Scientific computing in R

Karline Soetaert and Filip Meysman
Royal Netherlands Institute of Sea Research (NIOZ)
Yerseke, The Netherlands
May 2013

Abstract

The principal goal of these lecture notes is to learn how to produce clever R scripts. These R scripts should support and facilitate your research activities, and more specifically, they should help with the computational aspects of your scientific work.

R (R Development Core Team 2011) is the open-source version of the language S. It is best known as a software environment that performs statistical analysis and graphics. However, R is so much more: it is a high-level language in which one can perform complex calculations, implement new methods, and make high-quality figures.

R has high-level functions to operate on matrices, perform numerical integration, advanced statistics,... which are easily triggered and which make it ideally suited for data-visualization, statistical analysis and mathematical modeling.

It is the aim of these lecture notes to make you acquainted with the R language. Part of these lecture notes are based on a book (Soetaert and Herman 2009) about ecological modelling in which R is extensively used for developing, solving and applying and models.

Keywords: Variables, Functions, Figures, Interpolation, Fitting, Roots, Ordinary differential equations, R.

---

1. Getting started in R

R programming basically involves two activities:

- *the development of an R script.* First you need to write down a suitable sequence of R commands, i.e. you tell the computer what it should do. To make sure that the computer understands these R commands, they should follow the syntax of the R programming language.

- *the execution of the R script.* Next the computer will execute the R commands line by line as specified in the R script, i.e. the computer will do what you have told it to do.

Because of this two-step process, you will typically use two software programs when working with R. Firstly, the *editor* is the program in which you write, edit and save your R scripts. Secondly, the *R console* is the program that effectively executes the commands in the script. The purpose of this first chapter is to get your R installation up and running, and at the same time, walk you through its most basic features.

1.1. The R console

The R console provides an integrated software environment for data manipulation, calculation and graphical display. Technically speaking, the R programming language is an interpreted language, where the R console acts as a command line interpreter. The R console is developed and maintained as an open source project (R Development Core Team 2011). The R console software (referred to as the "base distribution" in R terminology) is freely downloadable from the CRAN website:

`http://cran.r-project.org/`

CRAN is the acronym of "Comprehensive R Archive Network" and is a network of servers that hosts the code and documentation for R . On the CRAN site, go to `Download and Install R` and choose the link to your operating system (**Windows**, **MacOSX** or **Linux**).

This document assumes that you are operating under **Windows**. However, most of the material presented here is independent of the operating system. Accordingly, **MacOSX** or **Linux** users who have their R system running, will be able to directly use the material. To install R under **Windows**, click on the `base` link and subsequently `Download R 2.15.1 for Windows` (or some higher version of R).

On this same CRAN website, you will also find various accompanying documentation and other information. It is worthwhile to check out what's available. For example, the `R for Windows FAQ` is very helpful. In addition, there are many good introductions to programming in R that are available at the CRAN website. Our favorite is the R introduction by Petra Kuhnert and Bill Venables (Kuhnert and Venables 2005). Note however that this "introduction" contains more than 300 pages!

1.2. The R editor

When writing R scripts, it is most productive to do this in a special software application, called an integrated development environment (IDE). An IDE will provide you with a set of useful coding tools to help you develop R scripts. There are different free IDEs around on the internet, some of which are specific for R (e.g., Tinn-R), while others can also be used for other programming languages (e.g., Eclipse, Emacs). Recently, a new IDE has been made available, called **RStudio**, specifically designed for R. We recommend it because of it has an intuitive interface and helpful coding tools. It can be freely downloaded from the website:

`http://rstudio.org/`

If you first install the R base distribution, and subsequently **RStudio**, the latter will automatically have a proper connection with R (see the appendix for what to do if this does not happen).

If you launch **RStudio**, then four panels will appear - a screenshot is given below in figure 1. The R console is immediately started in the lower left panel. A script file can be opened in the editor panel on the top left. Note the colour coding (e.g. green = text strings, blue = reserved words). The upper right panel contains the workspace, which allows to see the content of the various R objects that you have defined. The lower right panel provides access to the file system and extension packages, and shows figures and help.

