R for Photobiology

A handbook

Pedro J. Aphalo

DRAFT

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).

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 photochemisatry.

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 explain how to .

4.1 Task:

CHAPTER 5

Photobiology R pacakges

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 generaly useful functions, including many used in the other, more especialized, packages (Table 5.1).

One of the main difficulties when working with specral 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 deffault 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 allways 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 higlighted with a red bullet ●, and those at 'beta' stage with a yellow bullet ●, those relativelly 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 photoCRAN 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 photoCRAN repo can be added to the list of repos that R knows about.

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

```r
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.

```r
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 repos gets corrupted by having two repos 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 startup) 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 afecting 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 pormpt and in scripts whether RStudio is used or not.

After adding the repo 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 photoCRAN repo as shown above, the photobiology packages can be installed and uninstalled through the normal RStudio menues 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 repo will persist only during the current R session. Adding it permanently requires editing the R configuration file.

5.3 How to install the pakages

The examples given in this page assume that photoCRAN is not in the list of repos known to the current R session. See the section 5.2 on the photoCRAN repo 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 bewteen 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 repo and a normal CRAN repo, 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:

```r
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 use the spectral data for light sources.

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)
```

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 $k_B = 1.381 \times 10^{-23}$ J K^{-1}, 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}.

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

```r
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 emmision spectra for different temperatures:

```r
black_body_spectrum(500, 5000)

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

The fucntion is vectorized:

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

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

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

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

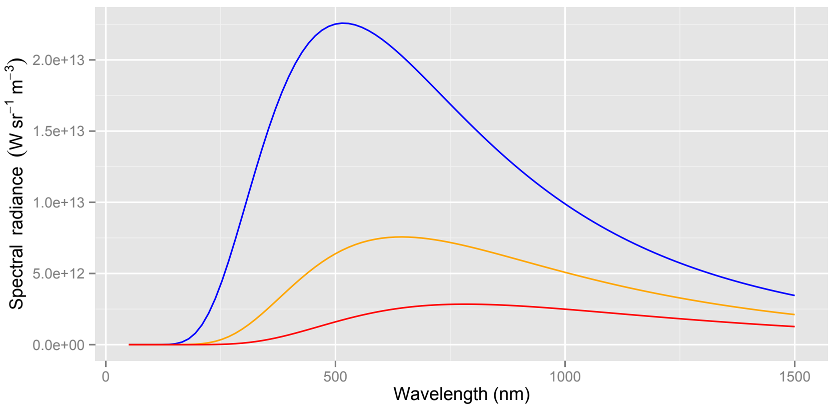
We aware that if two vectors are supplied, then the elements in each one are matched and recycled[1]:

```r
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 emmision 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 tempeartures (5600 K, 4500 K, and 3700 K):

```r
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] Excercise: 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 blackbody as a function of its absolute temperature.

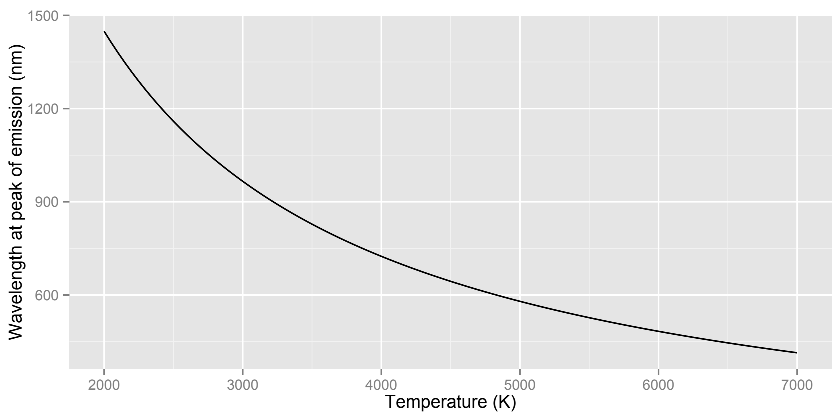
$$\lambda_{max} \cdot T = 2.898 \times 10^6 \,\mathrm{nm\,K} \tag{6.2}$$

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

Basic operations on spectra

Abstract

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

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(photobiologyFilters)
library(photobiologyLeds)

## Error:  there is no package called 'photobiologyLeds'
```

7.2 Task: uniform scaling of a spectrum

In R vectorized operators are available for "generic.scpt", "source.spct", and "filter.spct" objects, and 'recycling' takes places when needed[1]:

```
head(sun.dt)

##    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
```

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

```
## 5:      297 1.533e-04 3.807e-10
## 6:      298 3.670e-04 9.141e-10

head(sun.dt * 2)

##    w.length s.e.irrad s.q.irrad
## 1:      586 5.219e-06 1.278e-11
## 2:      588 1.228e-05 3.019e-11
## 3:      590 4.352e-05 1.073e-10
## 4:      592 1.356e-04 3.355e-10
## 5:      594 3.067e-04 7.614e-10
## 6:      596 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.

7.3 Task: simple operations between two spectra

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

## Warning:  Incompatible methods ("*.generic.spct", "Ops.data.table")
for "*"
## Error:  non-numeric argument to binary operator

head(filtered_sun.dt)

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

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

7.4 Task: 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 math operators. These operators are vectorized, which means that an addition between two vectors adds the elements at each position. A non-sensical 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 7.918e-11
## [6] 4.519e-10
```

These operators cannot be used if the walengths 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 invididual 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:

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

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 is used to define the functions for the four basic math operators.

7.5 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 sspectrum returned includes the limits given as arguments. By default the trimming is done by deleting both spectral irradiance and wavelength values outside the range delinited 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)))
```

```
##   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)))

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

7.6 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,

$$q = h \cdot \nu = h \cdot \frac{c}{\lambda} \tag{7.1}$$

[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 7.1.

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 (7.1) 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

$$q' = h' \cdot \nu = h' \cdot \frac{c}{\lambda} \tag{7.2}$$

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 7.1 while as_quantum_mol uses equation 7.2. To obtain µmol m^{-2} s^{-1} we multiply by 10^6:

```
as_quantum_mol(550, 200) * 1e+06

## [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 1.533e-04
##  [6] 3.670e-04 7.845e-04 1.265e-03 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 3.807e-10
##  [6] 9.141e-10 1.961e-09 3.171e-09 6.602e-09 9.903e-09
```

7.7 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 300 nm has about 400 kJ mol^{-1}, therefore, 1 µmol photons has 0.4 J. Equations 7.1 and 7.2 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) * 0.001

## [1] 199.4

as_energy(300, 1) * 0.001

## [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 3.807e-10
##  [6] 9.141e-10 1.961e-09 3.171e-09 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 1.533e-04
##  [6] 3.670e-04 7.845e-04 1.265e-03 2.624e-03 3.923e-03
```

7.8 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 6.142e-06
##  [6] 2.176e-05 6.780e-05 1.533e-04 3.670e-04 7.845e-04
## [11] 1.265e-03

with(sun.data, interpolate_spectrum(w.length, s.e.irrad, 290:300,
    fill = 0))

##  [1] 0.000e+00 0.000e+00 0.000e+00 2.610e-06 6.142e-06
##  [6] 2.176e-05 6.780e-05 1.533e-04 3.670e-04 7.845e-04
## [11] 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.

7.9 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 recduce errors in calcualtions.

