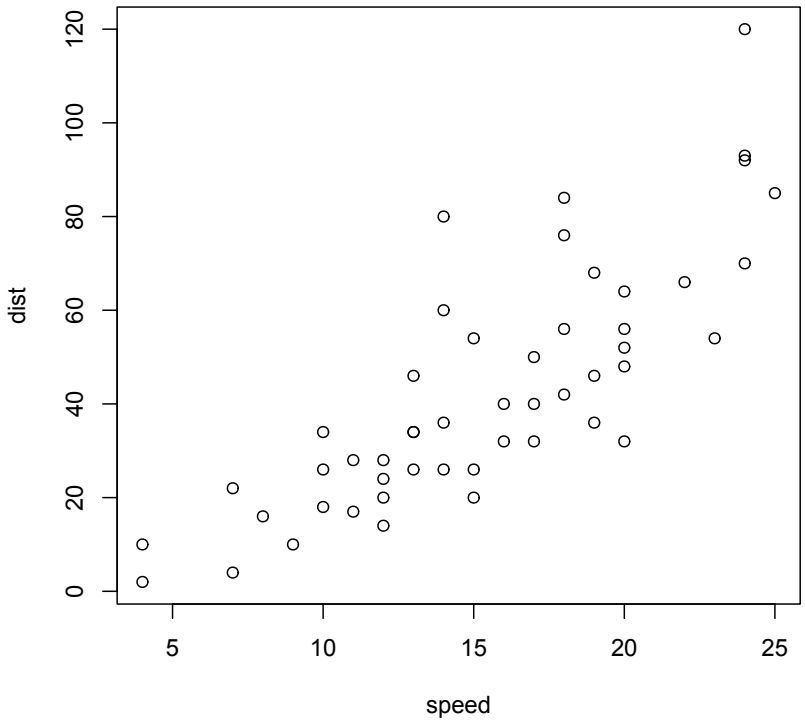
Before we can plot anything, we need some data.

```
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)
```



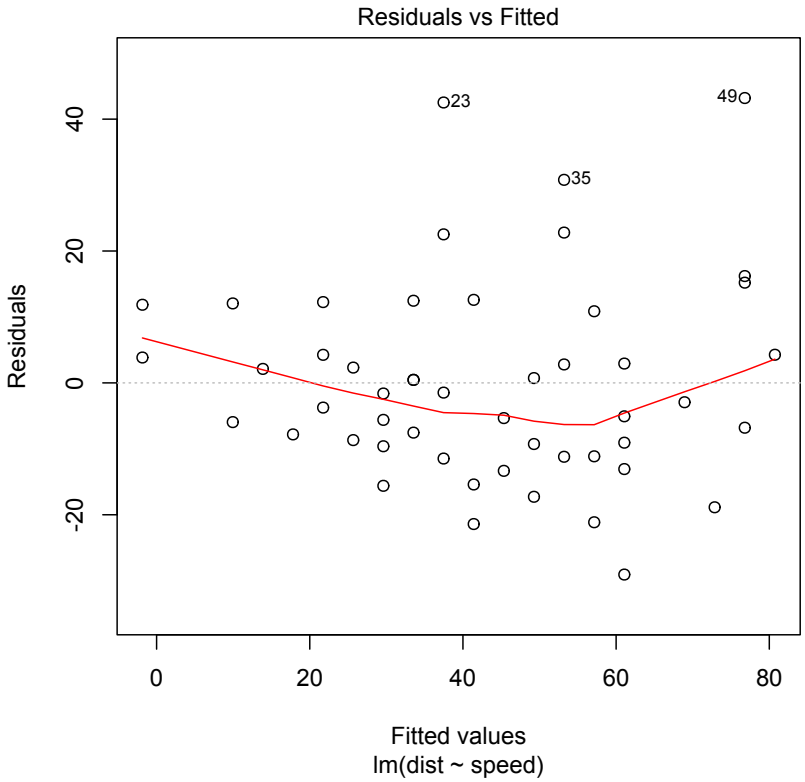
Fitting linear models

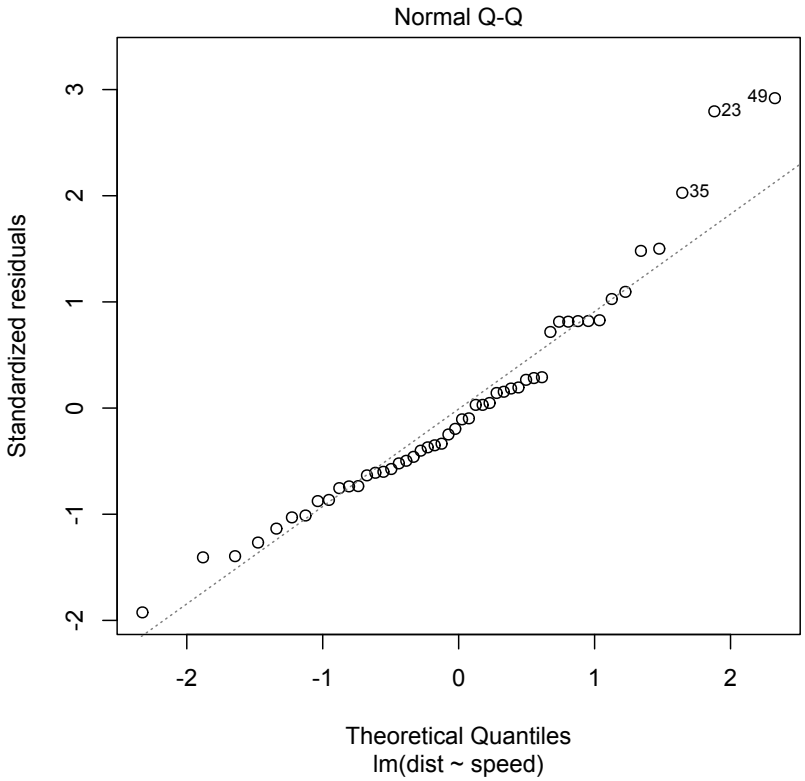
Regression

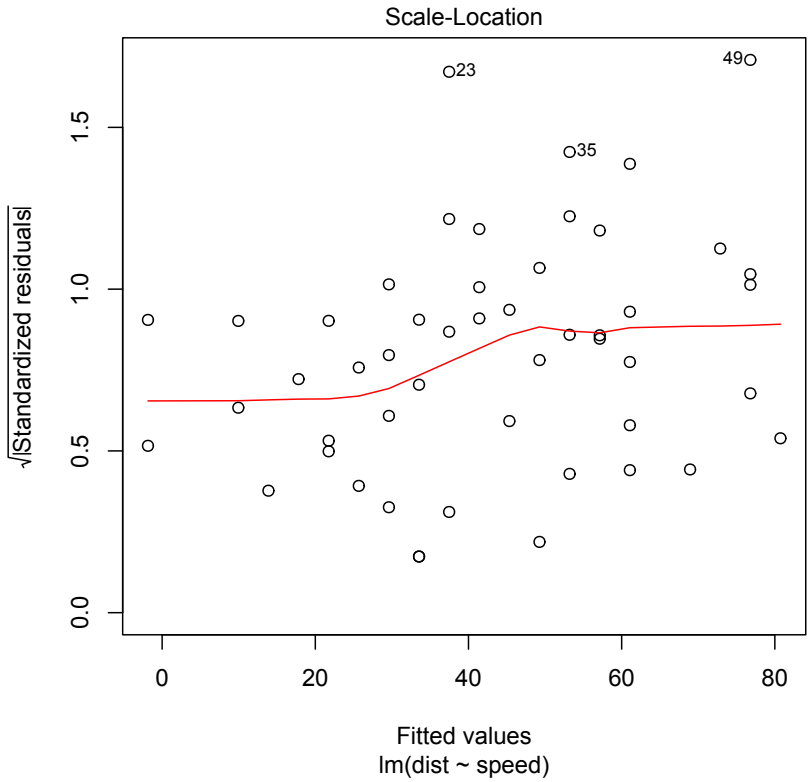
The R function `lm` is used next to fit a linear regression.

