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).

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"))
```

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

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.

```r
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:

```
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

Unweighted irradiance

Abstract

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

6.1 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 ($Q(\lambda)$) in $\mathrm{W\,m^{-2}\,nm^{-1}}$ and the corresponding wavelengths (λ) in nm.

$$Q_{\lambda_1<\lambda<\lambda_2} = \int_{\lambda_1}^{\lambda_2} E(\lambda)\,\mathrm{d}\,\lambda \tag{6.1}$$

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
##         268.9
```

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
```

6.2 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.

The energy of a quantum of radiation in a vacuum, q, depends on the wavelength, λ, or frequency[1], ν,

$$q = h \cdot \nu = h \cdot \frac{c}{\lambda} \tag{6.2}$$

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 (6.2) 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

$$q' = h' \cdot \nu = h' \cdot \frac{c}{\lambda} \tag{6.3}$$

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

[1]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 6.2.

radiation at 300 nm has about 400 $\mathrm{kJ\,mol^{-1}}$, therefore, 1 µmol photons has 0.4 J. Equations 6.2 and 6.3 are valid for all kinds of electromagnetic waves.

Combining equations 6.1 and 6.3 we obtain:

$$Q_{\lambda_1<\lambda<\lambda_2} = \int_{\lambda_1}^{\lambda_2} E(\lambda)\,\frac{h'\cdot c}{\lambda}\mathrm{d}\,\lambda \tag{6.4}$$

Let's assume that we want to calculate photosynthetically active radiation (PAR) photon irradiance. In this example we will use example data for sunlight.

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

##       PAR
## 0.0008938
```

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

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

##   PAR
## 893.8
```

PAR() is predefined in package photobiologyVIS as a convenience function, see section 6.1 for an example with arbitrary values for λ_1 and λ_2.

6.3 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:

$$I_{\lambda_1<\lambda<\lambda_2} = \int_{\lambda_1}^{\lambda_2} Q_\lambda\,\frac{\lambda}{h'\cdot c}\mathrm{d}\,\lambda \tag{6.5}$$

And the code[2]:

```
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 6.1, and the code is:

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

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

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

6.4 Task: irradiances for more than one waveband

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

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

6.5 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 21

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

```r
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`.

```r
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.ISO"

color(red)

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

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

```r
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
```

6.6 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
```

6.7 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[3]:

$$\zeta = \frac{Q_{655\mathrm{nm}<\lambda<665\mathrm{nm}}}{Q_{725\mathrm{nm}<\lambda<735\mathrm{nm}}} \tag{6.6}$$

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
```

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

6.8 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.

6.9 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 $\mathrm{mol\,W^{-1}}$. In this case only one waveband definition is used to calculate the quotient:

$$\bar{q}' = \frac{Q_{\lambda_1<\lambda<\lambda_2}}{E_{\lambda_1<\lambda<\lambda_2}} \qquad (6.7)$$

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

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

For obtaining the same quotient in $\mathrm{\mu mol\,W^{-1}}$ we just need to multiply by 10^6. We can use such a multiplier to convert E [$\mathrm{W\,m^{-2}}$] into Q [$\mathrm{\mu mol\,m^{-2}\,s^{-1}}$], 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
```

6.10 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
```

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
```

Although 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
```

6.11 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 6.10 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 7

Weighted and effective irradiance

Abstract

In this chapter we explain how to .

7.1 Task:

CHAPTER 8

Colour

Abstract

In this chapter we explain how to .

8.1 Task:

CHAPTER 9

Photoreceptors

Abstract

In this chapter we explain how to .

9.1 Task:

CHAPTER 10

Radiation sources

Abstract

In this chapter we explain how to .

10.1 Task:

CHAPTER 11

Filters

Abstract

In this chapter we explain how to .

11.1 Task:

CHAPTER 12

Plotting spectra

Abstract

In this chapter we explain how to .

12.1 Task:

CHAPTER 13
Calibration

Abstract

In this chapter we explain how to .

13.1 Task:

CHAPTER 14

Simulation

Abstract

In this chapter we explain how to .

14.1 Task:

CHAPTER 15

Measurement

Abstract

In this chapter we explain how to .

15.1 Task:

CHAPTER 16

Optimizing performance

Abstract

In this chapter we explain how to .

16.1 Task:

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] "c"                  "d"
##  [5] "incl_apdx"          "incl_chaps"
##  [7] "incl_ckbk"          "my_version"
##  [9] "photobio.cache"     "photobioPhy.cache"
## [11] "Q.RGB"              "red"
## [13] "uvb"                "wb1"
## [15] "wb2"                "z"

rm(z)
z
```

```
## Error:  object 'z' not found
ls()

##  [1] "a"                    "b"
##  [3] "c"                    "d"
##  [5] "incl_apdx"            "incl_chaps"
##  [7] "incl_ckbk"            "my_version"
##  [9] "photobio.cache"       "photobioPhy.cache"
## [11] "Q.RGB"                "red"
## [13] "uvb"                  "wb1"
## [15] "wb2"
```

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 54.