The function `integrate_irradiance` is used internally for intergrating 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 8

Unweighted irradiance

Abstract

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

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(photobiologyVIS)
library(photobiologyUV)
```

8.2 Introduction

8.3 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.

$$E_{\lambda_1<\lambda<\lambda_2} = \int_{\lambda_1}^{\lambda_2} E(\lambda)\ \mathrm{d}\,\lambda \tag{8.1}$$

Let's assume that we want to calculate photosynthetically active radiation (PAR) energy irradiance, for which the most accepted limits are $\lambda_1 = 400$nm and $\lambda_1 = 700$nm. In this example we will use example data for sunlight to calculate $E_{400\,\mathrm{nm}<\lambda<700\,\mathrm{nm}}$:

```
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
```

8.4 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 8.1 and 7.2 we obtain:

$$Q_{\lambda_1<\lambda<\lambda_2} = \int_{\lambda_1}^{\lambda_2} E(\lambda)\, \frac{h' \cdot c}{\lambda} \mathrm{d}\,\lambda \tag{8.2}$$

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

8.5 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:

$$E_{\lambda_1 < \lambda < \lambda_2} = \int_{\lambda_1}^{\lambda_2} Q(\lambda)\, \frac{\lambda}{h' \cdot c} \mathrm{d}\,\lambda \tag{8.3}$$

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 8.1, and the code is:

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

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

8.6 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:

[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`

```
with(sun.data, photon_irradiance(w.length, s.e.irrad, list(Red(),
    Green(), Blue()))) * 1e+06

##   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()))) * 1e+06
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()))) * 1e+06

##     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
```

8.7 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 35

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
```

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

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

UVB("ISO")

## UVB.ISO
## low (nm) 280
## high (nm) 315
```

```
UVB("CIE")

## UVB.CIE
## low (nm) 280
## high (nm) 315

UVB("medical")

## UVB.medical
## low (nm) 290
## high (nm) 320

UVB("none")

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

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
```

8.8 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

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

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

## my.colour
##     314.1
```

8.9 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]:

$$\zeta = \frac{Q_{655\mathrm{nm}<\lambda<665\mathrm{nm}}}{Q_{725\mathrm{nm}<\lambda<735\mathrm{nm}}} \tag{8.4}$$

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
```

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
```

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

8.10 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.

8.11 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:

$$\bar{q}' = \frac{Q_{\lambda_1<\lambda<\lambda_2}}{E_{\lambda_1<\lambda<\lambda_2}} \tag{8.5}$$

```
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 $\mathrm{W} = \mathrm{J\,s^{-1}}$), 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())) *
    1e+06

## [1] 4.544
```

8.12 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
```

8.13 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 8.12 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 9

Weighted and effective irradiance

Abstract

In this chapter we explain how to calculate weighted 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

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 calculataions for weighted irradiances compared to those for unweighted irradiances decribed in chapter 8. In particular it is important that you read sections 8.3, 8.4, **??**, and 8.6 before reading the present chapter.

9.3 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
```

9.4 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 calcualtion 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 Timijan'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"
```

9.5 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
```

9.6 Introduction

CHAPTER 10

Colour

Abstract

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

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)
```

10.2 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 avaialble, one for monochromatic light taking as argument wavelength values, and one for polychromatic light taking as argument spectral energy irradiances and the corresponding wave length values. The third function can be used to calculate a representative RGB colour for a band of the spectrum represented as a range of wavelengths, based on the assumption of a flat energy irradiance accross this range.

By default CIE coordinates for *typical* human vision are used, but the functions have a parameter that can be used for suppying a different chromaticity definition. 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.

10.3 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"
```

10.4 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"
```

10.5 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"
```

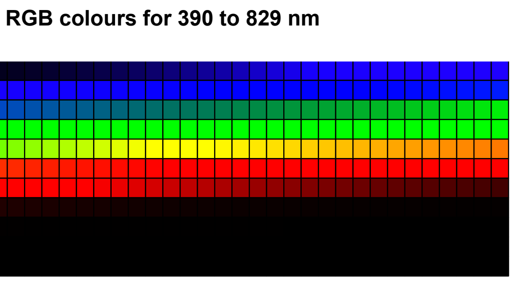
```
with(sun.data, s_e_irrad2rgb(w.length, s.e.irrad, sens = ciexyzCC2.data))

## [1] "#B63C37"

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

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

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

10.6 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)])
}
```



CHAPTER 11

Photoreceptors

Abstract

In this chapter we explain how to .

11.1 Task:

CHAPTER 12

Radiation sources

Abstract

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

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)
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, opts,
##     stat_density2d, stat_smooth, theme, theme_bw,
##     theme_classic, theme_get, theme_gray, theme_grey,
##     theme_minimal, theme_set, theme_update

library(photobiologygg)
```

12.2 Introduction

12.3 Task: using the data

12.4 Task: extraterrestrial solar radiation spectra

12.5 Task: terrestrial solar radiation spectra

12.6 Task: incandescent lamps

12.7 Task: discharge lamps

12.8 Task: LEDs

CHAPTER 13

Filters

Abstract

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

13.1 Packages used in this chapter

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

```
library(photobiology)
library(photobiologyFilters)
library(photobiologySun)
library(photobiologyLamps)
library(photobiologyLEDs)
library(photobiologyVIS)
library(photobiologyUV)
library(ggplot2)
library(ggtern)
library(photobiologygg)
```

13.2 Introduction

13.3 Task: using the data

13.4 Task: spectral transmittance for optical glass filters

13.5 Task: spectral transmittance for plastic films

13.6 Task: spectral transmittance for plastic sheets

CHAPTER 14

Plotting spectra and colours

Abstract

In this chapter we explain how to plot spectra and colours, using packaages 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.

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:

```r
library(photobiology)
library(photobiologyVIS)
library(photobiologyUV)
library(ggplot2)
library(ggtern)
library(photobiologygg)
```

14.2 Introduction to ploting spectra

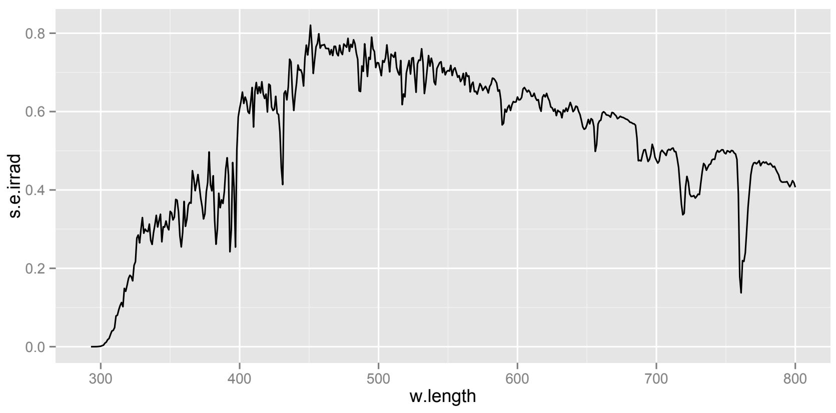
We show in this chapter examples of how one can plot spectra. All the examples are done with package ggplot2, sometimes using in additon other packages.

ggplot2 is feature-frozen. Consequently is a good basis for adding application especific functionality on separate packages. It uses the grammar of graphics for describing the plots. This grammar, because it is consistent, tends to be easier to understand, and it is easier to design no functionality that uses similar 'language' to that used by the original package.

14.3 Task: ploting spectra with `ggplot2`

We create a simple line plot, assign it a variable called `fig_sun` and then on the next line `print` it.

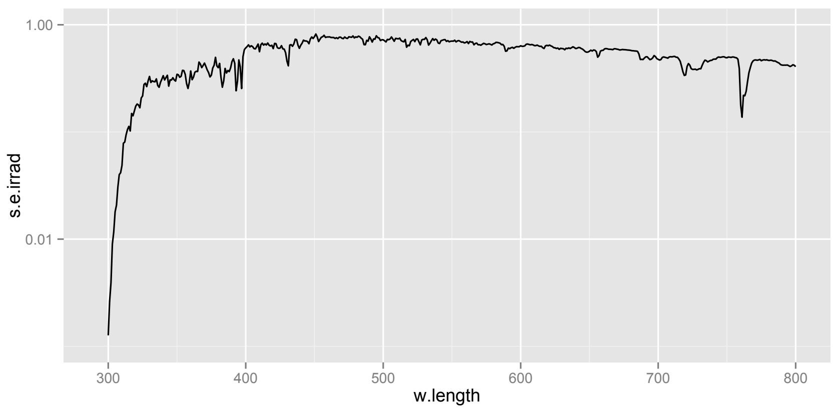
```
fig_sun <- ggplot(data = sun.data, aes(x = w.length, y = s.e.irrad)) +
    geom_line()
fig_sun
```



14.4 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 + scale_y_log10(limits = c(0.001, 1))

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



14.5 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_thershold` the fraction of the total span along the irradiance is taken into account (a value of 0.5, that only peaks in the upper half of the curve are returned; a value of -0.5 works similarly but for the lower half of the y-range):