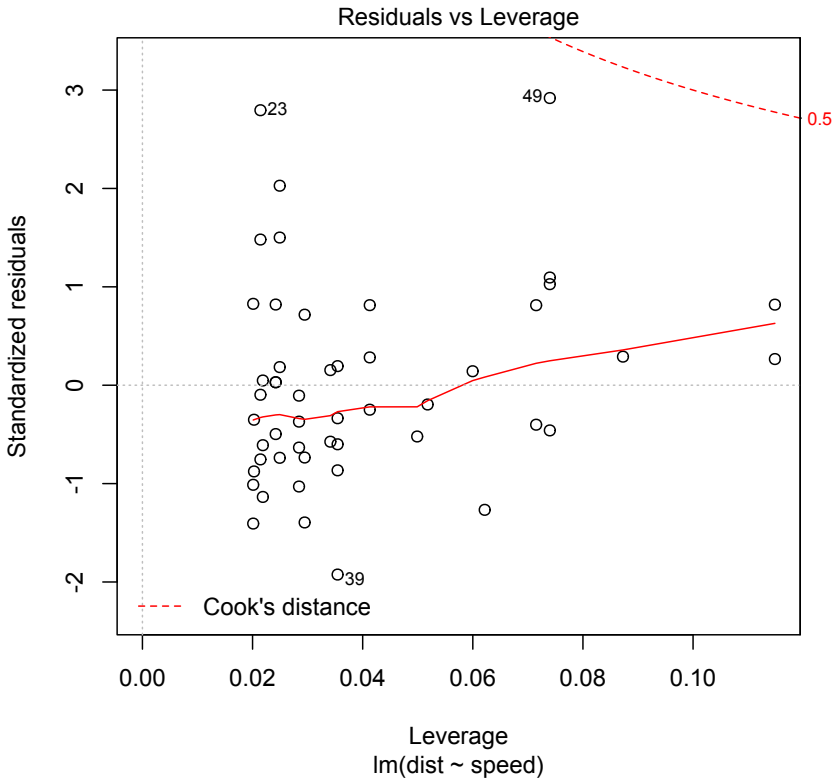
```
fm1 <- lm(dist ~ speed, data=cars) # we fit a model, and then save the result
plot(fm1) # we produce diagnosis plots
summary(fm1) # we inspect the results from the fit

##
```

```
## Call:
## lm(formula = dist ~ speed, data = cars)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -29.07  -9.53  -2.27   9.21  43.20
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  -17.579      6.758   -2.60    0.012
## speed          3.932      0.416    9.46  1.5e-12
##
## (Intercept) *
## speed       ***
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 15.4 on 48 degrees of freedom
## Multiple R-squared:  0.651,Adjusted R-squared:  0.644
## F-statistic: 89.6 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  21185   21185    89.6 1.5e-12 ***
## Residuals 48  11354     237
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```


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


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

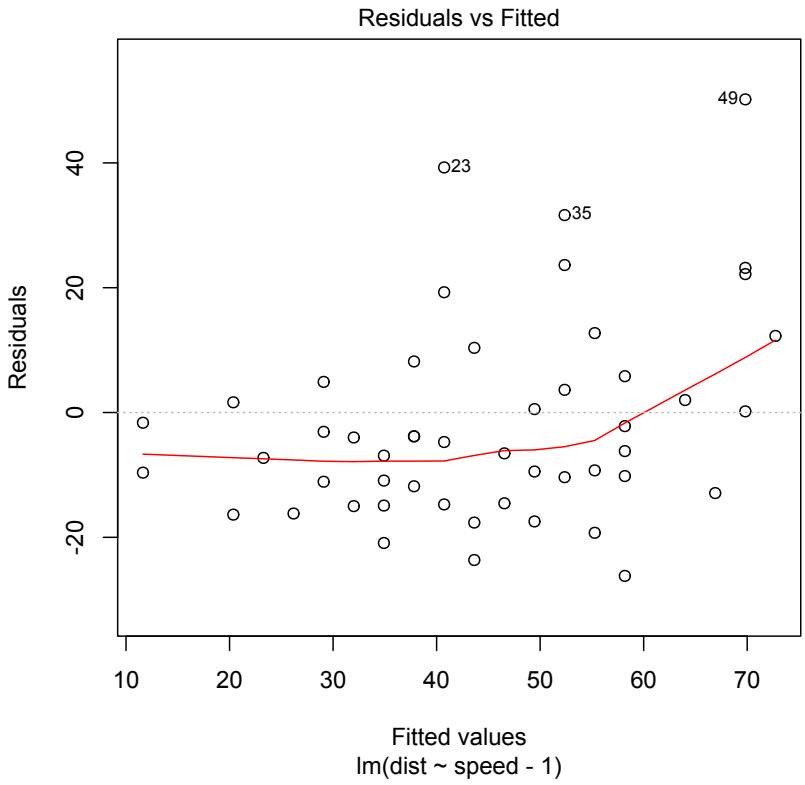

Scale-Location
√|Standardized residuals|
0.0
0.5
1.0
1.5
0
20
40
60
80
23
35
49
Fitted values
lm(dist ~ speed)

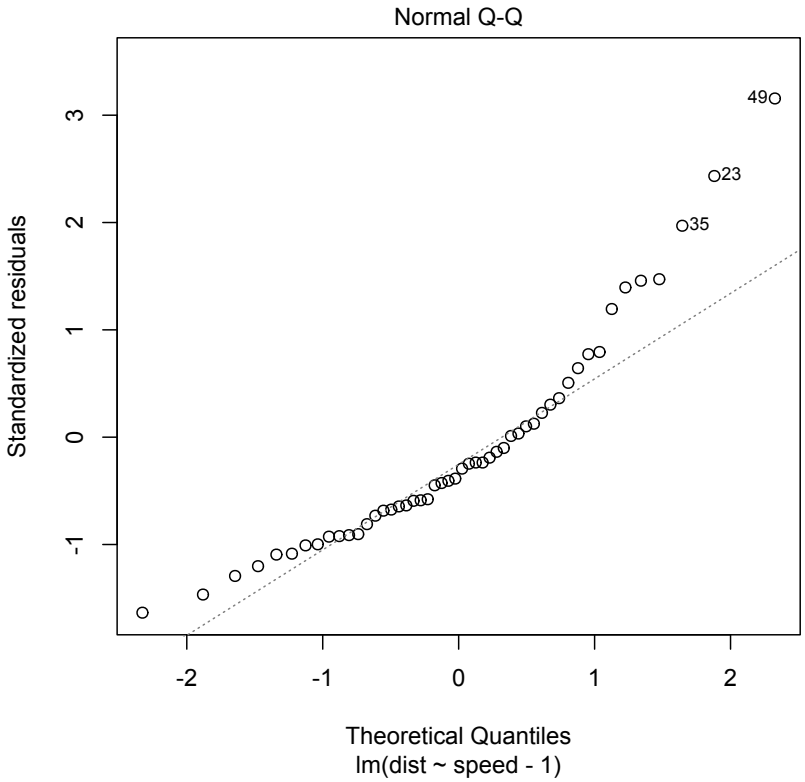


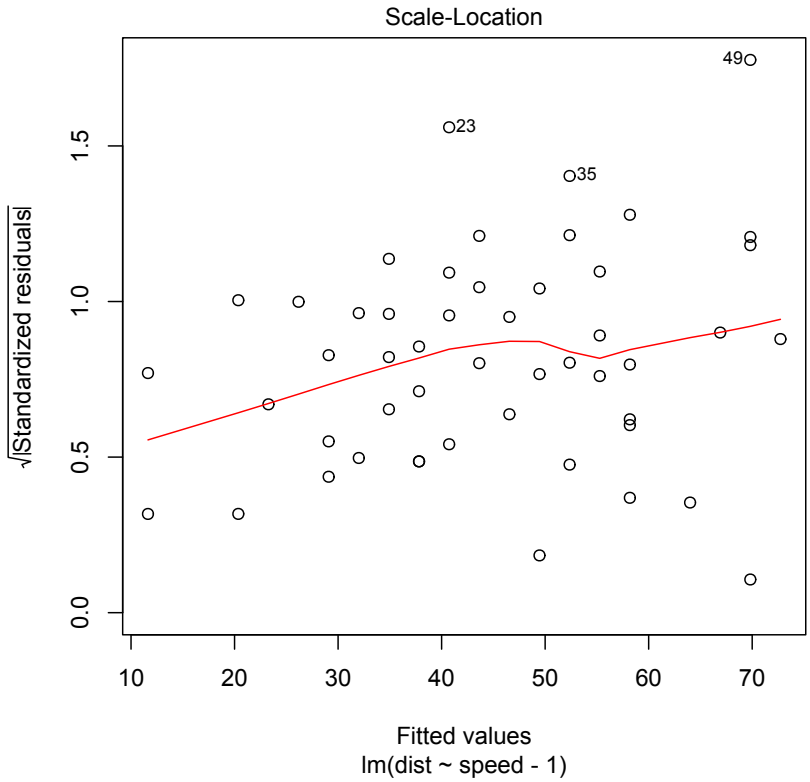
Let's look at each step separately: `dist ~ speed` is the specification of the model to be fitted. The intercept is always implicitly included. To 'remove' this implicit intercept from the earlier model we can use `dist ~ speed - 1`.

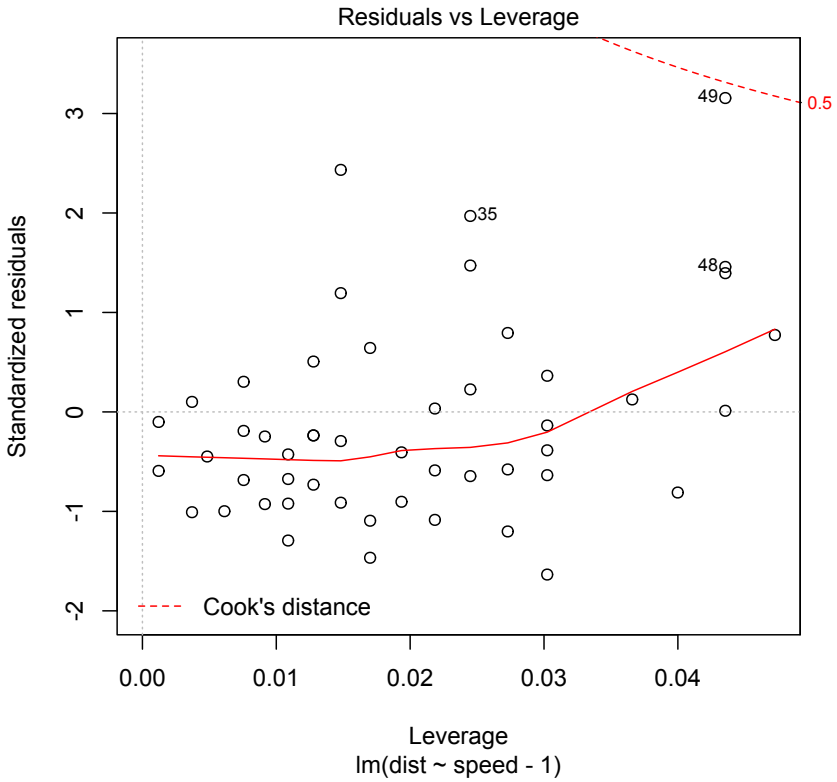
```
fm2 <- lm(dist ~ speed - 1, data=cars) # we fit a model, and then save the result
plot(fm2) # we produce diagnosis plots
summary(fm2) # we inspect the results from the fit