```
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:

```r
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.

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

## [1] 12
```

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

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

## [1] 1
```

Any result that needs to be made available outside the 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.

```
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

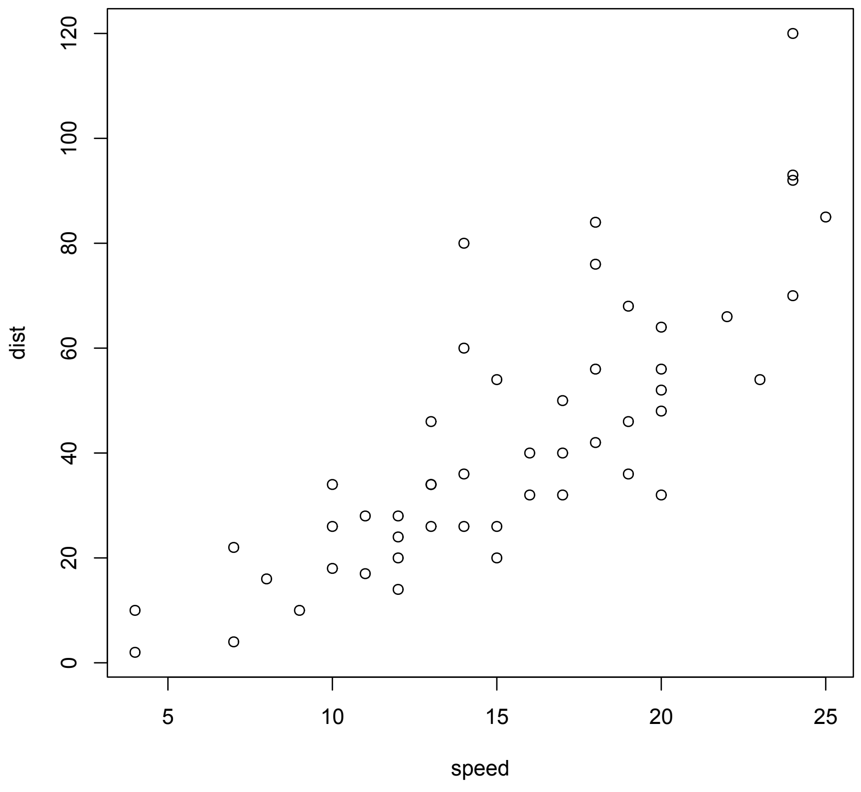
The built-in generic function `plot` can be used to plot data. It is a generic function, that has suitable methods for different kinds of objects.