```
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

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

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

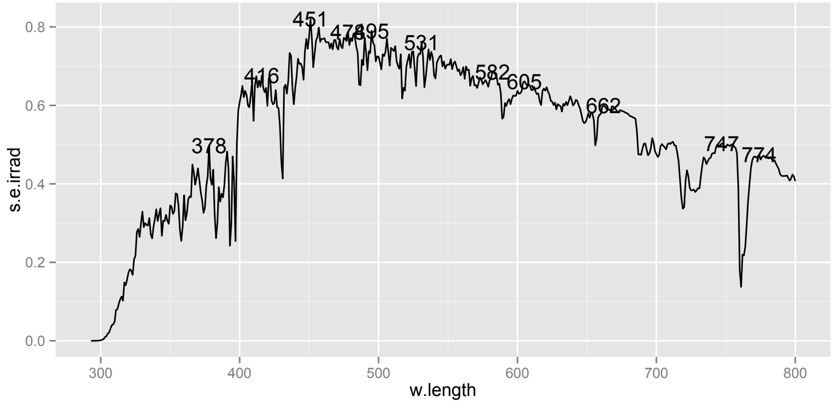
The equivalent function for finding valleys is `get_valleys` taking the same paraameters as `get_peaks` but returning the wavelengths at which the valleys are located.

```
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
```

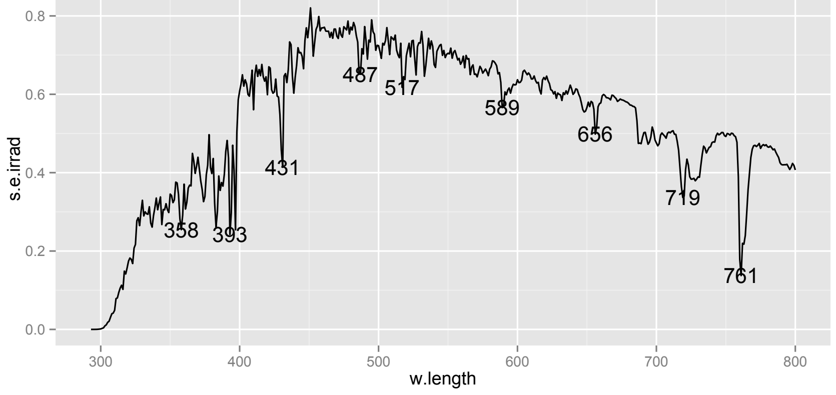
14.6 Task: annotating peaks and valleys in spectra

Here we show the 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 14.5. We reuse once more `fig_sun` saved in section 14.3.

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

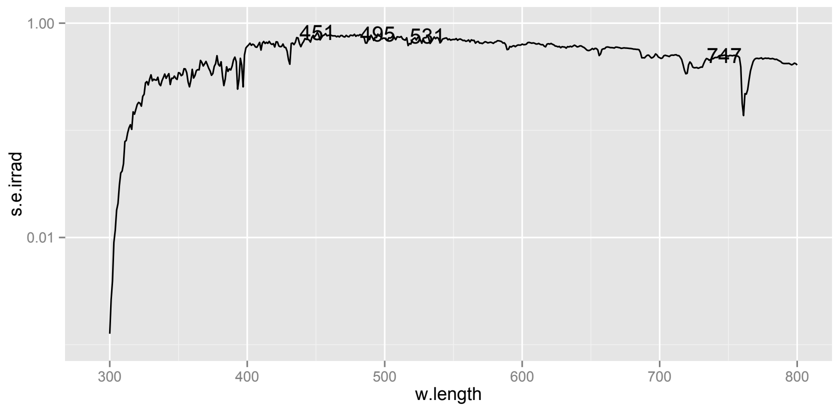


```
fig_sun + stat_valleys(span = 51)
```



Here we use the stats together with a logarithmic scale.

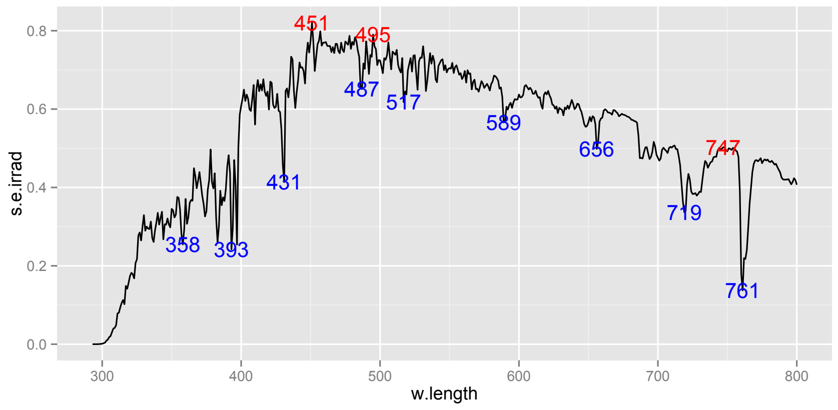
```
fig_sun + stat_peaks(span = 41, ignore_threshold = 0.01) + scale_y_log10(limits = c(
    1))

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



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 choesen geom. By default geom_text is used.

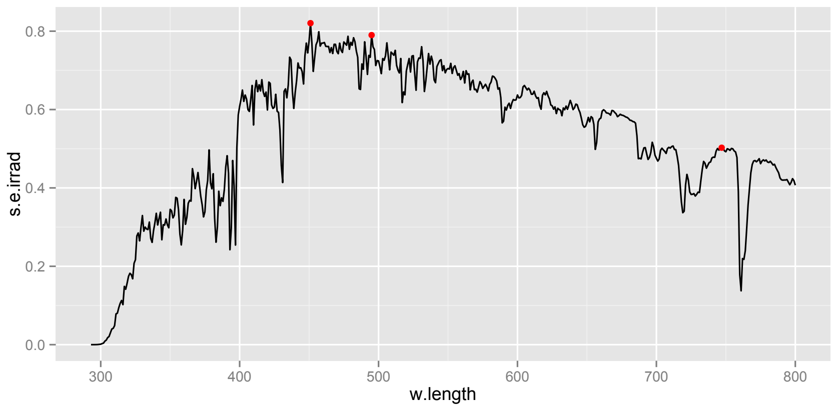
We can change for example the colour:

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



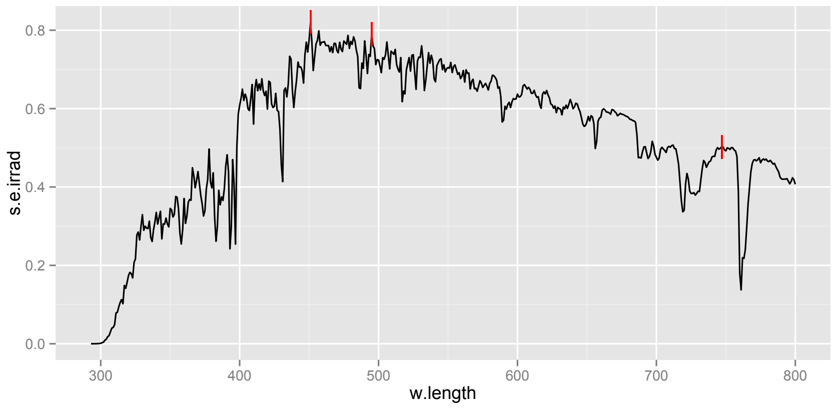
We can also use a different geom, in this case geom_point, however, be aware that the geom parameter takes as argument a character string giving the name of the geom, in this case "point". To keep the code simpler, we will use a single stat in in the next two figures.