##
## Call:
## lm(formula = dist ~ speed - 1, data = cars)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -26.18 -12.64  -5.46   4.59  50.18
##
## Coefficients:
##       Estimate Std. Error t value Pr(>|t|)
## speed    2.909      0.141    20.6   <2e-16 ***
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 16.3 on 49 degrees of freedom
## Multiple R-squared:  0.896,Adjusted R-squared:  0.894
```

```
## F-statistic:  423 on 1 and 49 DF,  p-value: <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 <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
-2
-1
0
1
2
-1
0
1
2
3

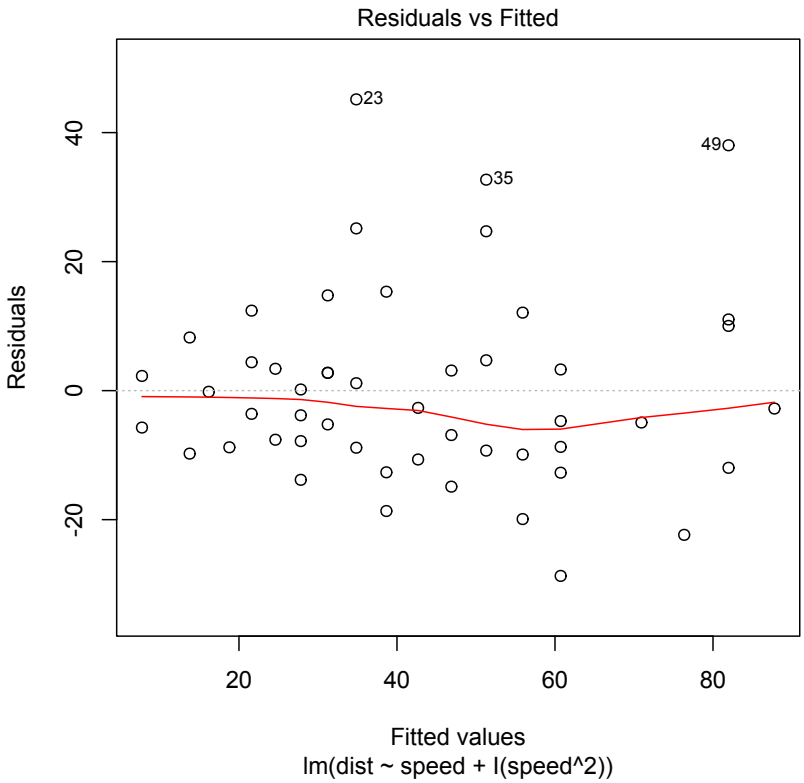

Scale-Location
49
23
35
√|Standardized residuals|
0.0
0.5
1.0
1.5
10
20
30
40
50
60
70
Fitted values
lm(dist ~ speed - 1)

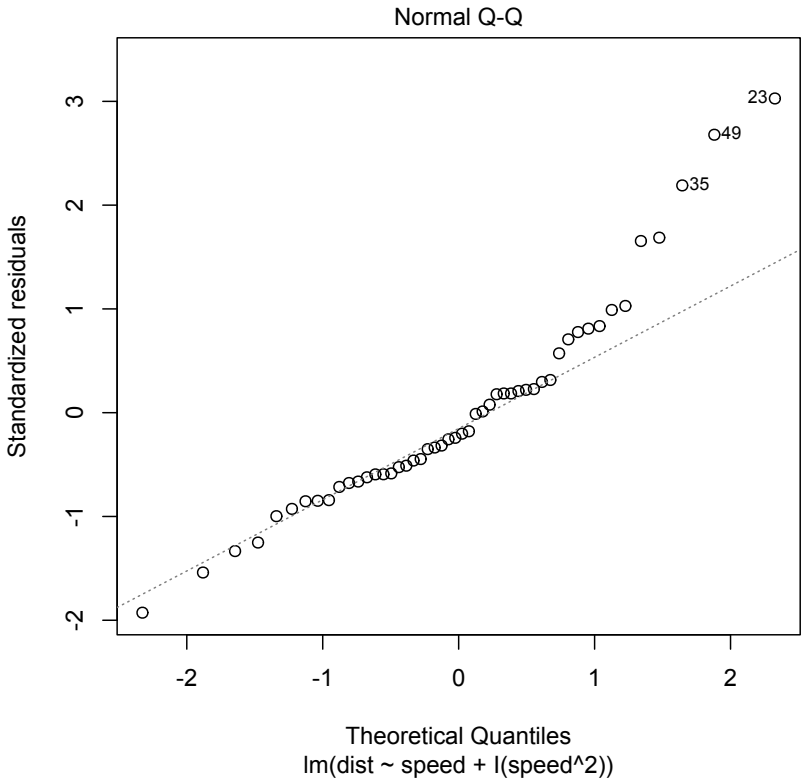


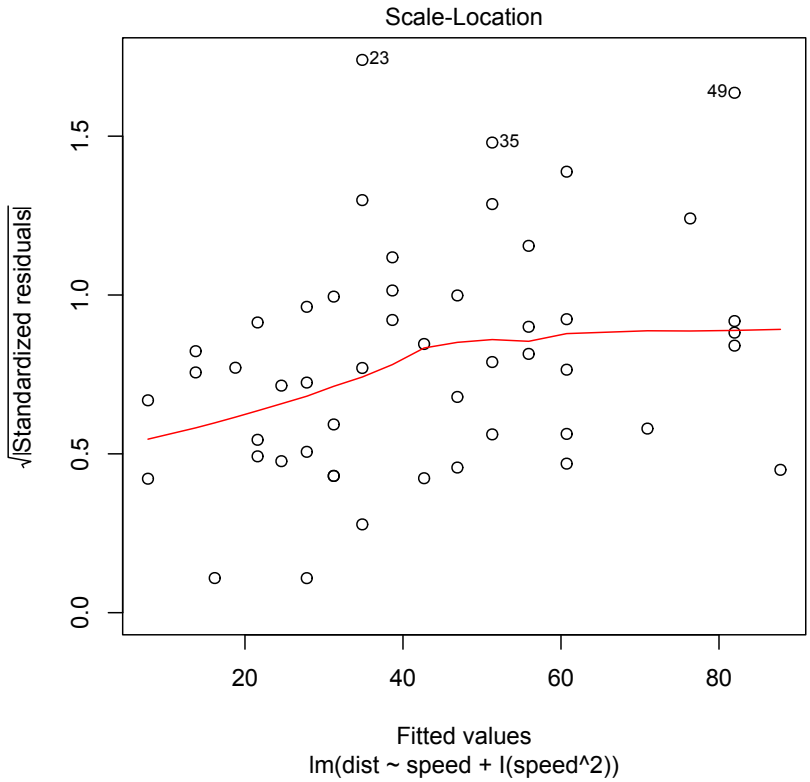
We now we fit a second degree polynomial.

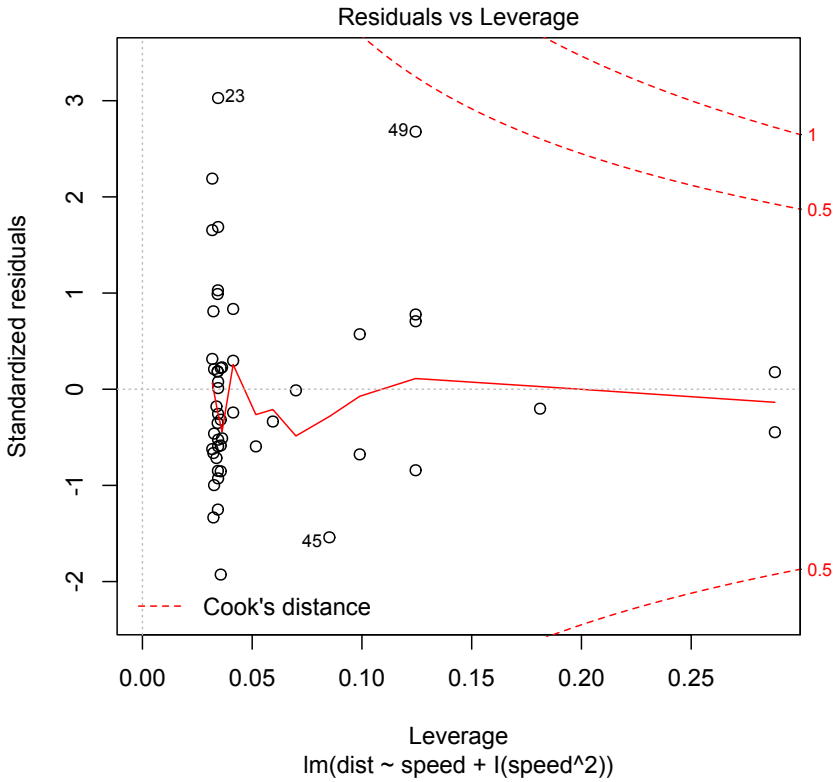
```
fm3 <- lm(dist ~ speed + I(speed^2), data=cars) # we fit a model, and then save the result
plot(fm3) # we produce diagnosis plots
summary(fm3) # we inspect the results from the fit

##
## Call:
## lm(formula = dist ~ speed + I(speed^2), data = cars)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -28.72  -9.18  -3.19   4.63  45.15
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)    2.470     14.817    0.17     0.87
## speed          0.913      2.034    0.45     0.66
## I(speed^2)     0.100      0.066    1.52     0.14
##
## Residual standard error: 15.2 on 47 degrees of freedom
## Multiple R-squared:  0.667,Adjusted R-squared:  0.653
## F-statistic: 47.1 on 2 and 47 DF,  p-value: 5.85e-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   21185    92.0 1.2e-12 ***
## I(speed^2)   1    529     529     2.3    0.14
## Residuals   47  10825     230
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```




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


Scale-Location
23
49
35
√|Standardized residuals|
0.0
0.5
1.0
1.5
20
40
60
80
Fitted values
lm(dist ~ speed + I(speed^2))



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 6.77  0.012 *
## ---
## 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 6.95  0.011 *
## 3     47 10825  1       529 2.30  0.136
## ---
## 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

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.

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.

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.

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

Making publication quality plots 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(ggplot2)
library(scales)
library(ggtern)
library(ggmap)
library(rgdal)

## Loading required package:  sp
## rgdal:  version:  0.8-16, (SVN revision 498)
## 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/gdal
## 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/proj

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(plyr)
library(grid)
library(photobiologygg)
```

C.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`, together with our own `photobiologygg`. In this appendix we give a very brief introduction to the 'grammar of graphics' and `ggplot2`.