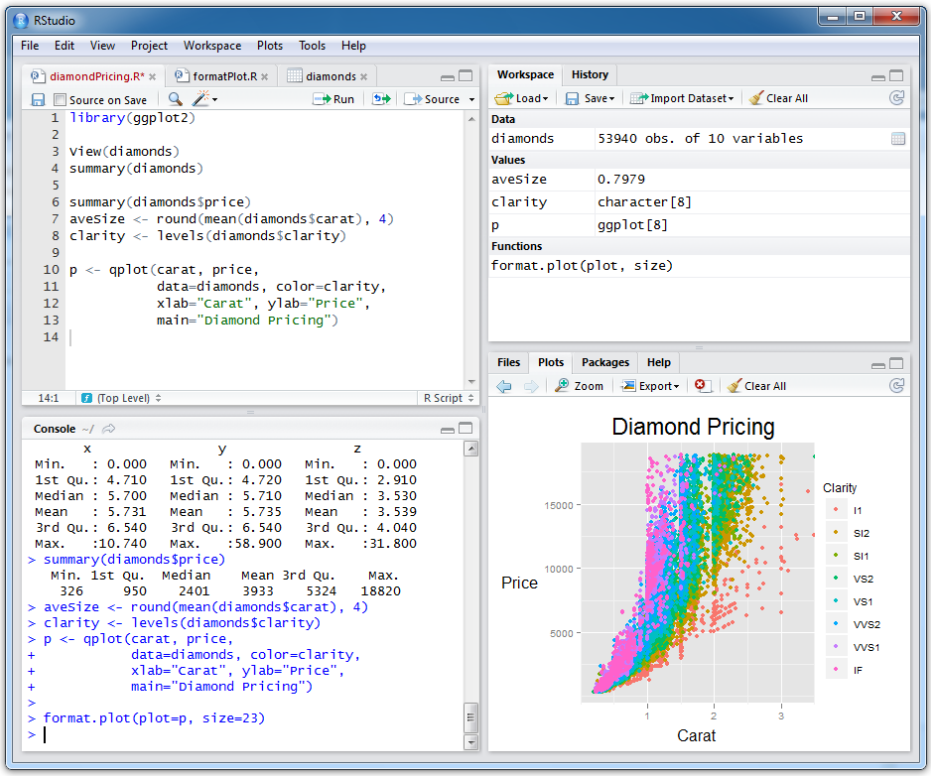


Figure 1: Screenshot of Rstudio, a freely downloadable development environment for R.

1.3. Installing extension packages

An R package essentially contains a set of functions that can perform certain tasks. When downloading R , one only installs the base distribution, which comes with a limited set of pre-installed packages. For specific applications, you will need to download and install additional packages. For this course, we will need:

- **deSolve**. Performs integration of differential equations (Soetaert, Petzoldt, and Setzer 2010).
- **rootSolve**. Finds the root of equations (Soetaert 2009).
- **scatterplot3d**. For 3-D graphics. (Ligges and Machler 2003)
- **AquaEnv**. Aquatic chemistry. (Hofmann, Soetaert, Middelburg, and Meysman 2010)
- **seacarb**. Aquatic chemistry. (Lavigne, Gattuso, Epitalon, Gentili, Hofmann, Orr, Proye, and Soetaert 2010)
- **marelac**. Functions and constants from the marine and lacustrine sciences (Soetaert, Petzoldt, and Meysman 2009).

Packages can be downloaded from the CRAN website. In **RStudio**, downloading packages can be done in the `Packages` menu of the lower right panel. Select `Install package(s)` and

type the name of the package. If you install package **marelacTeaching** then all other packages will be automatically installed as well.

Once installed, you can generate a list of all available packages, or load a package, or obtain the contents of a package by the following commands:

```
library()
library(deSolve)
library(help = deSolve)
help(package = deSolve)
```

2. First steps into R programming

In this chapter, we assume that you have little experience with programming. So we will start from the very basics and illustrate some basal programming techniques in R. More advanced programmers can quickly "browse" this chapter and move on to the next one. Note however that some features in R are unconventional as compared to other languages. If you have written software in other languages, the website *R programming for those coming from other languages* located at

http://www.johndcook.com/R_language_for_programmers.html

could be a useful resource. It gives a concise overview of those aspects in which R differs from other languages.