```
fig_sun + stat_peaks(colour = "red", geom = "point", span = 51)
```



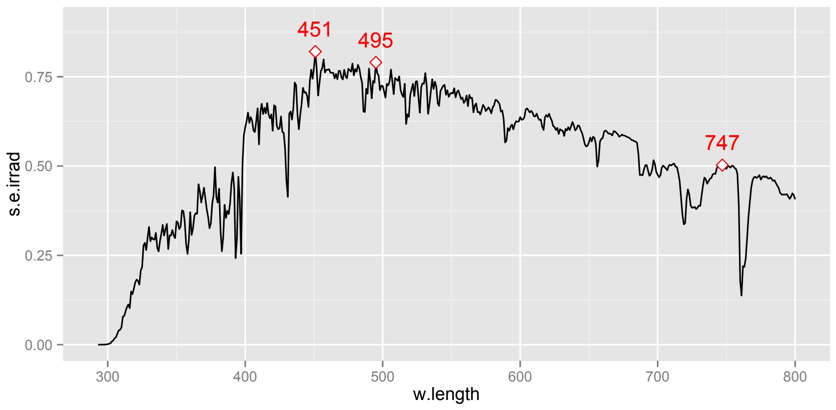
We change a few additional aesthetics of the points: we set `shape` to a character, and set its size to 6.

```
fig_sun + stat_peaks(colour = "red", geom = "point", shape = "|",
    size = 6, span = 51)
```



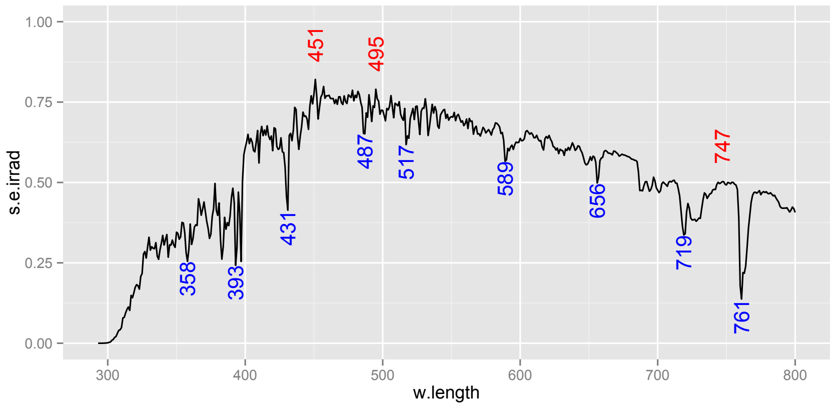
We can add the same stat twice, each time with a different geom in the next example. First we add points to mark the peaks, and afterwars add labels using geom "text". For the shape we use one that supports 'fill', and set the fill to "white" but keep the border of the symbol "red" by setting `color`, we also change the `size`. With the labels we use `vjust` to 'justify' the text moving the labels vertically. In addition we expand the y-axis scale so that the labels fall within the plotting area.

```
fig_sun + stat_peaks(colour = "red", geom = "point", shape = 23,
    fill = "white", size = 3, span = 51) + stat_peaks(colour = "red",
    vjust = -1, span = 51) + expand_limits(y = 0.9)
```



Finally an example with rotated labels, using different colours for peaks and valleys. Be aware that the 'justification' direction is referenced to the position of the text, and for this reason to mode the labels upwards we use `hjust` because the displacement is horizontal with respect to the text of the label.

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



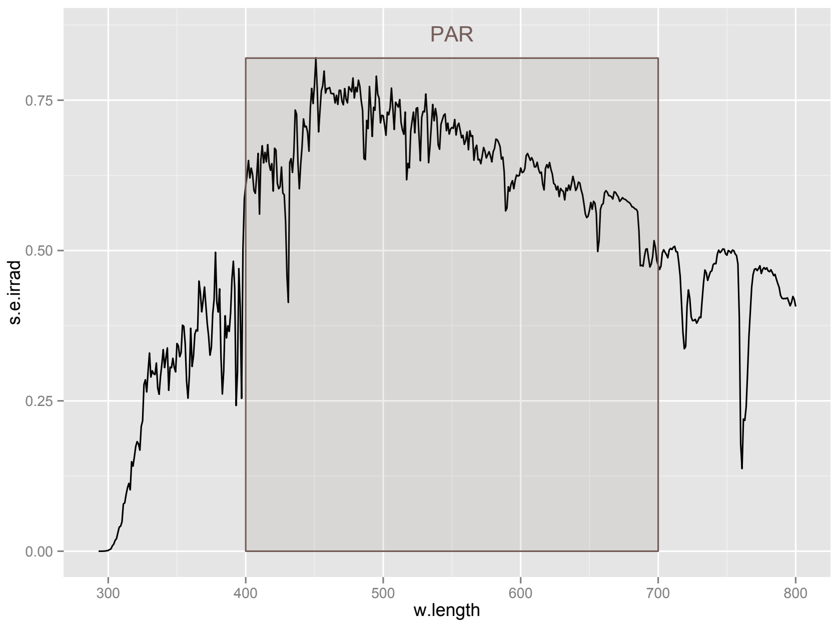
See section **??** in chapter 12 for an example these stats together with facets.

14.7 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 deffault 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`.

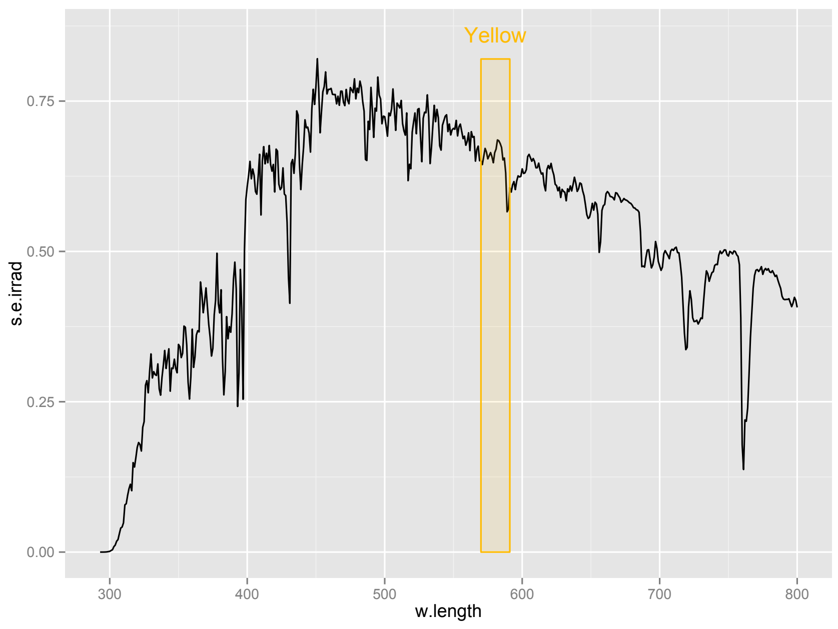
Here is an example for PAR using defaults, and with arguments supplied only for parameters with no defaults. The example does the annotation using two different 'geoms', `"rect"` for marking the region, and `"text"` for the labels.

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



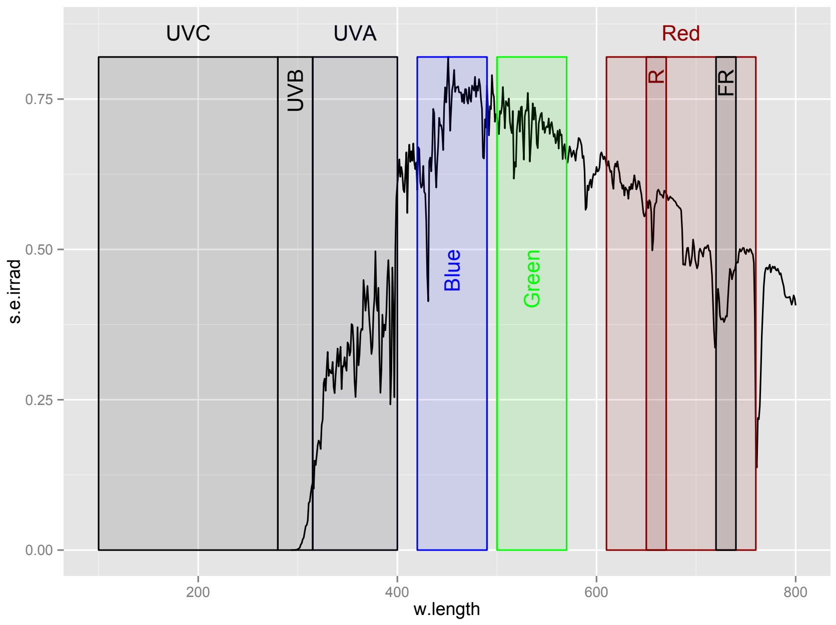
This example annotates a narrow waveband.

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



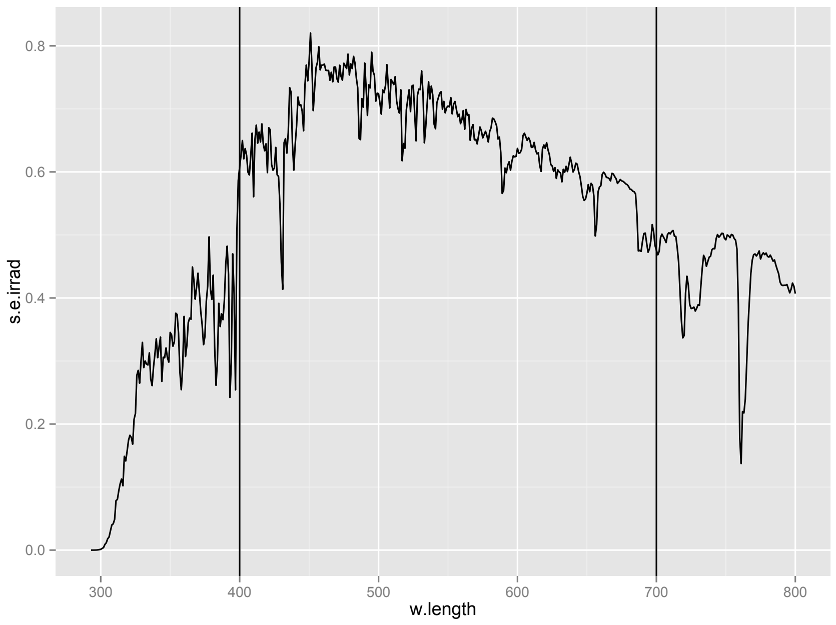
Now an example that is more complex, and demostrates the flexibility of plots produced with `ggplot2`. We add annotations for eight different wavebands, some of them overlaping. For each one we use two 'geoms' and some labels are rotated and justified to keep them centred.