C.3 Bases of plotting with `ggplot2`

The grammar of graphics is based on aesthetics (`aes`) as for example colour, 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 geom:

```
ggplot(sun.spc, aes(x=w.length, y=s.q.irrad)) +
  geom_line()

## Error:  object 'sun.spc' not found
```

Aesthetics can be 'assigned' to data variables, and to constants:

```
ggplot(sun.spc, aes(x=w.length, y=s.q.irrad, colour="red")) +
  geom_line()

## Error:  object 'sun.spc' not found
```

Data can be the 'result of a calculation':

```
ggplot(sun.spc * gg400.spc, aes(x=w.length, y=s.q.irrad)) +
  geom_line()

## Error:  object 'sun.spc' not found
```

The aesthetics and data given as `ggplot`'s arguments become the defaults for the geoms, but geoms also take aesthetics and data as arguments, which then override the defaults:

```
ggplot(sun.spc, aes(x=w.length, y=s.q.irrad)) +
  geom_line(colour="red") +
  geom_line(data=sun.spc * gg400.spc, colour="blue")
```

```
## Error:  object 'sun.spc' not found
```

Another way of plotting these data, is to first assemble a single data.frame or data.table with all the data. We will use the example data that is as data.tables:

```
unfiltered_sun.spc <- copy(sun.spc)

## Error:  object 'sun.spc' not found

unfiltered_sun.spc[ , filter := "none"]

## Error:  object 'unfiltered_sun.spc' not found

filtered_sun.spc <- sun.spc * gg400.spc

## Error:  object 'sun.spc' not found

filtered_sun.spc[ , filter := "GG400"]

## Error:  object 'filtered_sun.spc' not found

data4plot.spc <- rbind(unfiltered_sun.spc, filtered_sun.spc)

## Error:  object 'unfiltered_sun.spc' not found

data4plot.spc[ , filter := factor(filter)]

## Error:  object 'data4plot.spc' not found
```

Now we can do the plot assigning `filter` to an aesthetic:

```
ggplot(data4plot.spc, aes(x=w.length, y=s.q.irrad, colour=filter)) +
  geom_line()

## Error:  object 'data4plot.spc' not found
```

We can assign the same variable to more than one aesthetic:

```
ggplot(data4plot.spc, aes(x=w.length, y=s.q.irrad,
                          colour=filter, linetype=filter)) +
  geom_line()

## Error:  object 'data4plot.spc' not found
```

We can change the labels for the different aesthetics, and give a title:

```
ggplot(data4plot.spc,
       aes(x=w.length, y=s.q.irrad * 1e6, colour=filter)) +
  geom_line() +
  labs(x="Wavelength (nm)",
       y="Spectral photon irradiance\n(umol m-2 s-1 nm-1)",
       colour="Filter:",
       title="Filtered and unfiltered solar spectrum")

## Error:  object 'data4plot.spc' not found
```

We can assign a ggplot object to a variable:

```
myplot <- ggplot(data4plot.spc,
       aes(x=w.length, y=s.q.irrad * 1e6, colour=filter)) +
  geom_line() +
  labs(x="Wavelength (nm)",
       y="Spectral photon irradiance\n(umol m-2 s-1 nm-1)",
       colour="Filter:",
       title="Filtered and unfiltered solar spectrum")

## Error:  object 'data4plot.spc' not found
```

And now we can add more elements to the plot stored in the variable:

```
myplot + theme_bw()

## Error:  object 'myplot' not found

myplot + scale_y_log10()

## Error:  object 'myplot' not found
```

C.4 Adding fitted curves, including splines

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

```
ggplot(sun.spc, aes(x=w.length, y=s.q.irrad)) +
  geom_line() +
  stat_smooth()

## Error:  object 'sun.spc' not found
```

By adjusting one of its parameters, `span`, we make the spline more "flexible":

```
ggplot(sun.spc, aes(x=w.length, y=s.q.irrad)) +
  geom_line() +
  stat_smooth(span=0.1)

## Error:  object 'sun.spc' not found
```

Instead of using the default spline, we can use a linear model fit. In this example we use a polynomial of order 8, fitted by lm, as smoother:

```
ggplot(sun.spc, aes(x=w.length, y=s.q.irrad)) +
  geom_line() +
  stat_smooth(method="lm", formula=y~poly(x,8))

## Error:  object 'sun.spc' not found
```

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.

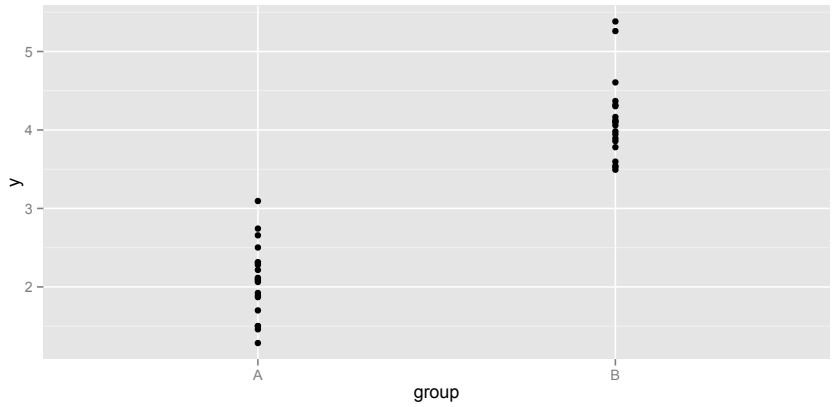
C.5 Adding statistical "summaries"

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)))
  )
```

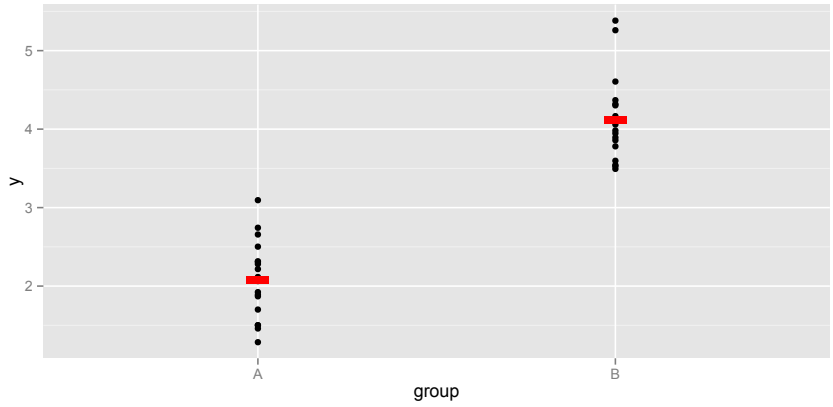
Now we use these data to plot means and confidence intervals by group:

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

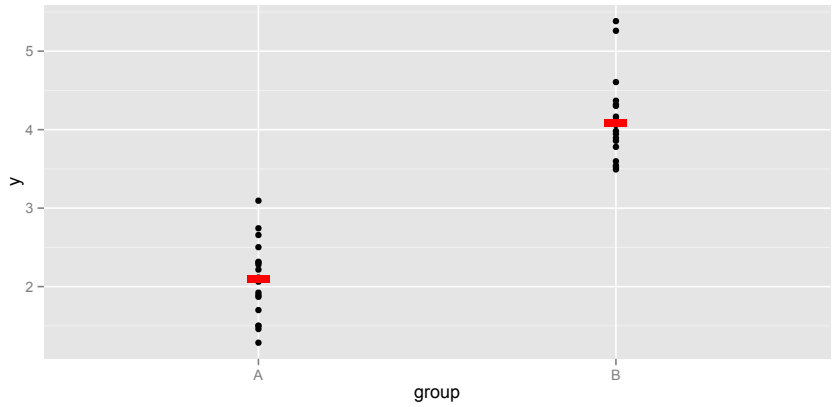


We have saved the base figure in `fig2`, so now we can play with different summaries. We first add just the mean. In this case we need to add as argument to `stat_summary` the geom to use, as the default one expects data for plotting error bars, in later examples, this is not needed.

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

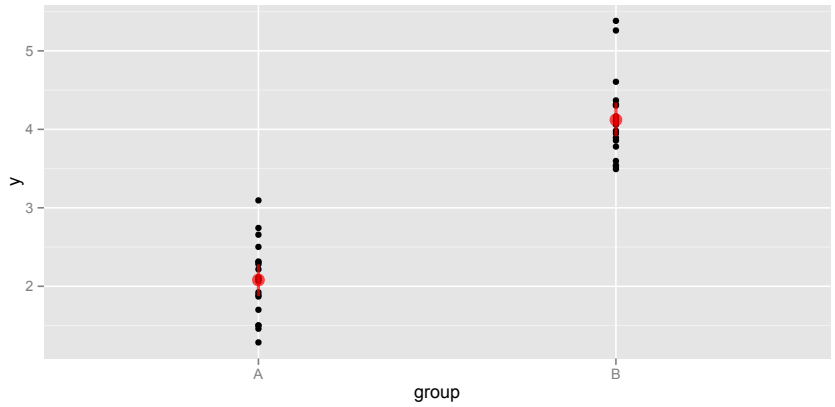