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)
```



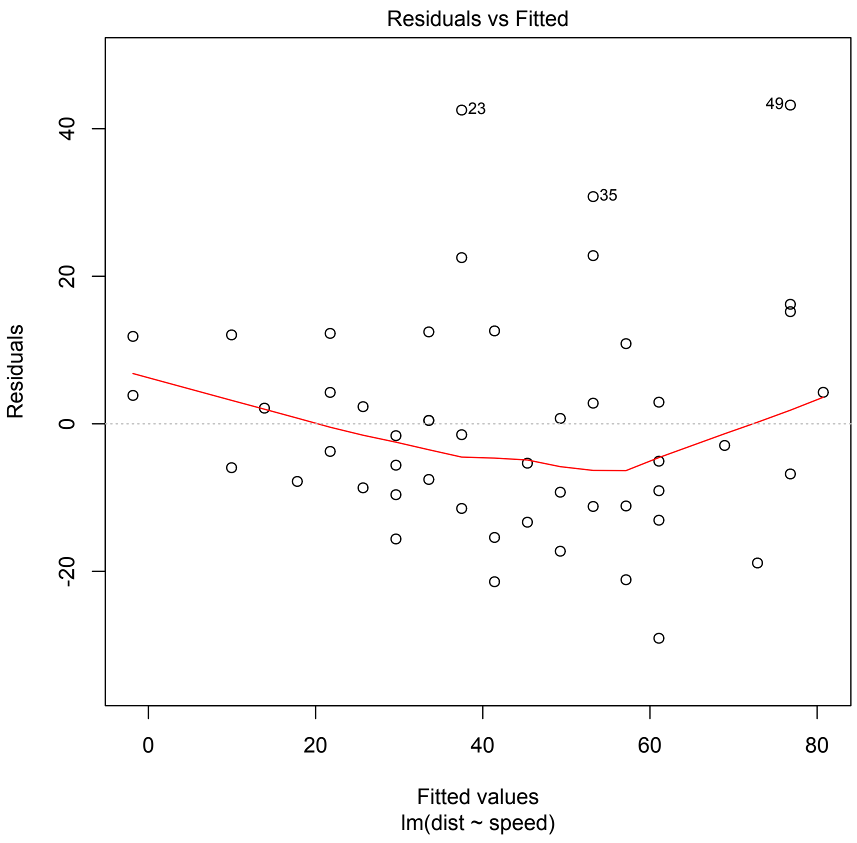
Fitting linear models

Regression

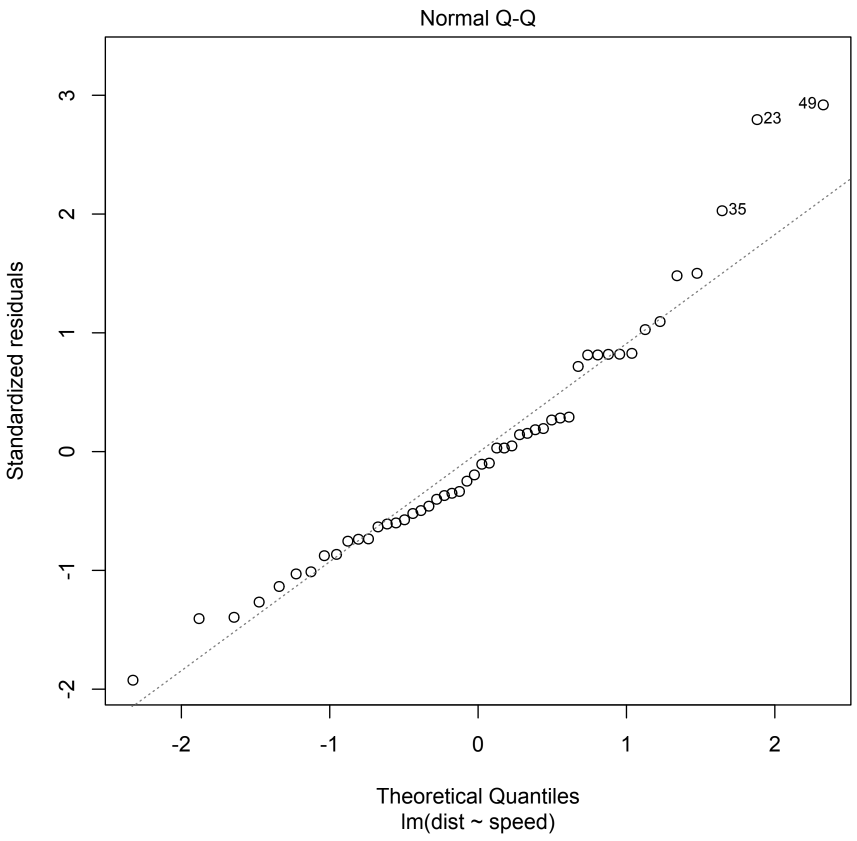
The R function `lm` is used next to fit a linear regression.

```
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:
```

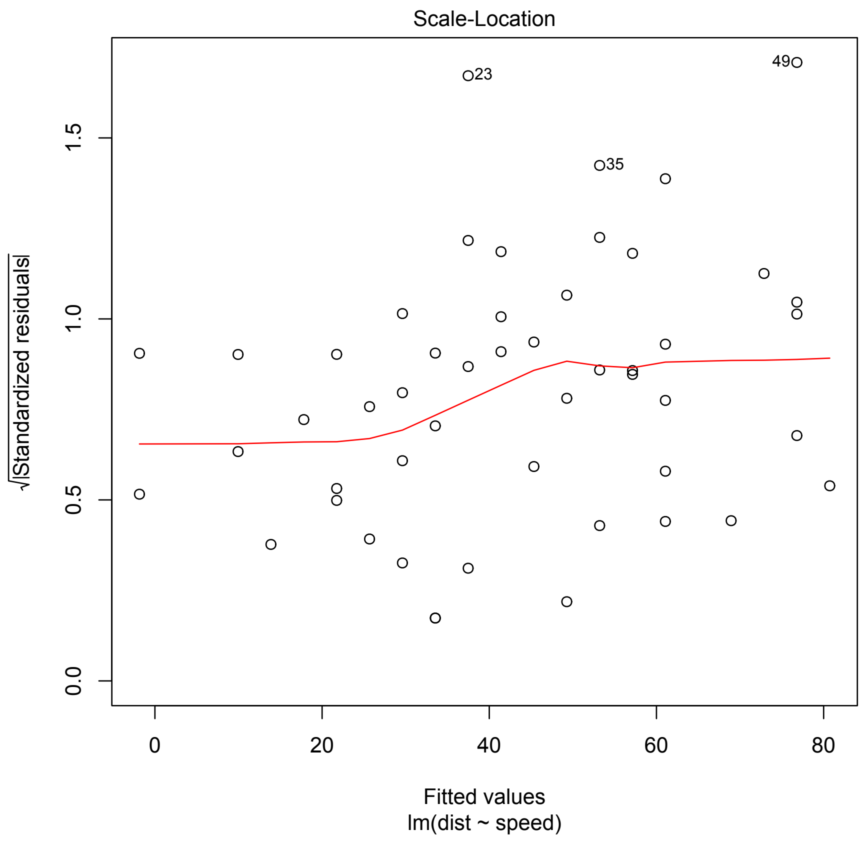
```
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 15.4 on 48 degrees of freedom
## Multiple R-squared:  0.651,Adjusted R-squared:  0.644
## F-statistic: 89.6 on 1 and 48 DF,  p-value: 1.49e-12