```
figv2 <- fig_sun + 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 = 0.8,
    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 = 0.5, 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 = 0.8, angle = 90, hjust = 1) + annotate_waveband(Far_red("Smith"),
    "rect", ymax = 0.82) + annotate_waveband(Far_red("Smith"),
    "text", y = 0.8, angle = 90, hjust = 1)
figv2
```



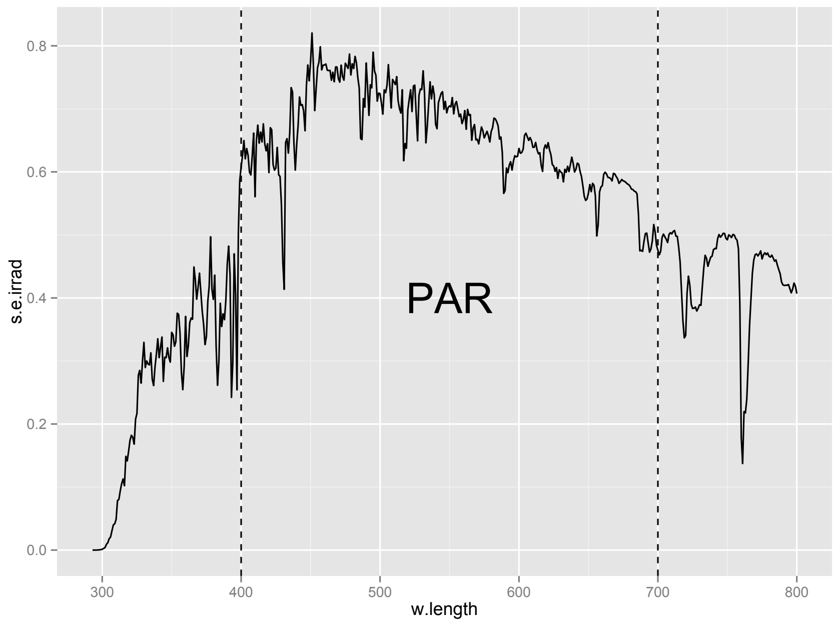
A simple example using `geom_vline`:

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



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

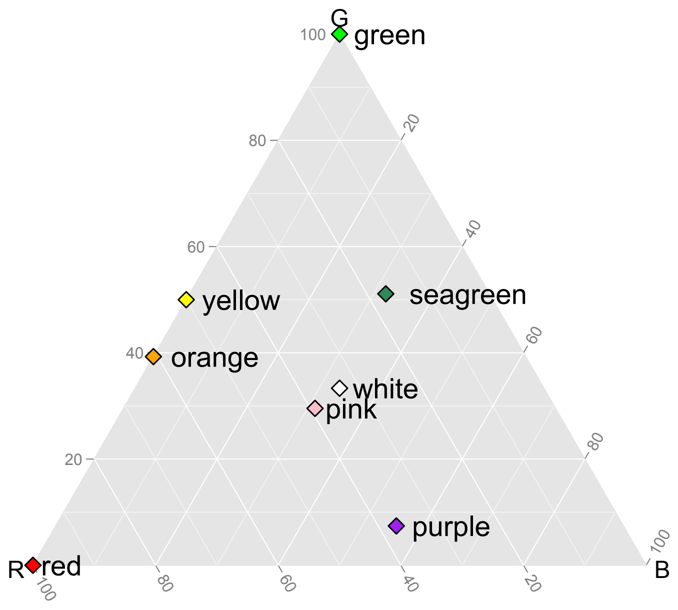
```
figvl4 <- fig_sun + geom_vline(xintercept = range(PAR()), linetype = "dashed") +
    annotate_waveband(PAR(), "text", y = 0.4, size = 10, colour = "black")
figvl4
```



14.8 Task: ploting colours in Maxwell's triangle

Given a color definition, we can convert it to RGB values by means of R's function `col2rgb`. We can obtain a color definiton 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 facetting function normally in most cases. Of course, being a ternary plot, the easthetics `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
```


G
green
100
80
60
40
20
yellow
orange
seagreen
white
pink
purple
R
red
B
20
40
60
80
100

CHAPTER 15

Calibration

Abstract

In this chapter we explain how to .

15.1 Task:

CHAPTER 16

Simulation

Abstract

In this chapter we explain how to .

16.1 Task:

CHAPTER 17

Measurement

Abstract

In this chapter we explain how to .

17.1 Task:

CHAPTER 18

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 sppectra is minimized by uisng 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.