```
fig2 + stat_summary(fun.y = "median", geom="point",
                    colour="red", shape="-", size=20)
```



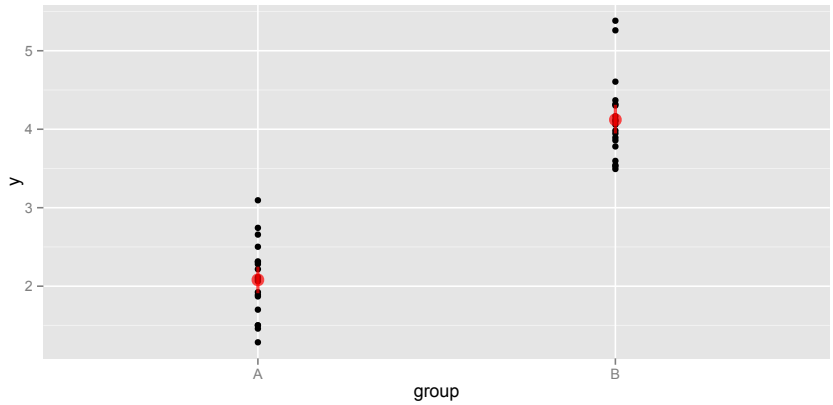
We can add the means and $p = 0.95$ confidence intervals not assuming normality (using the actual distribution of the data by bootstrapping):

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



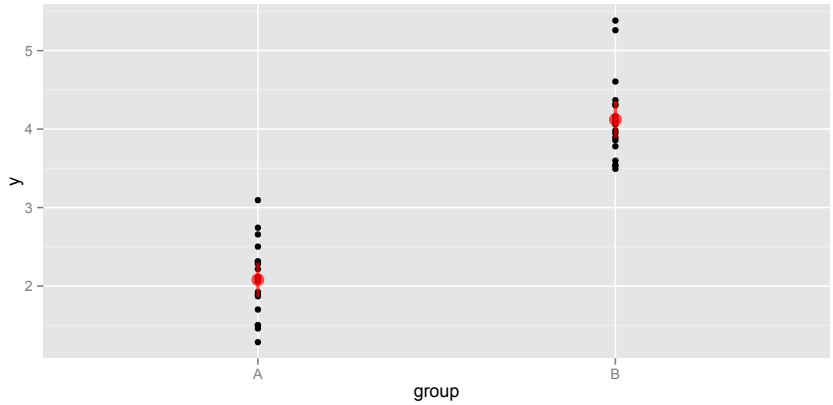
We can instead add the means and $p = 0.90$ confidence intervals, by supplying a value to parameter `conf.int`:

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



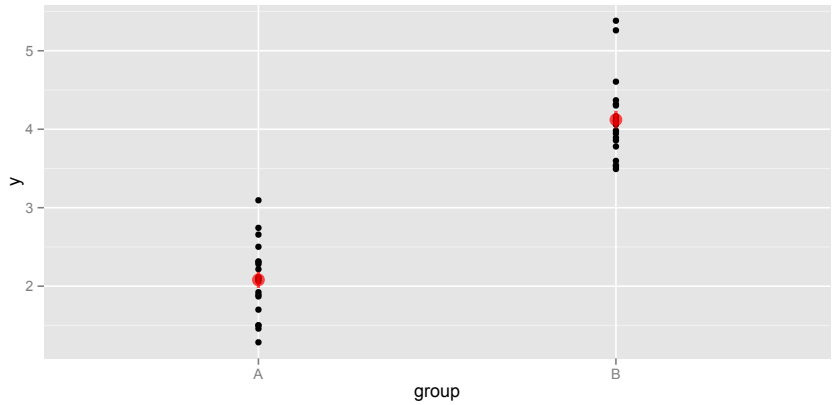
We can add the mean and $p = 0.95$ confidence intervals assuming normality (using the t distribution):

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



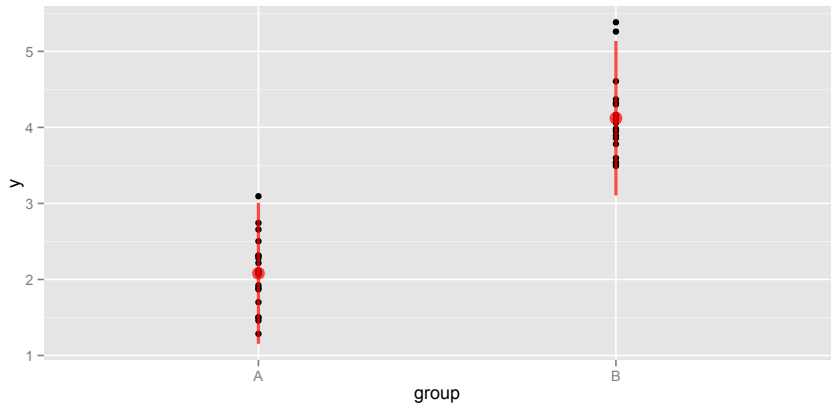
In this case the multiplier `mult` is by default is calculated from the t distribution according to degrees of freedom, but if we force the multiplier to 1, then we get error bars corresponding to ±s.e. (standard errors).

```
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.

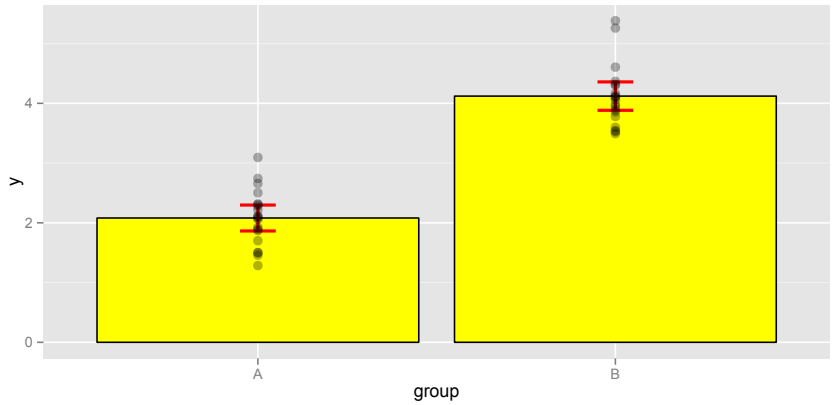
```
fig2 + stat_summary(fun.data = "mean_sdl",
                    colour="red", size=1, alpha=0.7)
```



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.

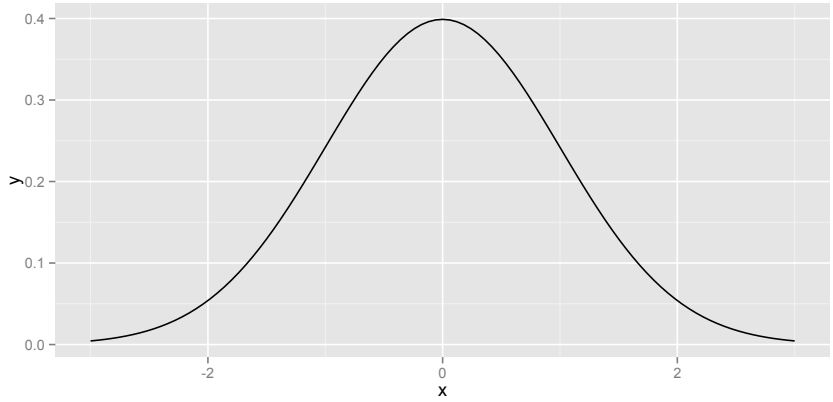
```
ggplot(data=fake.data, aes(y=y, x=group)) +
  stat_summary(fun.y = "mean", geom = "bar",
               fill="yellow", colour="black") +
  stat_summary(fun.data = "mean_cl_normal",
               geom = "errorbar",
               width=0.1, size=1, colour="red") +
  geom_point(size=3, alpha=0.3)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



C.6 Plotting functions

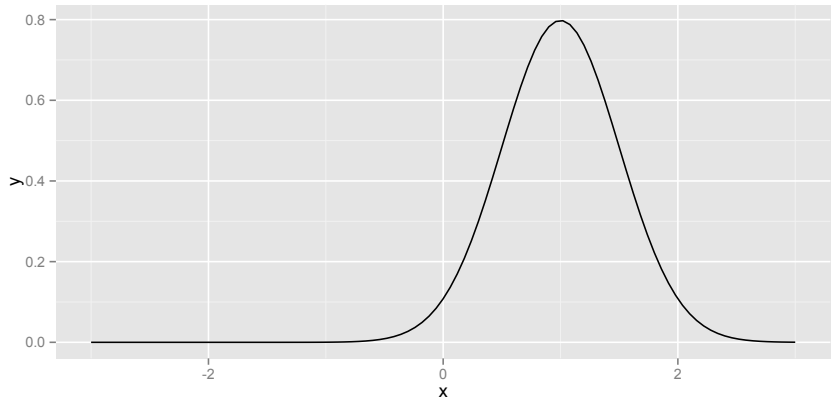
We can also directly plot functions, without need to generate data beforehand:

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



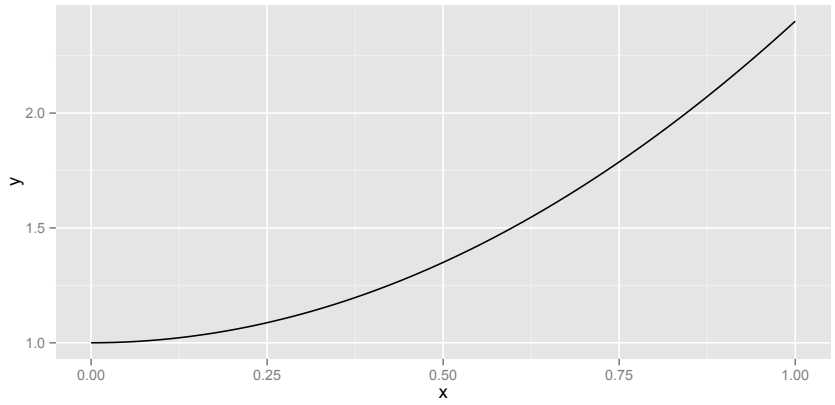
We can even pass additional arguments to a function:

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



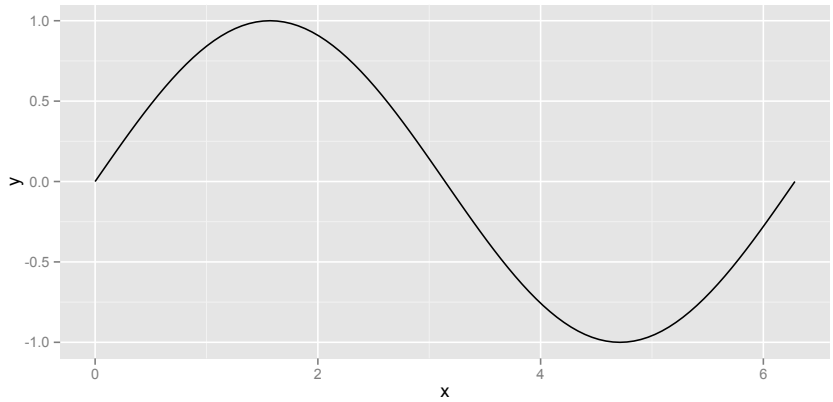
Of course, user-defined functions (not shown), and anonymous functions can also be used:

```
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))
```



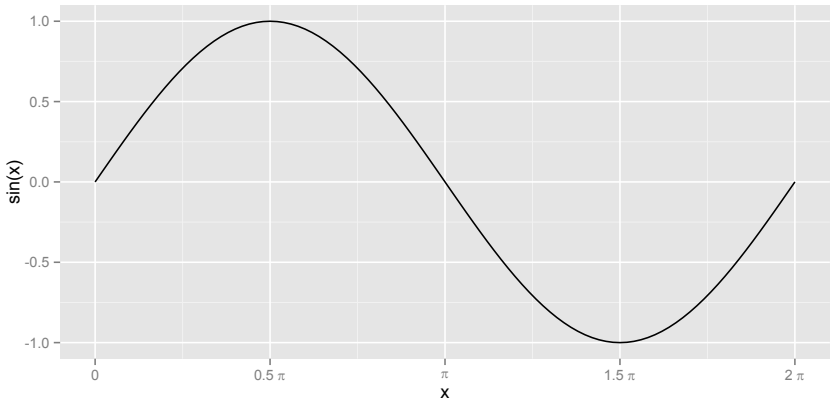
Here is another example of a predefined function, but in this case the default scale is not the best:

```
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:

```
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)")
```

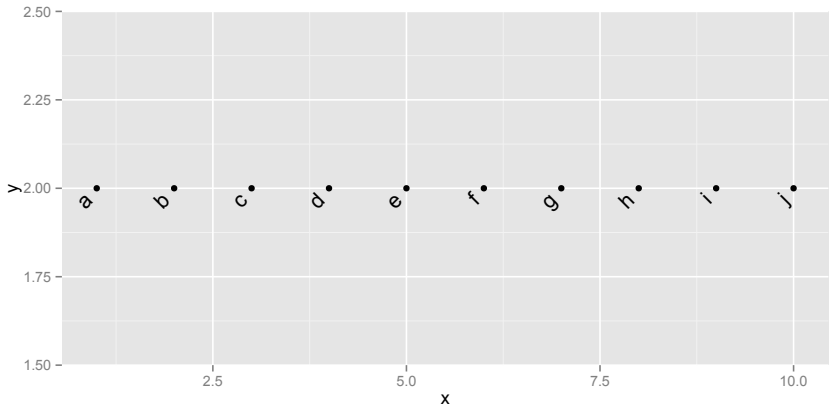


C.7 Plotting text

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 labe 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=1:10, y=rep(2, 10), label=paste(letters[1:10], " "))
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.)

C.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`, posititon (`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 posiiton 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.

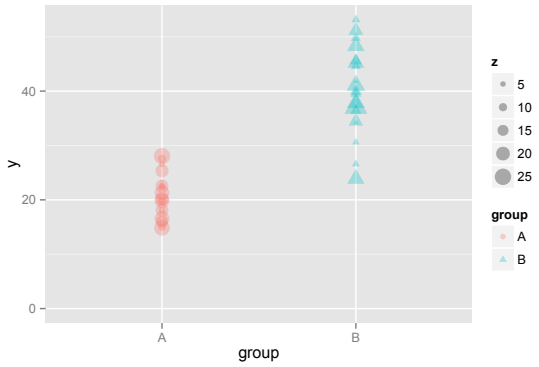
Discrete scales work in a similar way. We can use `scale_colour_identity` and have in our data a variable with values that are valid colour names like "red" or "blue". However we can also assign

the `colour` aesthtic 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.

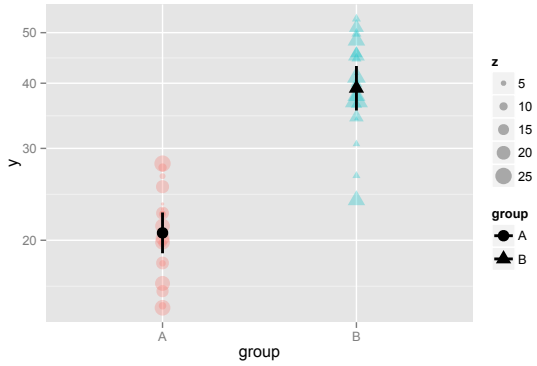
Advanced scale manipulation requires the package `scales` to be loaded. Some simple examples follow.

```
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
```



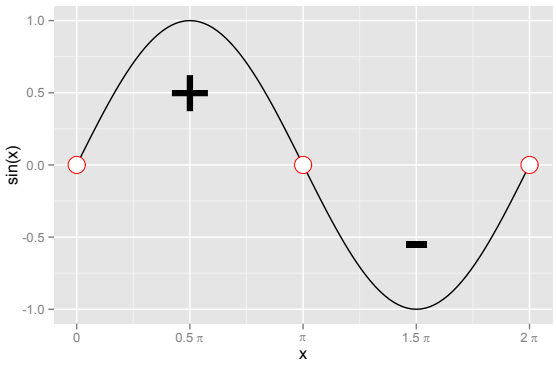

```
fig2 +
  scale_y_log10(breaks=c(10,20,30,40,50,60)) +
       stat_summary(fun.data = "mean_cl_normal",
                    colour="black", size=1, alpha=1)

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



C.9 Adding annotations

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

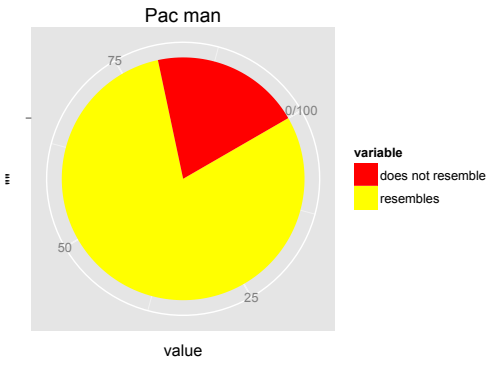
```
ggplot(data.frame(x=c(0, 2 * pi)), aes(x=x)) +
  stat_function(fun=sin) +
  scale_x_continuous(
    breaks=c(0, 0.5, 1, 1.5, 2) * pi,
    labels=c("0", expression(0.5~pi), expression(pi),
             expression(1.5~pi), expression(2~pi))) +
  labs(y="sin(x)") +
  annotate(geom="text",
         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



C.10 Circular plots

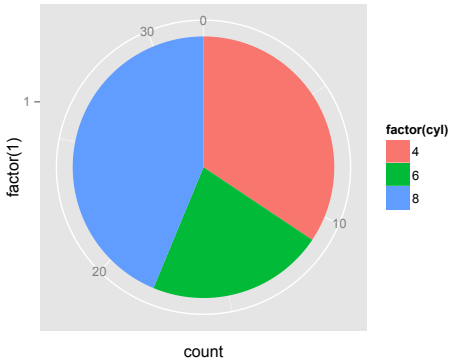
A funny example stolen from the ggplot2 website at `http://docs.ggplot2.org/current/coord_polar.html`.

```
# 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")
```