2.1. Command line execution in the console

The most basic way to excute a given R command is to simply type this command directly into the R console (the lower left panel in **RStudio**). In this case, one does not yet use the script editor (the upper left panel in **RStudio**). Let's try this with a silly example. First type the expression `4*2` after the so-called prompt (>) and subsequently press `Enter`. This is what will appear:

```
> 4*2
```

```
[1] 8
```

This illustrates how the execution of instructions in the R console essentially works by means of a question-and-answer model. One first enters the instruction after the prompt (>). Subsequently the computer carries out the instruction after pressing `Enter`. The answer from the computer is written on the next line. The `[1]` in front of the result is the standard way that R prints numbers and vectors. For the moment, we should not bother too much about this `[1]`. Later, when we work with vectors and matrices, the meaning of this notation will become clear.

The above example consists of a very simple arithmetic expression. Obviously, more complex expressions can be constructed using built-in mathematical functions like `sqrt()` (square root), `exp()` (exponential) and `log10()` (base 10 logarithm). Try:

```
> (2*sqrt(25) + log10(100)) * exp(-2)/2
```

```
[1] 0.8120117
```

In essence, we are using the R console here as a powerful scientific calculator. The crucial point is that we should use the proper R syntax when coding our mathematical expressions. For novice programmers, the struggle with the R syntax can be frustrating, but this is part of the learning process of any programming language.

Tips and tricks:

The <UP> and <DOWN> arrows on your keyboard can be used to navigate through previously typed sentences in the console panel. This way, it is easy to recycle earlier commands and implement small modifications (without the need to type the whole command from scratch).

2.2. Working with symbolic variables

Like any other programming language, one can use symbolic variables in R. The use of symbolic variables allows one to store intermediate results in the memory of the computer. This technique is one of the basic pillars of programming. Type the following:

```
> A <- 1
```

The two characters "`<-`" should be read as one arrow symbol representing the assignment operator. The command `A <- 1` thus creates a symbolic variable with the name `A` and assigns the value 1 to this variable. Note that R also allows to use "`=`" as the assigment operator, but this is not as flexible as "`<-`" (and therefore not recommended).

Note that the assignment has no immediate visible result. The instruction only creates the variable `A`, but does not print its content to the console. If we want to display the value `A` on the screen, we need to explicitly ask R to do so. This can be done in three ways: either by typing `A` on a new command line,

```
> A
```

```
[1] 1
```

by putting round brackets around the original expression,

```
> (A <- 1)
```

```
[1] 1
```

or by using the `print()` command.

```
> print(A)
```

```
[1] 1
```

The advantage of using symbolic variables is enormous. From now on, the object `A` has the value 1, and we can use this object in subsequent commands, such as:

```
> exp(2*A + 0.5)
```

```
[1] 12.18249
```

Names of variables can be chosen quite freely in R. They can consist of letters, digits and the dot symbol. Beware that some single letters (e.g. `c`, `F`, `T`) and names (e.g. `seq`, `rnorm`) have already been claimed by the R system. To avoid confusion, it is not a good idea to use these as variable names (although it is allowed and mostly it will not cause trouble). Similarly, names may not start with a digit, while names starting with a dot are special and should be avoided. The dot symbol is however often used to construct a composite name. For example the variable `O2.exp1` could represent the oxygen ("O2") concentration as measured in the first experiment ("exp1").

A final important note is that *names are case sensitive!* To see this, try:

```
> var <- 1; Var <- 2
> var + Var
```

```
[1] 3
```

The terms `Var` and `var` are treated as separate variables. Mistakes with capital and small letters are therefore common (watch out!). In the above code we have used a semicolon ";" to separate two R statements.