18.1 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
```

18.2 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 especific 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.

18.3 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()`.

18.4 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.

18.5 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 defualts can speed up calculations when the same calculations are done repeatedly on spectra measured at exactly the same wavelengths, something which is usual when analysing 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 0.176 ms to 0.177 ms, by -0.46% 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 uisng an unweighted waveband the cache reduces the median execution time from 0.282 ms to 0.174 ms, by 38%.

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 0.554 ms to 0.197 ms, by 64%.

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 0.176 ms to 0.13 ms, by 26%.

Using stored wavebands

Saving a waveband object and reusing it, can give an additonal 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 0.135 ms to 0.117 ms, by 13%.

Saving a waveband object that uses weighting and reusing it, gives an additonal 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 0.145 ms to 0.116 ms, by 20%.

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 discountinuities in the slope require several hinges, while un 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 0.177 ms to 0.65 ms, by a factor of 3.6705.

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 0.19 ms to 0.663 ms, by a factor of 3.4936.

18.6 Overall speedup 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 0.996 ms, accepting all defaults yields a median execution time 0.467 ms, and disabling all checks, optimizations for precision and enabling all optimizations for speed yields a median execution time of 0.105, 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 densily measured data the effect of not using hinges becomes even smaller. In contrast, with data measured ar wider wvaelength 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 1.1 ms, accepting all defaults yields a median execution time 0.553 ms, and disabling all checks, optimizations for precision and enabling all optimizations for speed yields a median execution time of 0.111, 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 densily measured data the effect of not using hinges becomes even smaller. In contrast, with data measured ar wider wvaelength 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.

```r
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 =
```

When using split_irradiance, the cache reduces the median execution time from 0.967 ms to 0.656 ms, by 32%.

Using hinges slows down calculations:

```r
res1 <- microbenchmark(split_photon_irradiance(w.length, s.e.irrad,
    c(400, 500, 600, 700), use.cache = TRUE), times = 100L, control = list(warmup =
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 0.675 ms to 1.24 ms, by a factor of 1.8371. 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.

```r
res1 <- microbenchmark(split_photon_irradiance(w.length, s.e.irrad,
    c(400, 500, 600, 700), use.cache = TRUE), times = 100L, control = list(warmup =
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 0.679 ms to 0.611 ms, by 10%.

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.

18.7 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 rewritting these parts in a compiled language like C or C++, or by use of a more efficient calcualtion algorithm. A detailed discussion is outside the scope of this handbook, so only a brief example will be shown here.

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

Part III

Appendixes

APPENDIX A

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

0.03 * 2

## [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 -0.1  0.0
## [12]  0.1  0.2  0.3  0.4  0.5  0.6  0.7  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. If you are using RStudio, this same information is visible in the Environment pane.

```
z <- numeric(0)
z

## numeric(0)

ls()

##  [1] "a"                  "b"
##  [3] "base"               "black_body_peak_wl"
##  [5] "black_body_spectrum" "c"
##  [7] "cie"                "CIE"
##  [9] "colCount"           "colours"
## [11] "d"                  "fig_sun"
## [13] "figv2"              "figvl"
## [15] "figvl3"             "figvl4"
## [17] "gen.g"              "h"
## [19] "incl_apdx"          "incl_chaps"
## [21] "incl_ckbk"          "j"
## [23] "k.wein"                "kB"
## [25] "maxwell.tern"          "med1"
## [27] "med2"                  "med3"
## [29] "my.colors"             "my_version"
## [31] "myCIE"                 "myPAR"
## [33] "photobio.cache"        "photobioPhy.cache"
## [35] "Q.RGB"                 "red"
## [37] "remaining"             "res1"
## [39] "res2"                  "res3"
## [41] "rgb.values"            "rowCount"
## [43] "RowSize"               "s.e.irrad"
## [45] "s.q.irrad"             "test.data"
## [47] "toy.wb"                "uvb"
## [49] "wb1"                   "wb2"
## [51] "wl"                    "z"

rm(z)
z

## Error:  object 'z' not found

ls()

##  [1] "a"                     "b"
##  [3] "base"                  "black_body_peak_wl"
##  [5] "black_body_spectrum"   "c"
##  [7] "cie"                   "CIE"
##  [9] "colCount"              "colours"
## [11] "d"                     "fig_sun"
## [13] "figv2"                 "figvl"
## [15] "figvl3"                "figvl4"
## [17] "gen.g"                 "h"
## [19] "incl_apdx"             "incl_chaps"
## [21] "incl_ckbk"             "j"
## [23] "k.wein"                "kB"
## [25] "maxwell.tern"          "med1"
## [27] "med2"                  "med3"
## [29] "my.colors"             "my_version"
## [31] "myCIE"                 "myPAR"
## [33] "photobio.cache"        "photobioPhy.cache"
## [35] "Q.RGB"                 "red"
## [37] "remaining"             "res1"
## [39] "res2"                  "res3"
## [41] "rgb.values"            "rowCount"
## [43] "RowSize"               "s.e.irrad"
## [45] "s.q.irrad"             "test.data"
## [47] "toy.wb"                "uvb"
## [49] "wb1"                   "wb2"
## [51] "wl"
```

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, 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 smallest 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
```

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!

```
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

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 & FALSE & NA

## [1] FALSE

TRUE && FALSE && NA

## [1] FALSE

TRUE && TRUE && NA

## [1] NA

a & b & c(NA, NA)

## [1]    NA FALSE
```

A.4 Comparison operators

Comparison operators yield as a result logical values.

```
1.2 > 1

## [1] TRUE

1.2 >= 1

## [1] TRUE

1.2 == 1  # be aware that here we use two = symbols

## [1] FALSE

1.2 != 1

## [1] TRUE

1.2 <= 1

## [1] FALSE

1.2 < 1

## [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  TRUE  TRUE
## [10]  TRUE

a < 5

##  [1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE
## [10] FALSE

a == 5

##  [1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
## [10] FALSE

all(a > 5)

## [1] FALSE

any(a > 5)

## [1] TRUE

b <- a > 5
b

##  [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE
## [10]  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  TRUE  TRUE
## [10]  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 FALSE FALSE
## [10] FALSE

is.na(c)

##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [10] 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 91.

```
1e+20 == 1 + 1e+20

## [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  # may not always work

##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [10] FALSE

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

##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [10] FALSE

sin(pi) == 0  # angle in radians, not degrees!

## [1] FALSE

sin(2 * pi) == 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"
```

Skip this on first read, but look at this again later because it can be useful in some cases.

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 ‘scape sequences’. There are several of them, and here we will show just a few.

```
c <- "abc\ndef\txyz"
c

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

Above, you will not see any effect of these escapes: `\n` represents ‘new line’ and `\t` means ‘tab’ (tabulator). These work only in some contexts, but they can be useful for example when one wants to split an axis-label in a plot into two 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(3e+10)

## [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]`. (Some other computer languages use indexes that start at zero.)

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

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

a[]

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

a[numeric(0)]  # a bit tricky

## character(0)

a[NA]

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

a[c(1, NA)]

## [1] "a" NA

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"

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

Another way of indexing, which is very handy, but not available in most other computer 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 one vector is short 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  TRUE  TRUE
## [10]  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 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 staements contained in the file will appear in the console. The commands themselves are not shown (the sourced file is not echoed) and the results will not be printed unless you include an explicit `print` command. This also applies in many cases also to plots. A fig created with `ggplot` needs to be printed if we want to see it when the script is run.

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

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

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

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

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

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

B.3 How to write a script?

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

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

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

If ones mostly playing around Then if one is using RStudio, one type statements at the console prompt. As you should know by now, everything you run at the console is saved to the “History”. In RStudio the History is displayed in its own pane, and in this pane one can select any previous statement and by pressing a single having copy and pasted to either the console prompt, or the cursor position in the file visible in the editor. In this way one can build a script by copying and pasting from the history to your script file the bits that have worked as you wanted.

B.4 The need to be understandable to people

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

How does one achieve an understandable script or program?