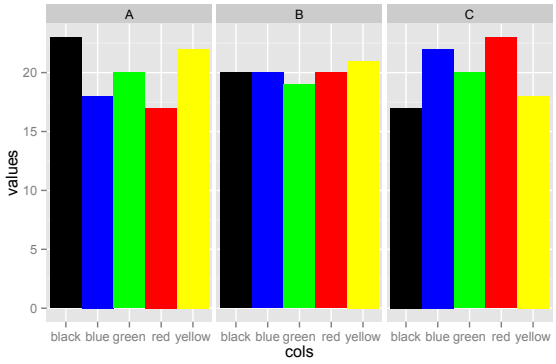
C.11 Pie charts vs. bar plots example

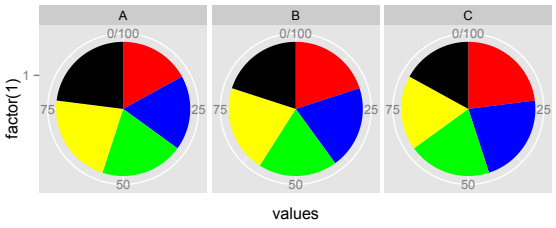
There is an example figure widely used in Wikipedia to show how much easier it is to 'read' bar plots than pie charts (`http://commons.wikimedia.`

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")
```





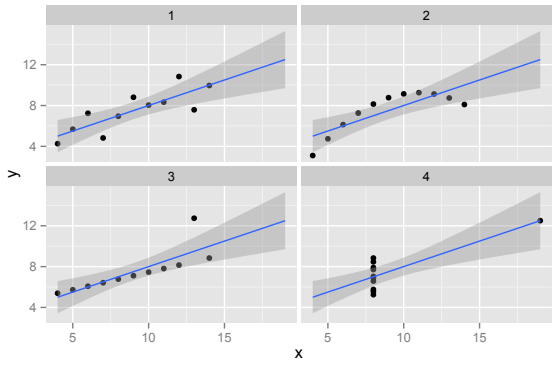
C.12 A classical example about regression

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()
```

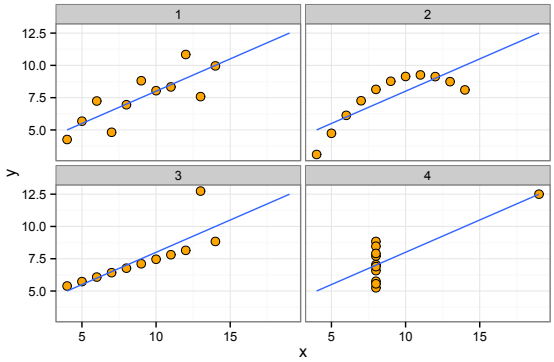
My version using `ggplot2`:

```
# 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, rep(11,4))
# we draw the figure
ggplot(my.anscombe, aes(x,y)) +
  geom_point() +
  geom_smooth(method="lm") +
  facet_wrap(~case, ncol=2)
```



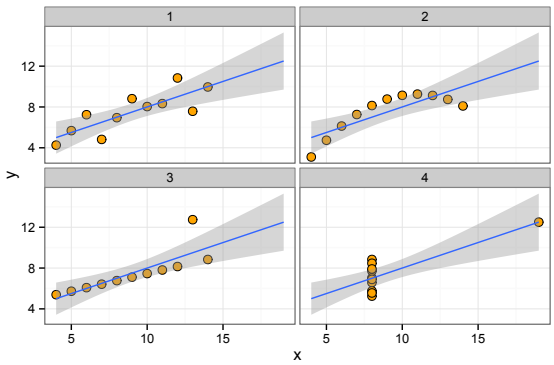
It is not much more difficult to make it look simlar to the original

```
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.

C.13 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`:

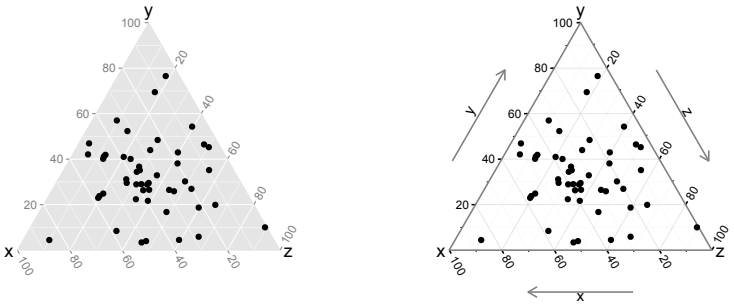
```
fig.trn <- ggplot(my.trn1.data, aes(x,y,z)) +
  coord_tern(L="x",T="y",R="z")
```

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

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

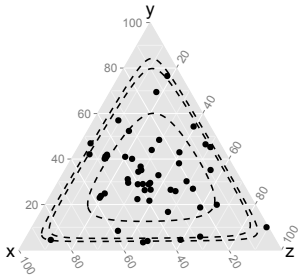
As with any other plot based on `ggplot2` one builds the plot by adding 'layers'. Themes are also supported.

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



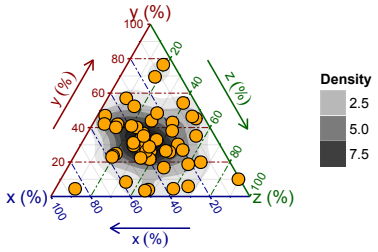
It is possible to also draw confidence regions:

```
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()
```

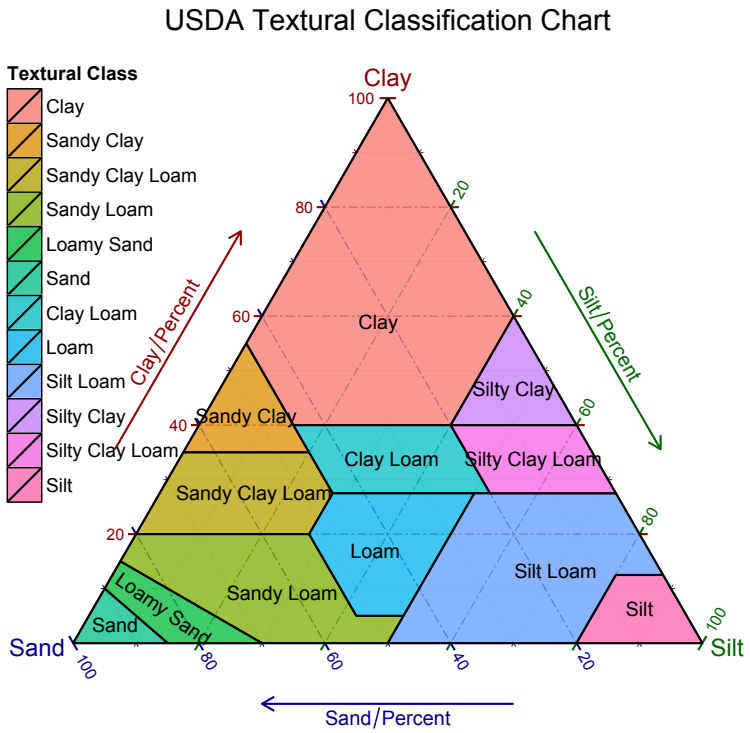


As a final example we reproduce an elaborate ternary plot from `http://www.ggtern.com/2014/01/15/usda-textural-soil-classification/`, the website of the package.

```
# 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')
```



C.14 Plotting data onto maps

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`

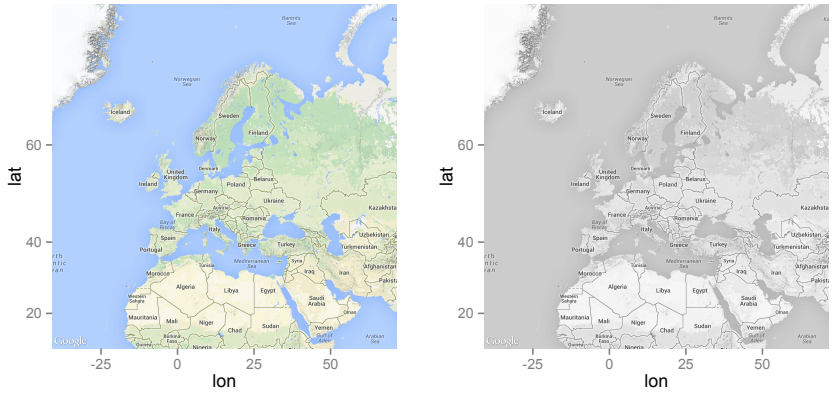
Map from URL: `http://maps.googleapis.com/maps/api/staticmap?center=Europe&zoom=3&size=%20640x640&scale=%202&maptype=terrain&sensor=false`

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")
```

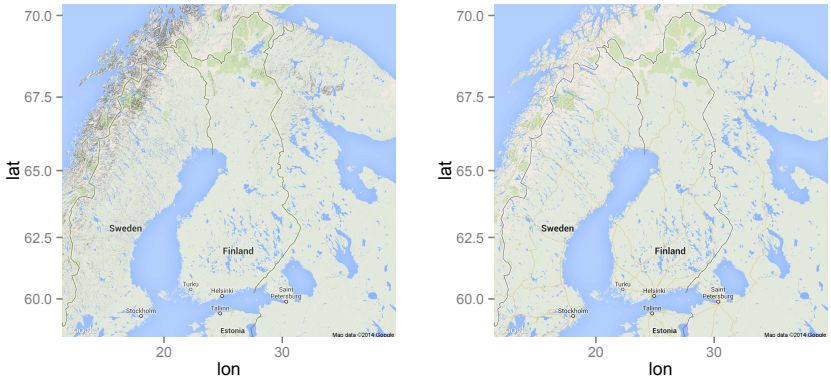
To demostrate the option to fetch a map in black and white instead of the default colour version, we use a map of Europe of type `terrain`.