Note: In an R script, it is sometimes difficult to see the difference between the letter "l" (from "leo"), the capital letter "I"(from "Iris") and the number "1" (one), and also between the capital letter "O" (from "Octopus") the and number "0" (zero).

2.3. File management

During programming, the number of files (e.g. different versions of scripts) will rapidly mount, and so, it is important to stick to a proper "file hygiene". It is a good idea to create a separate file directory on your computer system for each R project. This makes it easy to save your work and retrieve it in later sessions.

R always uses a so-called *working directory* where it reads and writes files. When opening an R session, this working directory is automatically set (see `Tools:Options:General:Initial working directory` in **Rstudio**). This default value is typically not the desired project directory. Therefore, at the start of your R session, it is very convenient to have the working directory set correctly to the proper project directory.

This can be done in a number of ways. A first option is to select the project directory in the `Files` panel of **Rstudio** (the lower right panel), and then use `More:Set as working directory` button on the toolbar. Alternatively, one can also `Tools:Set working directory` from the menu in the editor panel. As a third option, one can also set the working directory directly in the Console (i.e. without using the menus in **Rstudio**). The command

```
getwd()
```

will give you the value of the current working directory. You can change this directory to a new value via

```
setwd("Path_to_MyProjectDirectory")
```

Important note: Pathnames in R are written with forward slashes ("/") , although in Windows, backslashes, (\), are used. For example, the working directory could be set as:

```
setwd("C:/R code/MyProjectDirectory")
```

2.4. Objects and the workspace

All the entities that R creates and manipulates are treated as *objects*, where each object has a given name and a certain content. The variable `A` as introduced above is an example of such an object. Whenever an object is created during an R session, its name and its content are stored in the memory of the computer. The collection of all objects currently stored is called the *workspace*. To display the names of all the objects that are currently stored in the workspace, one can use the function `ls()` (abbreviation for "list")

```
ls()
```

The value of all user-defined objects are displayed in the upper right panel of **Rstudio** under the tab `Workspace`. To remove one or more objects from the workspace, you can use the function `rm()` (abbreviation for "remove"). The following will remove the variable `A` from the workspace (check this in the upper right panel)

```
rm(A)
```

The objects created during an R session can be stored permanently in a file with the extension `.Rdata` for use in future R sessions.

For example, we can save the variables `x` and `y` for later use in a file named `Objectfile.Rdata`.

```
x <- 1
y <- 2
save(x, y, file = "Objectfile.Rdata")
```

This file will be saved in the working directory (check ist appearance in the `Files` panel). We can now remove these two variables from the workspace via the remove command

```
rm(x, y)
```

If you check the content of the workspace with `ls()`, or look into the check this in the upper right `workspace` panel of **Rstudio**, you can verify that `x` and `y` are no longer there. At a later time, we can reload these two variables again into the workspace via the `load` function

```
load (file = "Objectfile.Rdata")
```

If we check the contents of the workspace with `ls()`, it will show that the variables `x` and `y` are again present. This combination of the `save` and `reload` commands provides a handy mechanism to be able to re-use the (intermediate) results of the present analysis in later R sessions.

At the end of each R session you are given the opportunity to save all the currently available objects. If you indicate that you want to do this, the objects are written to a nameless file called `.RData` in the working directory, and the command lines used in the session are saved to a similar file called `.Rhistory`. When R is started at later time it will reload this saved workspace and command history.

2.5. Working with scripts

In addition to simple command line execution, there is a second, more powerful way of using R. Suitable sequences of R commands can be put together in a so-called R script and saved in a file ("filename.R") for later re-use. This way, instructions do not have to be typed in the console over and over again. Scripts are most easily developed in a specifically dedicated IDE like **RStudio**.

Open a new, blank file in **RStudio** (go to menu `File:NewR script`). A new window will pop up in the upper left panel. Then type the following in this new panel (not in the console):

```
A <- 1
B <- 2
X <- A + B
print(X)
```

Save this file to the workingdirectory on your file system (go to menu `File`, select `Save as`, and type the name you'd like to give to the file, for example, `MyFirstFile.R`).