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
```

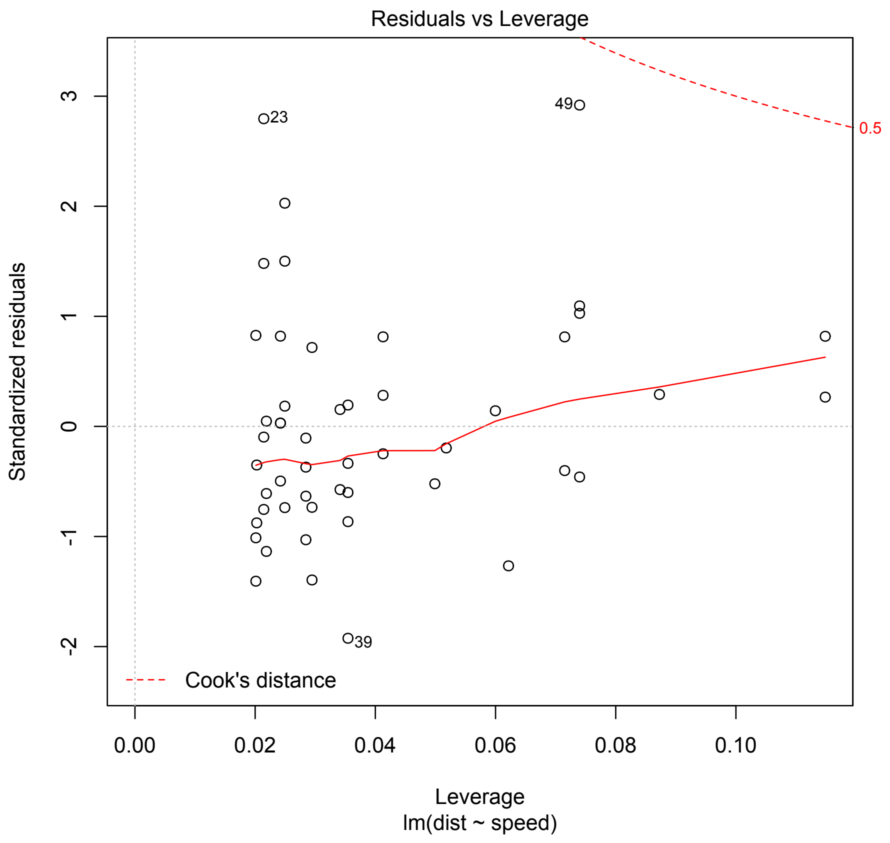


Residuals vs Fitted
Residuals
Fitted values
lm(dist ~ speed)
23
49
35



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

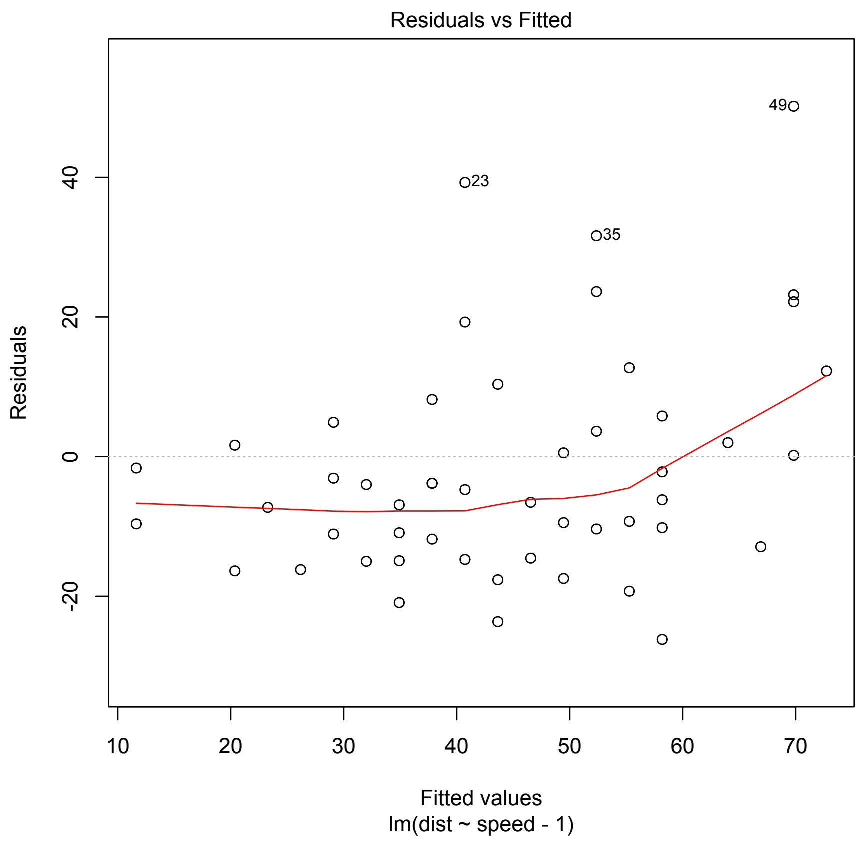

Scale-Location
√|Standardized residuals|
0.0
0.5
1.0
1.5
0
20
40
60
80
23
35
49
Fitted values
lm(dist ~ speed)

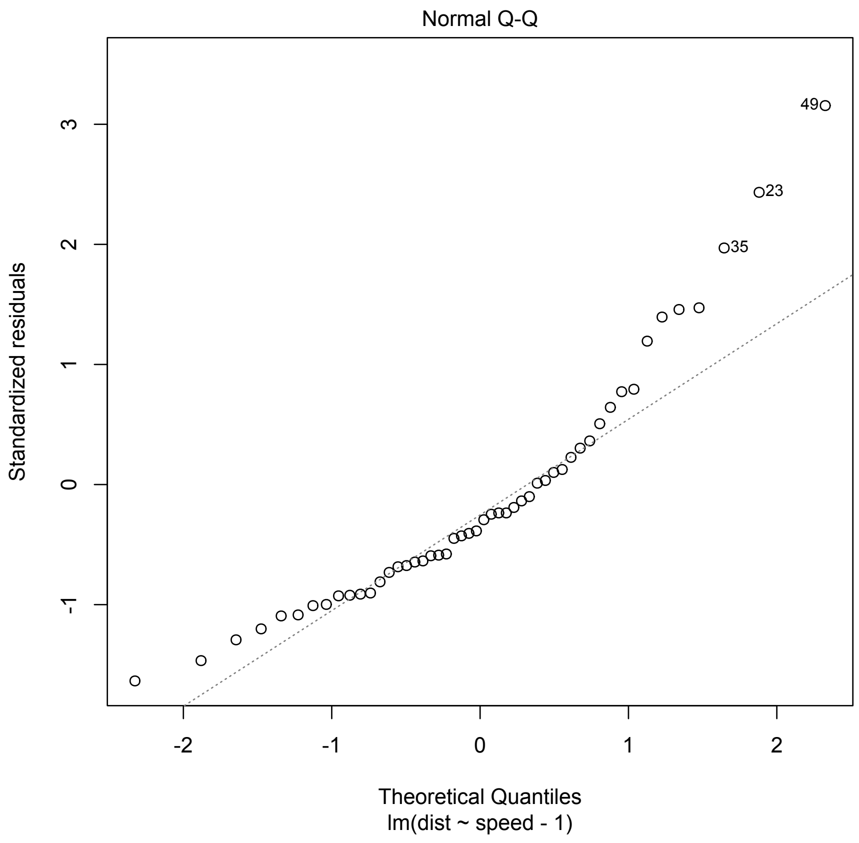