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

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



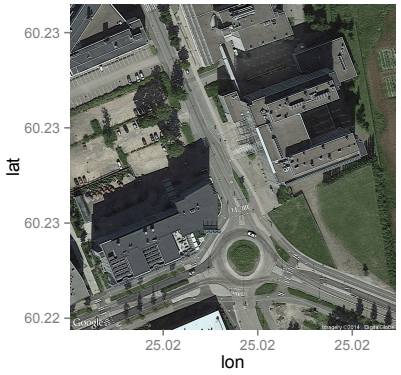
To demonstrate the difference between type `roadmap` and the default type `terrain`, we use the map of Finland. Note that we search for "Oulu" instead of "Finland" as Google Maps takes the position of the label "Finland" as the center of the map, and clips the northern part. By means of `zoom` we override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

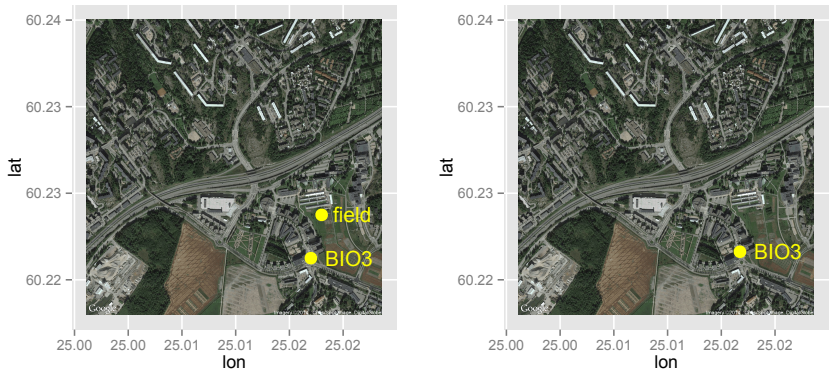
```
BIO3 <- get_map("Viikinkaari 1, 00790 Helsinki",
                zoom=18,
                maptype="satellite")
ggmap(BIO3)
```



We will now show a simple example of plotting data on a map, first by expliciply 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 writting. In addition a worked out example of how to plot shape files, and how to dowload 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"
```

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,

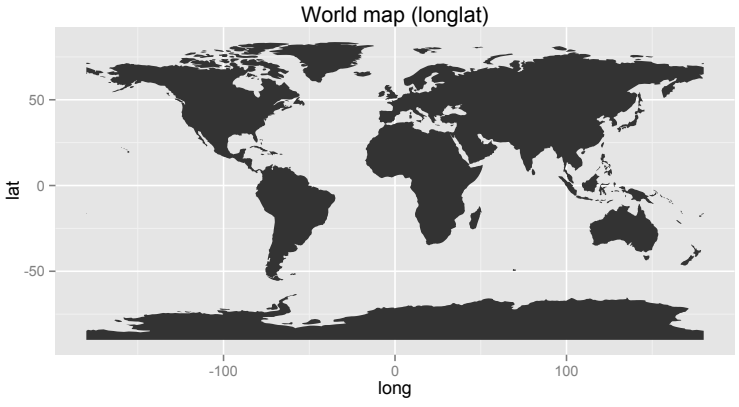
```
countries <- readOGR("./maps", layer="ne_110m_admin_0_countries")

## 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`.

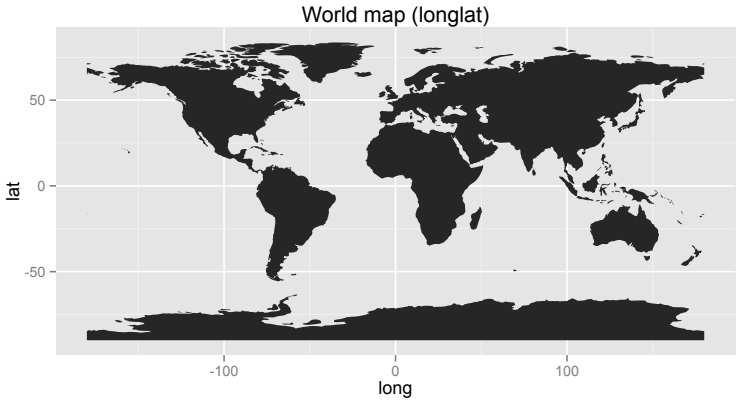
```
ggplot(wmap.data, aes(long,lat, group=group)) +
  geom_polygon() +
  labs(title="World map (longlat)") +
  coord_equal()
```



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()
```



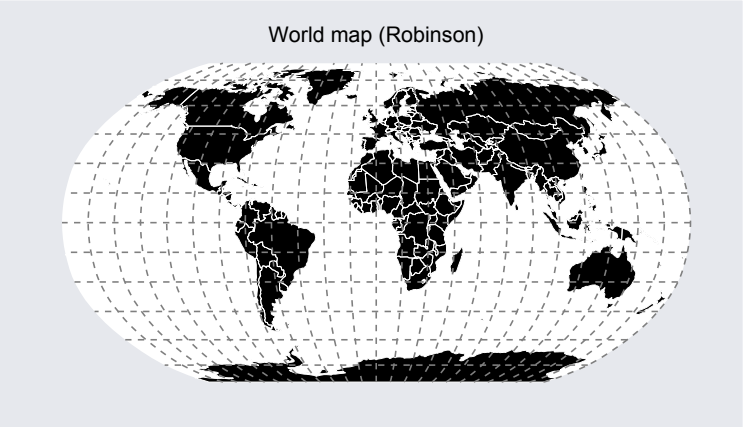
When plotting a map using a projection, many default elements of the `ggplot` theme need to be removed, as the data is no longer in units of degrees of latitude and longitude and axes and their labels are no longer meaningful.

```
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 transfromation 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)") +
```

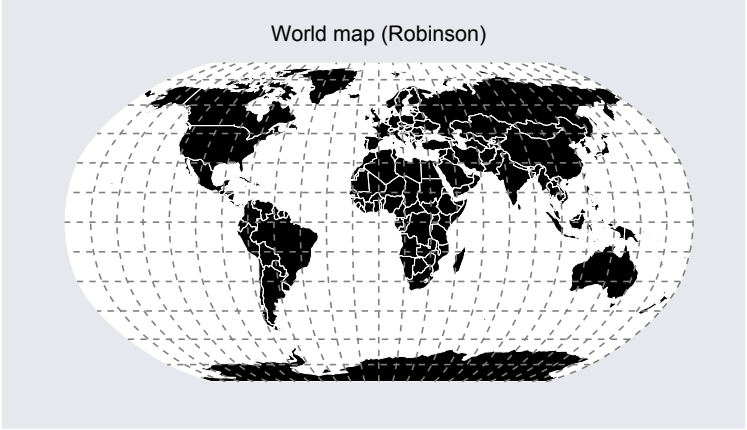
```
  coord_equal() +
  theme_map_opts +
  scale_fill_manual(values=c("black", "white"),
                    guide="none")
```



C.15 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/`.

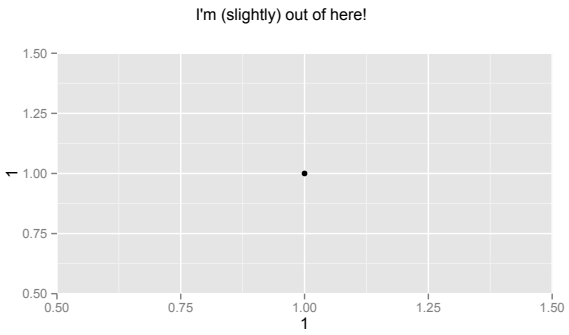
```
d = data.frame(x = sort(rlnorm(300)),
               y = sort(rlnorm(300)),
               grp = 1)

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 25 rows containing missing values
(geom_point).
```



C.16 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!")
```



C.17 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()
```

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 3.1.1 (2014-07-10)
## 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   grid      stats     graphics
## [5] grDevices utils     datasets  methods
```

```
## [9] base
##
## other attached packages:
##  [1] Hmisc_3.14-4
##  [2] Formula_1.1-1
##  [3] survival_2.37-7
##  [4] lattice_0.20-29
##  [5] rgdal_0.8-16
##  [6] sp_1.0-15
##  [7] scales_0.2.4
##  [8] microbenchmark_1.3-0
##  [9] gridExtra_0.9.1
## [10] ggtern_1.0.3.2
## [11] ggmap_2.3
## [12] photobiologygg_0.1.9
## [13] plyr_1.8.1
## [14] splus2R_1.2-0
## [15] proto_0.3-10
## [16] photobiologyAll_0.1.1
## [17] photobiologySun_0.1.2
## [18] photobiologyCry_0.1.2
## [19] photobiologyPhy_0.2.5
## [20] photobiologySensors_0.1.6
## [21] photobiologyLEDs_0.1.1
## [22] photobiologyLamps_0.1.10
## [23] photobiologyFilters_0.1.9
## [24] data.table_1.9.2
## [25] photobiologyVIS_0.1.5
## [26] photobiologyUV_0.2.8
## [27] photobiology_0.3.4
## [28] lubridate_1.3.3
## [29] ggplot2_1.0.0
## [30] knitr_1.6
##
## loaded via a namespace (and not attached):
##  [1] cluster_1.15.2      colorspace_1.2-4
##  [3] digest_0.6.4        evaluate_0.5.5
##  [5] formatR_0.10        gtable_0.1.2
##  [7] highr_0.3           labeling_0.2
##  [9] latticeExtra_0.6-26 mapproj_1.2-2
## [11] maps_2.3-7          MASS_7.3-33
## [13] memoise_0.2.1       munsell_0.4.2
## [15] png_0.1-7           RColorBrewer_1.0-5
## [17] Rcpp_0.11.2         reshape2_1.4
## [19] RgoogleMaps_1.2.0.6 rjson_0.2.14
## [21] RJSONIO_1.2-0.2     stringr_0.6.2
## [23] tools_3.1.1
```