This is your first R script, congratulations! But what does it mean? The first two lines initialize the variables `A` and `B`. The command `X <- A + B` creates a third variable, `X`, which stores the sum of `A` and `B`. The last line prints the content of `X` within the console.

To execute these 4 statements, they need to be submitted to the R console. Submitting implies that the R console will execute the separate commands in the script, one line after the other. Scripts can be submitted in a number of ways in **RStudio**. To submit a limited set of commands, one first selects these commands in the editor, and then click the `Run` button in the editor panel. To execute the previously selected code, press the `Re-Run` button. To execute all commands in the entire script file, click the `Source` button in the editor panel.

Throughout these notes, the following conventions are used. R statements in a script file lack the prompt `>` and are written as:

```
 X <- A + B
 X
```

However, when submitted to the R console, these same statements are always preceded by a prompt `>`:

```
> X <- A + B
> X
```

The resulting output is calculated by the console and is represented in the console window as:

```
[1] 3
```

A powerful and often used programming technique is the re-use of the variables. In the above example, the sum of `A` and `B` is stored in a newly created variable `X`. However one could also use a variable name that is already in use:

```
> A <- 1
> B <- 2
> A <- A + B
> A
```

```
[1] 3
```

When the value of `A + B` is calculated, it is again assigned to the variable `A`. The old value of `A` (= 1) is overwritten by the sum of `A` and `B` (= 3). Why would one consider such re-use of variables? Instead of three variables as above (`A, B, X`), the code now only uses two variables (`A, B`) and so less computer memory is taken up. For a simple example like this, such memory considerations are not important. However, when writing large programs, proper memory management can become an issue.

2.6. Errors and warnings

When a command is not correctly formulated according to R syntax, the computer will not execute it. For example if you use `ln` rather than `log` for the natural logarithm, the computer will issue an error message. This message will give you a (sometimes vague) clue of what the cause of the error could be.

```
>ln(10)
```

```
Error: could not find function "ln"
```

Now let's make another deliberate error by typing:

```
> log10(100
+
```

We forgot to add the closing bracket, and hence the expression is not complete. When we type `Enter` R has changed the ">" prompt to a "+" prompt, meaning that it expects *more*, i.e. it indicates that the previous command was not yet finished. To still get the result one can type the missing bracket ")" and `Enter`. Or one can press the `Esc` key to cancel the calculation.

If a sentence on one line is syntaxically correct, R will execute it, even if it is your intention that it proceeds on the next line. Suppose you want to calculate $3 + \cos(\pi) - \sqrt(5)$. We can write this statement on two lines as part of a script in the editor and submit it to the console. The result depend on how we do this. For instance, if we write the following:

```
3 + cos(pi)
    - sqrt(5)
```

then this will be erroneously interpreted as two separate commands.

```
> 3 + cos(pi)
```

```
[1] 2
```

```
>    - sqrt(5)
```

```
[1] -2.236068
```

The separate values of $(3 + cos(\pi))$ and $-\sqrt{5}$ will be printed (which is not intended). In contrast, if one writes:

```
3 + cos(pi) -
    sqrt(5)
```

a different output will appear in the console.

```
>  3 + cos(pi) -
     sqrt(5)
```

```
[1] -0.236068
```

R will interpret these two lines now as one single command and print the value of $3 + cos(\pi) - \sqrt{5}$. Because the expression on the first line was not syntaxically finished, R has (correctly) assumed that it continued on the next line. Note the change in R-prompt from ">" to "+".

Tips and tricks:

This example shows that you should be careful when you want to split a complex statement over several lines! These errors are very difficult to trace, so it is best to avoid them.

2.7. Getting help

In order to use a specific function, one should know how to use it. A great advantage of R is that it has an extensive built-in help facility. To get more information on any specific function, for example `seq()`, simply type

```
?seq
```

The longer alternative is `help(seq)`. After pressing `Enter` a HTML help file will pop up (in **Rstudio** in the lower right panel `Plots`). This panel explains you what this function does and how to use it. In the case of the `seq` command, the help file explains how to create a sequence of numbers. This help facility is a very powerful feature of R: make ample use of it!