Let's look at each step separately: `dist    speed` is the specification of the model to be fitted. The intercept is always implicitly included. To 'remove' this implicit intercept from the earlier model we can use `dist    speed - 1`.

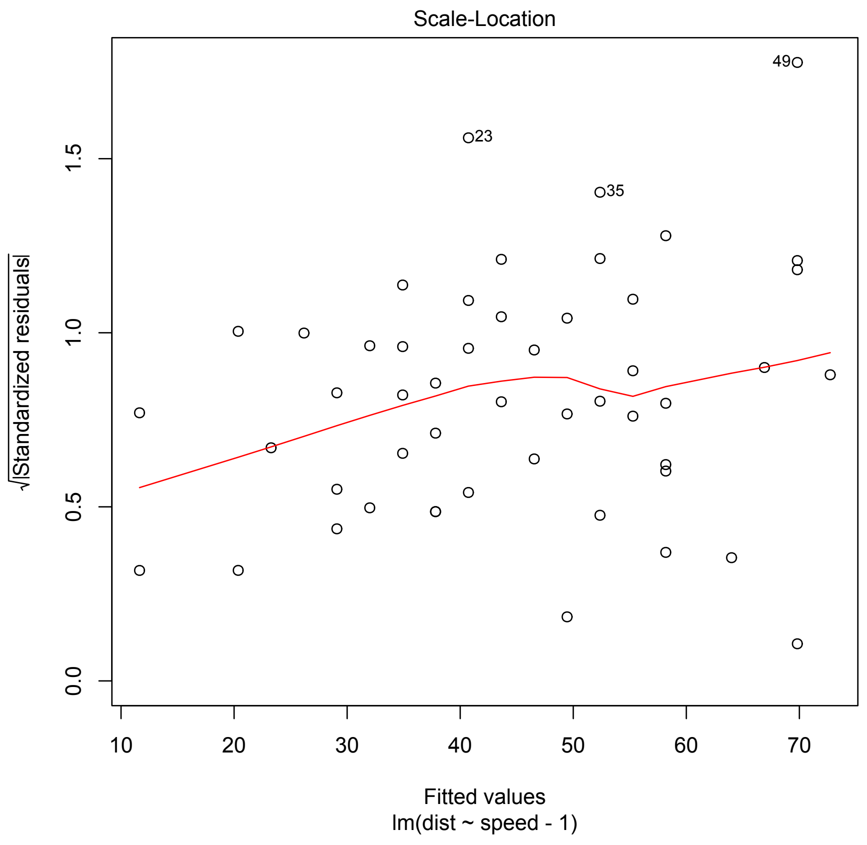
```
fm2 <- lm(dist ~ speed - 1, data = cars)  # we fit a model, and then save the resul
plot(fm2)  # we produce diagnosis plots
summary(fm2)  # we inspect the results from the fit