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

B.5 Exercises

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

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

B.6 Functions

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

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

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

```r
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.

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

## [1] 1
```

Any result that needs to be made available outside the funtion 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.

```r
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 usefull function: a fucntion 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

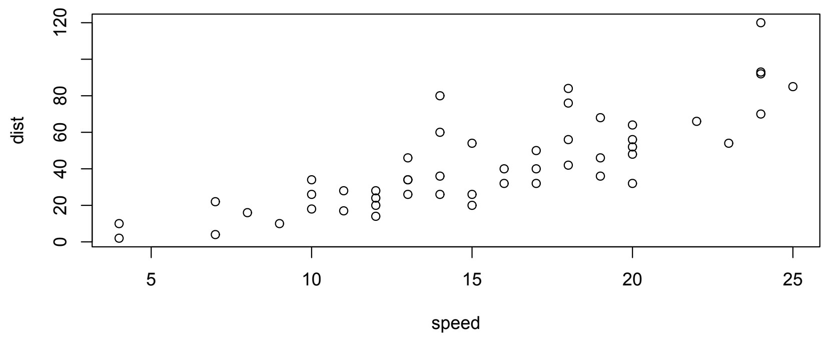
The built-in generic function `plot` can be used to plot data. It is a generic function, that has suitable methods for different kinds of objects.

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)
```

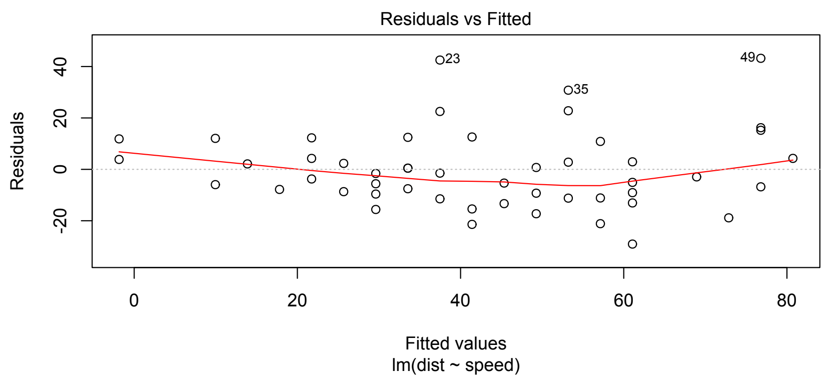


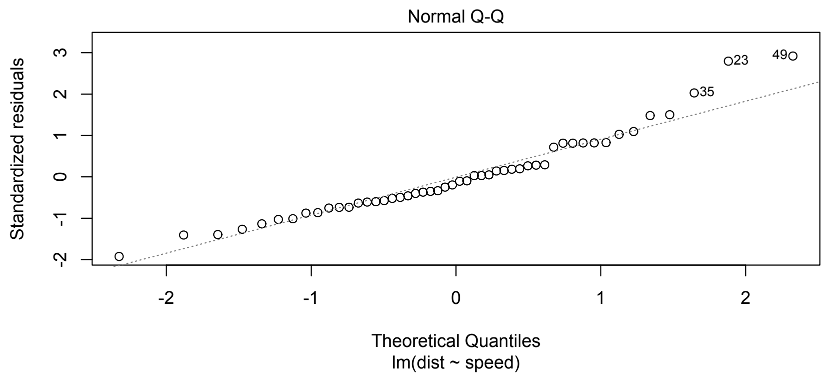
Fitting linear models

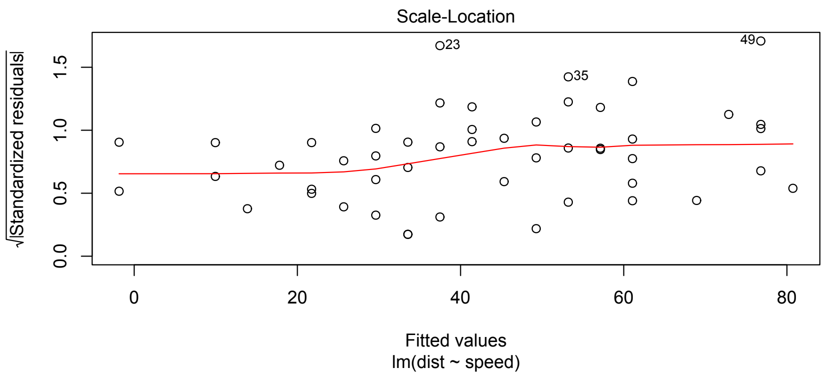
Regression

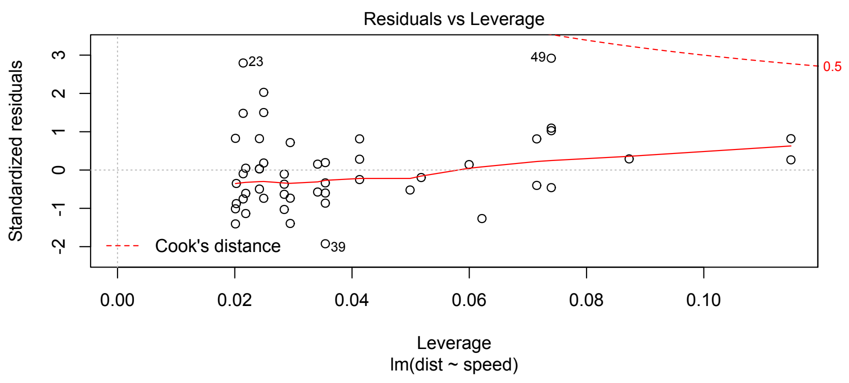
The R function `lm` is used next to fit a linear regression.

```
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 ***
## ---
## 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
35
49
40
20
0
-20
0
20
40
60
80

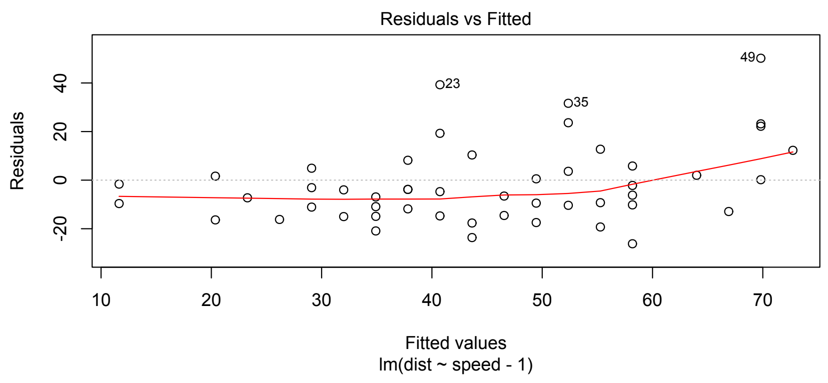

Normal Q-Q
Standardized residuals
Theoretical Quantiles
lm(dist ~ speed)
23
35
49
3
2
1
0
-1
-2
-2
-1
0
1
2

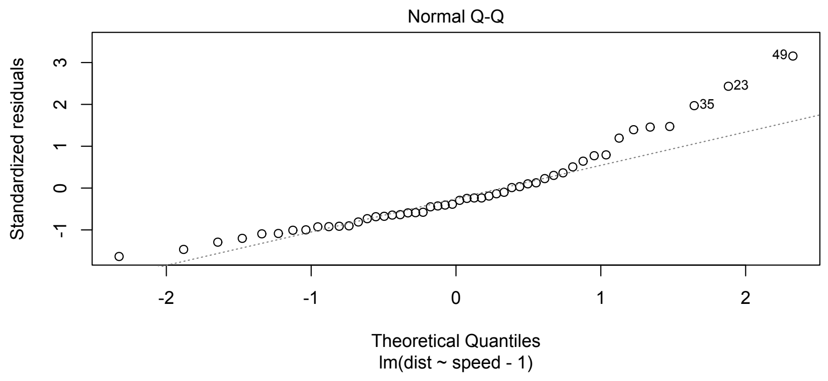

Scale-Location
$\sqrt{|\text{Standardized residuals}|}$
Fitted values
lm(dist ~ speed)
23
35
49
1.5
1.0
0.5
0.0
0
20
40
60
80

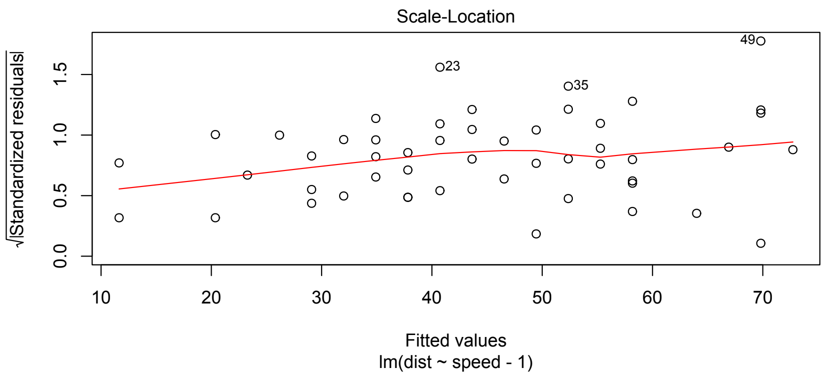


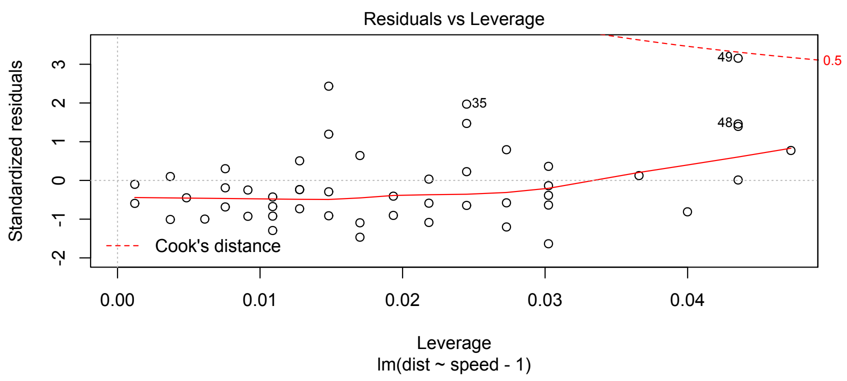
Let's look at each step separately: `dist   speed` is the specification of the model to be fitted. The intercept is always implicitly included. To 'remove' this implicit intercept from the earlier model we can use `dist   speed - 1`.