For special characters, the argument must be enclosed in quotes, making it a "character string". This is also necessary for a few words with syntactic meaning, like `if`, `for` and `function`.

```
help("[")
```

If you don't know the exact name of the function, you can search for a keyword. The `help.search` command (alternative `??`) allows searching for keywords. For example,

```
??seq
```

will list all occurrences of the word `seq` in the R documentation.

Most of the help files include examples. You can run all of them by using the command `example`. For instance, typing into the console window:

```
> example(seq)
```

will run all the examples from the `seq` help file. Alternatively, you may select one example, copy it to the clipboard (ctrl-C for windows users) and then paste it (ctrl-V) in the console window.

In addition, the R base installation and many R packages come with demonstration material. Typing:

```
> demo()
```

will give a list of available demonstrations in the base installation.

```
> demo(graphics)
```

will demonstrate some simple graphical capabilities. Finally, many R packages come with so-called package "vignettes". These are manuals that explain how to work with the package. For instance, typing

```
> vignette("deSolve")
```

will open a (technical) manual about how to use R-package **deSolve**.

Tips and tricks:

The best help is often provided by the very active mailing list on the internet. If you have a specific problem, just type `R: <problem>` *on your search engine (e.g. Google). Chances are that someone has already encountered the problem and solved it.*

2.8. A quick glance on things to come...

Open the file "QuickIntro.R" into the editor panel. This script illustrates how R can perform a rather sophistaced task with only a few statements. The script plots a linear regression line on a data graph, where the data are from an experiment which has lasted for 10 hours. At different time points we have measured the O_2 concentration in a closed incubation chamber.

We can execute the script in a stepwise fashion by selecting blocks of commands and using the `Run` button on top of the editor panel. Right now, we only briefly discuss what happens in the script. The separate commands will be dicussed in more detail in later chapters. Execute the first line.

```
# The "seq" function creates a sequential vector
```

Nothing happens. This is because this line starts with `#` and represents a comment, which is ignored during execution. As soon as the symbol `#` appears, anything to the right on the same line is ignored.

Let's move on to the next two lines. The `seq()` function creates a sequential vector that harbours the regular time points at which data were collected (expressed in units of hours). The `from =` , `to =` and `by =` are so-called *arguments* to the function, which need to be specified. Here we create a series of numbers starting from 0 to 10, and spaced by 1. The vector is subsequently displayed.

```
# The "seq" function creates a sequential vector
time.vector <- seq(from = 0, to = 10, by = 1)
time.vector
```

```
 [1]  0  1  2  3  4  5  6  7  8  9 10
```

The oxygen data do not follow a regular sequence and are introduced in a different way. The concatenation function `c()` assembles a vector out of individual numbers:

```
conc.vector <- c(270, 250, 232, 209, 193, 176, 159, 145, 130, 124, 113)
conc.vector
```

```
 [1] 270 250 232 209 193 176 159 145 130 124 113
```

Both `c()` and `seq()` are *built-in* functions in R (later you will learn how to make your own functions). Now test what happens if you execute:

```
# Making a first plot
plot(time.vector, conc.vector)
```

The `plot` command creates a pop-up graphics window in which the experimental data are plotted. The result can be seen in figure 2A. This graph has a rather primitive look. We can make it a little bit more professional by adding a title and labels to the axes. The `plot` statement is changed into:

```
plot(time.vector, conc.vector, ylim = c(0, 300), xlab = "Time [hr]",
  ylab = "O2 concentration [micromol L-1]", main = "Data O2 incubation")
```