##
## 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
##
```

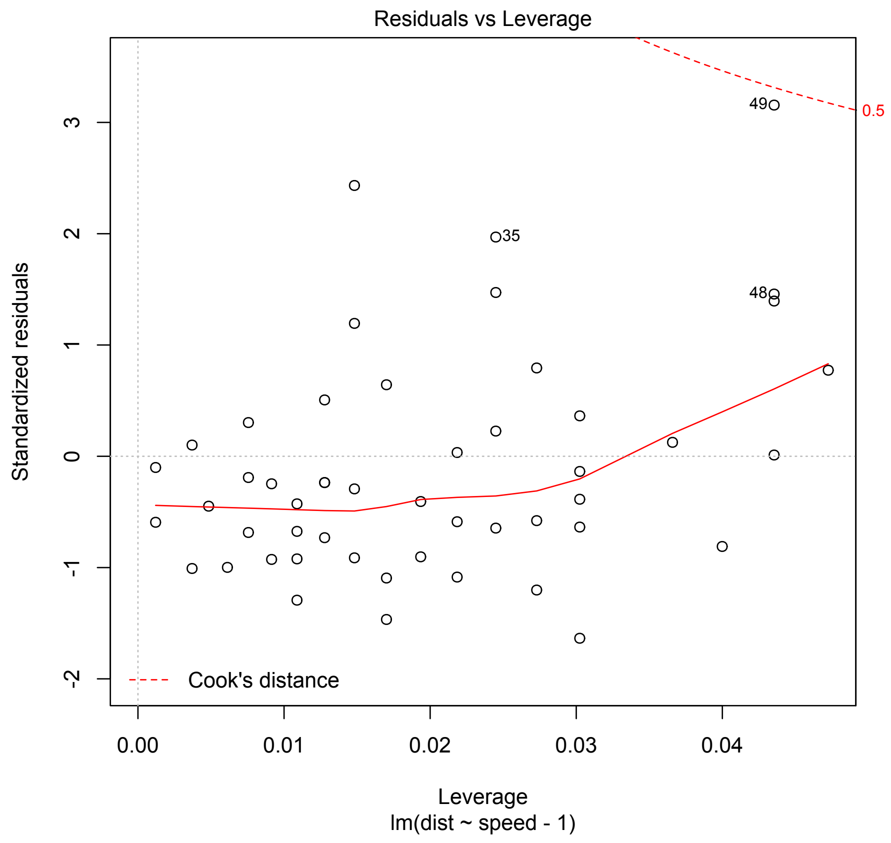
```
## Residual standard error: 16.3 on 49 degrees of freedom
## Multiple R-squared:  0.896,Adjusted R-squared:  0.894
## F-statistic:  423 on 1 and 49 DF,  p-value: <2e-16