```
fm2 <- lm(dist ~ speed - 1, data = cars)  # we fit a model, and then save the resul
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
```


Residuals vs Fitted
Residuals
Fitted values
lm(dist ~ speed - 1)
49
23
35
40
20
0
-20
10
20
30
40
50
60
70

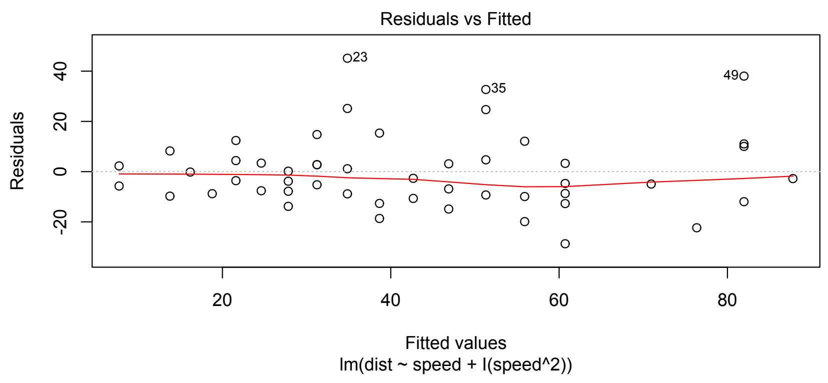

Normal Q-Q
Standardized residuals
Theoretical Quantiles
lm(dist ~ speed - 1)
49
23
35
3
2
1
0
-1
-2
-1
0
1
2

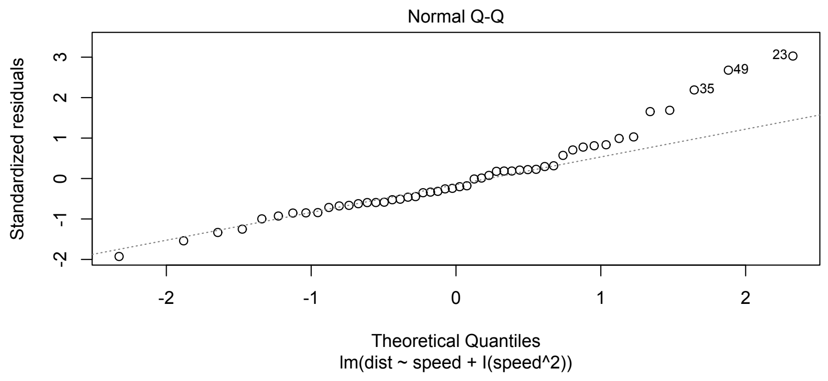

Scale-Location
√|Standardized residuals|
Fitted values
lm(dist ~ speed - 1)
49
23
35
1.5
1.0
0.5
0.0
10
20
30
40
50
60
70

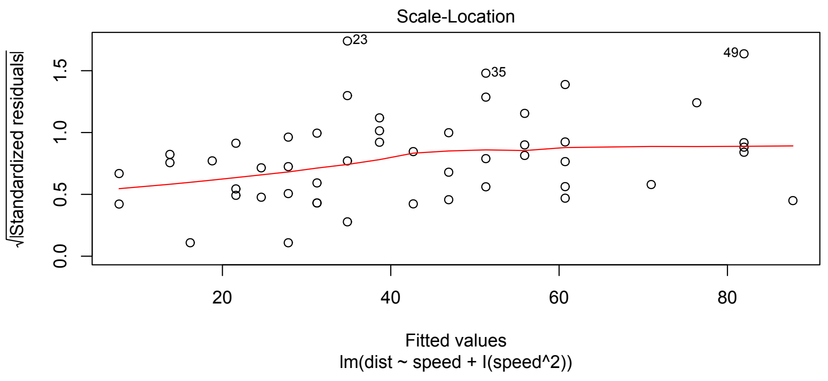


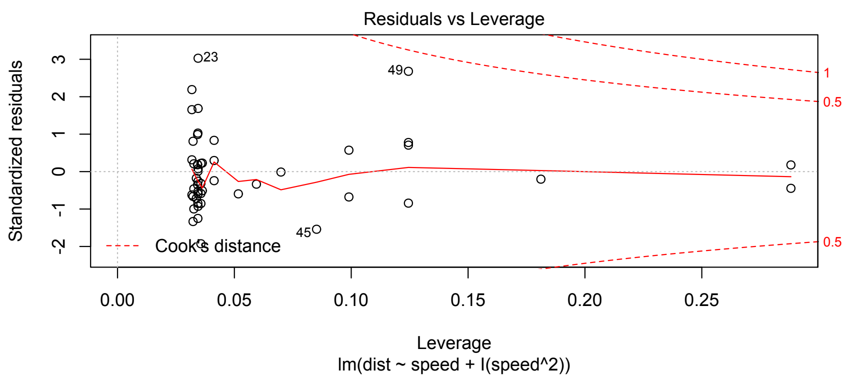
We now we fit a second degree polynomial.

```
fm3 <- lm(dist ~ speed + I(speed^2), data = cars)  # we fit a model, and then save t
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
```


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


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


Scale-Location
√|Standardized residuals|
Fitted values
lm(dist ~ speed + I(speed^2))
23
35
49



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
if (a < 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?

```r
# 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!.

```r
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 programmes 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.

```r
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.

```r
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.

```r
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.