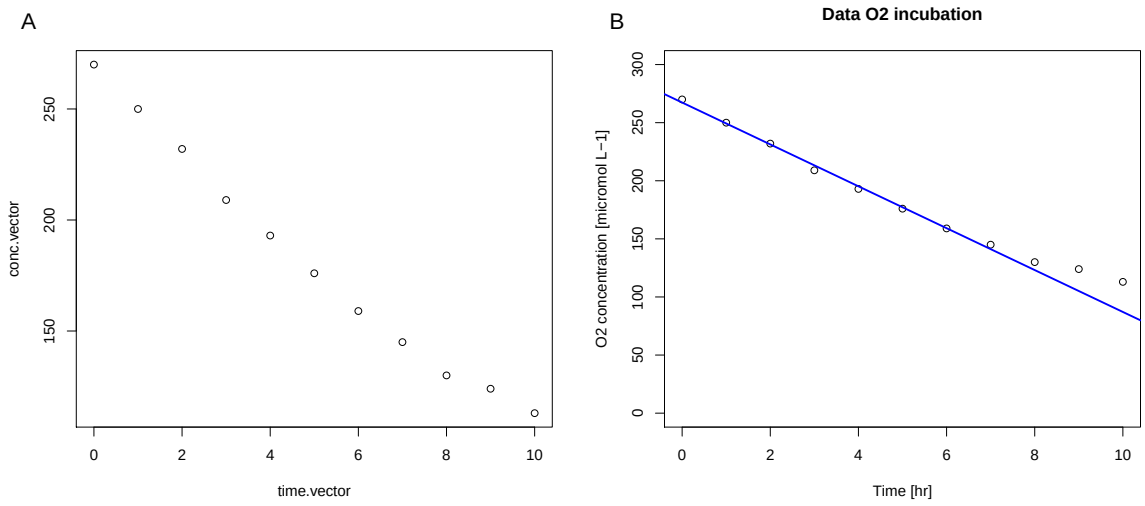


Figure 2: A. Function `plot` using default arguments. B. A nicer plot, overruling the default arguments of the `plot` function. See text for explanation.

At this point the arguments in the `plot` function may not mean much. We just include it at this stage to show that by a single (high-level) instruction one can obtain a quite complex response (the creation of a graph). In a later chapter we will go into more details about the graphics in R .

Now test what happens if you execute the rest of the code:

```
# Adding a trendline through first eight data points
index <- 1:8
model.fit <- lm(conc.vector[index] ~  time.vector[index]) # lm = linear model
abline(model.fit, lwd = 2, col = "blue")
```

This will fit a trendline through the first eight data points and produce the final graph. Try to fit a line through 6 or 10 points, and replot the graph.

3. Numbers and vectors

The basic data structure of R is the numeric `vector`, which is an ordered set of numbers. R calculates as easily with vectors (and matrices and arrays) as with single numbers. Learning how to create and manupulate these variables is essential if you want the make good use of the R software.

3.1. Working with single numbers

R can work with integer, real and complex numbers. Unlike other programming languages R has no single precision data type. All real numbers are stored in double precision format. So when initializing numeric variables, the associated values will always be stored in double precision. For most purposes, you need not be concerned too much how the numbers are actually stored.

```
> (X <- 3)
```

```
[1] 3
```

In this statement, the variable value `X` is created and assigned the value of 3 (which is stored internally in a double precision format). The arrow "`<-`" is the assignment operator, and the brackets "( )" denote that the result should be displayed on the screen (as detailed above).

A peculiar feature of the R language is that it has no provision for scalars (that is single numbers), like other programming languages. So R represents a single number as a vector of length one. This is why the `[1]` stands in front of the printed result, indicating that the first element of the vector `X` contains the value 3.

The elementary arithmetic operators can be listed by typing `?Arithmetic` which opens the help file.

```
> (X^2 + X*2) / (4-X)
```

```
[1] 15
```

There exists a priority listing for the various operators in expressions (see `?Syntax`). Round brackets are used to ensure that expressions are properly evaluated. To see this, try the above expression without brackets:

```
> X^2 + X*2 / 4-X
```

```
[1] 7.5
```

In addition, R contains a suite of built-in mathematical functions, such as logarithms and exponential (`log`, `log10`, `log2`, `exp`) and trigonometric functions (`sin`, `cos`, `tan`, `asin`, `acos`, `atan`).