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
```





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

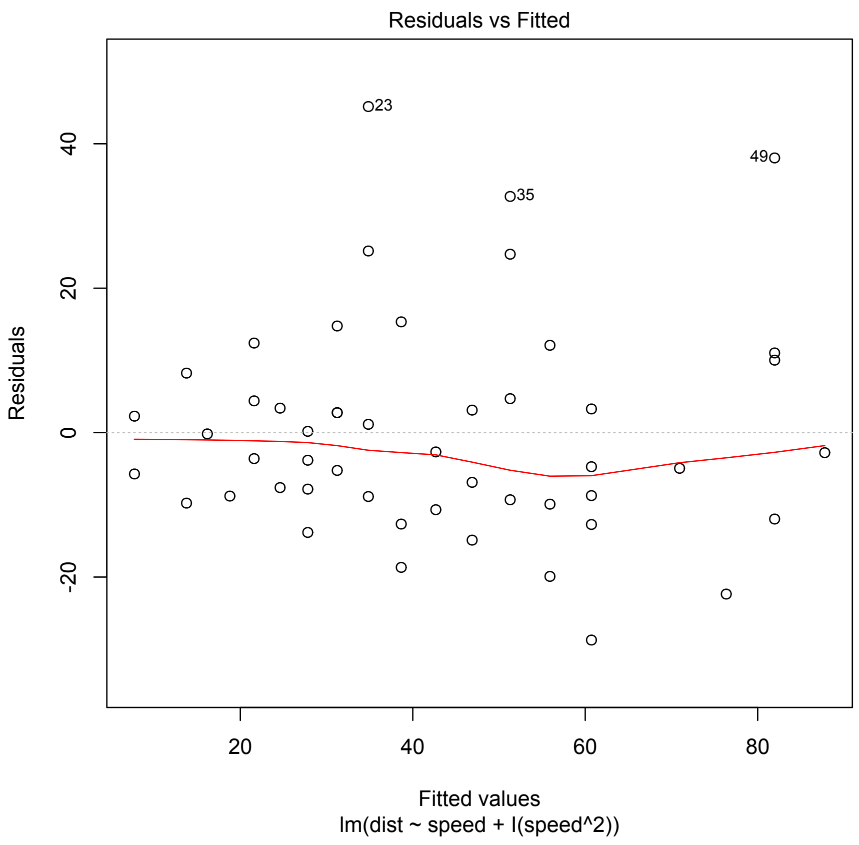

Scale-Location
√|Standardized residuals|
0.0
0.5
1.0
1.5
10
20
30
40
50
60
70
Fitted values
lm(dist ~ speed - 1)
49
23
35

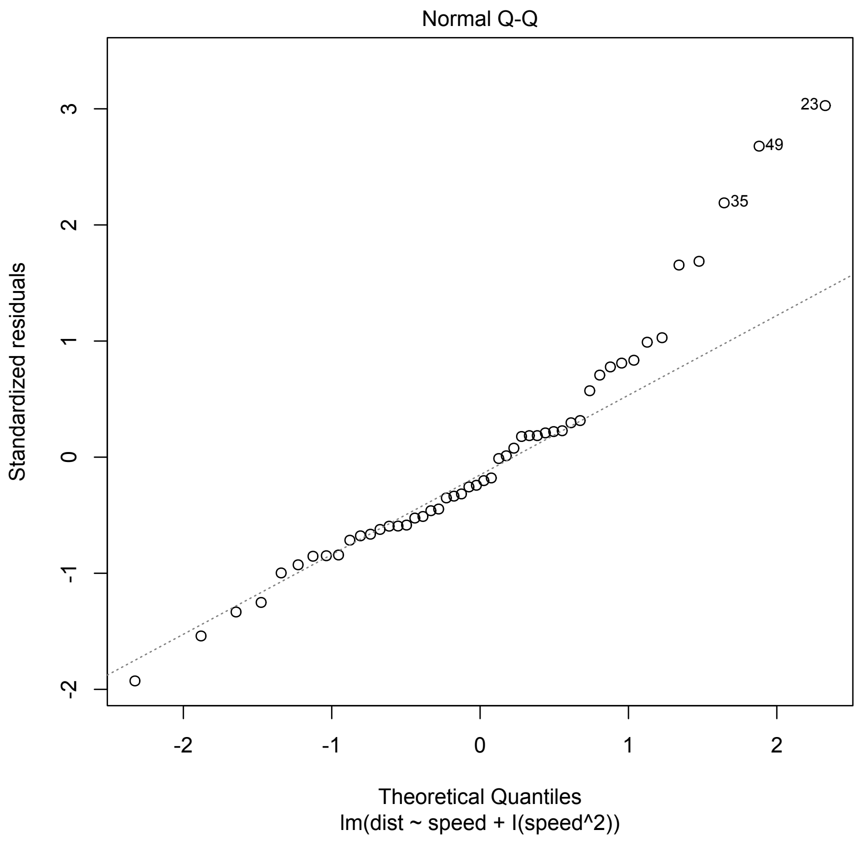


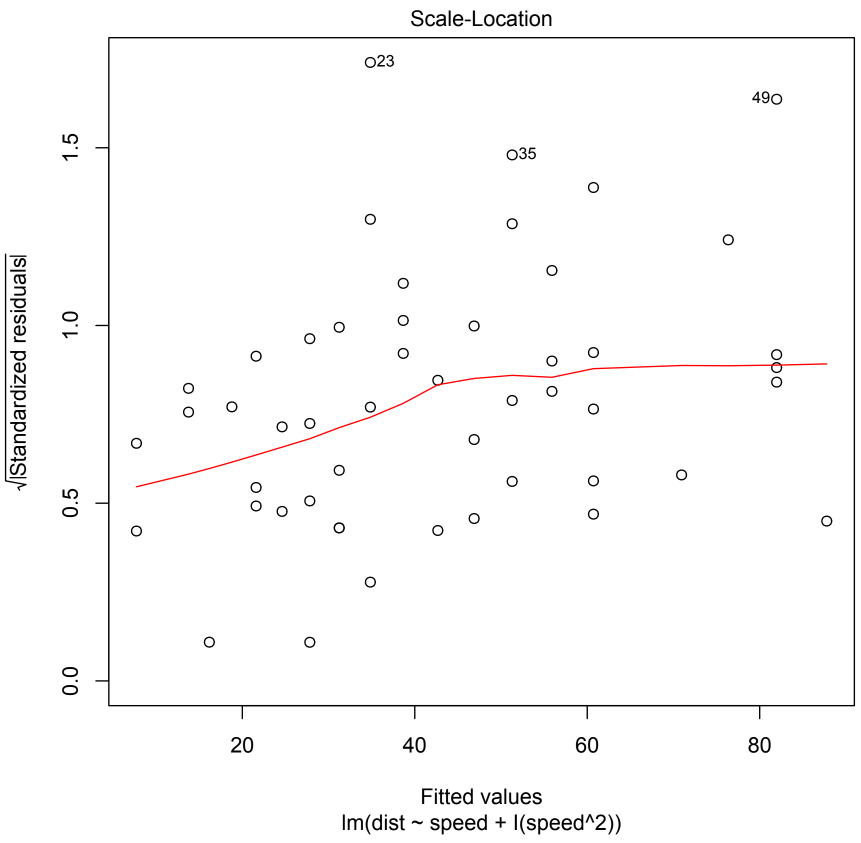
We now we fit a second degree polynomial.

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

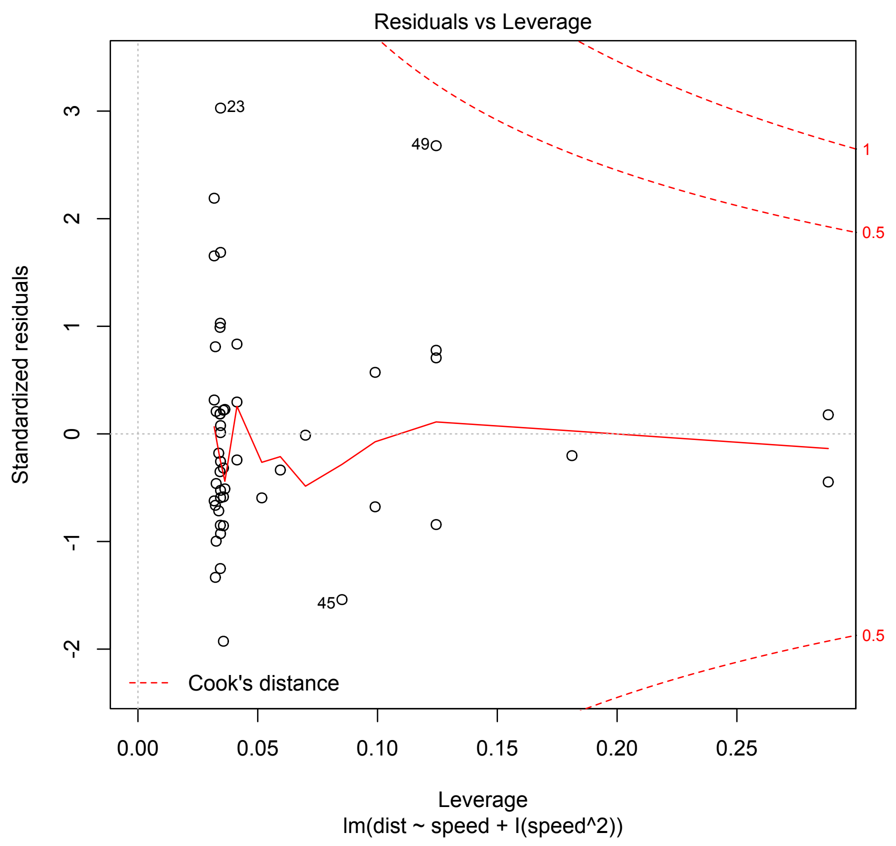
```
## Analysis of Variance Table
##
## Response: dist
##             Df Sum Sq Mean Sq F value  Pr(>F)
## speed        1  21185   21185    92.0 1.2e-12 ***
## I(speed^2)   1    529     529     2.3    0.14
## Residuals   47  10825     230
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```




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



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



We can also compare the two models.

```
anova(fm2, fm1)

## Analysis of Variance Table
##
## Model 1: dist ~ speed - 1
## Model 2: dist ~ speed
##   Res.Df   RSS Df Sum of Sq    F Pr(>F)
## 1     49 12954
## 2     48 11354  1      1600 6.77  0.012 *
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

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

```
anova(fm2, fm1, fm3)

## Analysis of Variance Table
##
## Model 1: dist ~ speed - 1
## Model 2: dist ~ speed
## Model 3: dist ~ speed + I(speed^2)
##   Res.Df   RSS Df Sum of Sq    F Pr(>F)
## 1     49 12954
## 2     48 11354  1      1600 6.95  0.011 *
## 3     47 10825  1       529 2.30  0.136
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

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

B.8 Control of execution flow

Conditional execution

Non-vectorized

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

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

```
print("A")

## [1] "A"

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

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

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

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

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

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

```r
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.

```r
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.

```r
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 (;).

```r
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.

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

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

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

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

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

Nesting

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Full vectorization can be achieved with `apply` functions.

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

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

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

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

B.9 Packages

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

```
library(graphics)
```

Currently there are thousands of packages available. The most reliable source of packages is CRAN, as only packages that pass strict tests and are actively maintained are included. In some cases you may need or want to install less stable code, and this is also possible.

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

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

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

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

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

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

SEM  # the function we defined earlier

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

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