There is a special number (`Inf`) which represents infinity, and which can be used in ordinary calculations. The value (`NaN`) represents an undefined value (Not a Number). Try:

```
> 1/0
> 0/0
> 1e-8 * 1000
```

The engineering representation `1e-8` denotes 10^{-8}. R also allows to represent missing values by `NA`, meaning Not Available.

Exercises

Open a new blank file in the editor window. Write your R code in this file and use the "#" for comments. This way you can keep the solutions to the exercises for later reference. Save this file imemdiately under an appropriate name (e.g. "Solutions Chapter 2.R").

Ex01: Using R as a calculator.

Use the R console to calculate the value of following mathematical expressions. Try the different ways to submit the statements to R. You may want to look at the help files for some of these functions. Typing `?Arithmetic` will open a help file with the common arithmetic operators.

- $(15/6 - 1)^{2/3}$ (Answer: 1.310371)
- $log_2(4096)/ln(20)$ (Answer: 4.005698)
- $(2\pi)^3$ (Answer: 248.0502)
- $\sqrt{2.3^2 + 5.4^2 - 5 * \cos(\pi/8)}$ (Answer: 5.46174)

Ex02: Unit conversion.

The primary production in the North Sea ranges from 50 to 400 gram Carbon (C) per meter squared per year. Write a small script to recalculate this range in the units of mmol Carbon per meter squared per day (knowing that the atomic weight of Carbon = 12 gram per mol). (Answer: 11.4 - 91.3 mmol m-2 d-1)

Ex03: Estimating the natural Greenhouse effect.

The current average surface temperature on earth is about 15 ^{o}C. If the earth would have no atmosphere containing greenhouse gases, its surface temperature (in Kelvin) could be calculated from the formula $T = ((1 - A) * S/(4\sigma))^{0.25}$ with known values for the planetary albedo A (0.3), the solar flux S (1366 W m-2), and σ is a constant (5.67e-08 W m-2 K-4). How chilly would it be on earth, and so how large is the natural greenhouse effect? Write a small R script. (Answer: -18.33 deg C)

3.2. Creating a vector

The easiest way to create a vector is by means of the `c()` command (where the `c` stands for concatenation)

```
V <- c(0, 5.1, 6, 12.3, 20)
```

This `c()` function is presumably THE most important function in R.

Regular sequences of numbers can be generated by the functions `seq` (sequence) and `rep` (repeat). The `seq()` command takes as input the arguments `from` (start value of the sequence), `to` (end value of the sequence) and either `by` (the increment) or `length.out` (the required number of elements in the sequence).

```
V <- seq(from = 0, to = 1, by = 0.2)
V <- seq(from = 0, to = 1, length.out = 6)
```

Another way to produce a sequence is by the colon notation `1:10`, which is an abbreviation for `seq(from = 1, to = 10, by = 1)` and represents the vector `c(1, 2, ..., 9, 10)`. Accordingly, the colon operator creates a special type of sequences where the increment is always 1. Other examples of the colon operator are:

```
> (V <- 0.5:10.5)
```

```
 [1]  0.5  1.5  2.5  3.5  4.5  5.5  6.5  7.5  8.5  9.5 10.5
```

```
> (V <- 6:1)
```

```
[1] 6 5 4 3 2 1
```

The related function `rep()` is used to create a vector in which elements are repeated:

```
> (V <- rep(1, times = 5))
```

```
[1] 1 1 1 1 1
```

The `rep()` command can be used to replicate parts of vectors in complicated ways

```
> (V <- rep(c(1, 2), times=5))
```

```
 [1] 1 2 1 2 1 2 1 2 1 2
```

```
> (V <- c(rep(1, 5), rep(2, 5)))
```

```
 [1] 1 1 1 1 1 2 2 2 2 2
```

3.3. Other types of vectors

A vector cannot only contain numbers. For example, the concatenation function also works for characters:

```
> (students <- c("Andreas", "Karel", "Pieter", "Anna"))
```

```
[1] "Andreas" "Karel"   "Pieter"  "Anna"
```

The *type* of a vector is the kind of the elements it contains and must be one of the following: logical, integer, double, complex, character, or raw. All elements of a vector must have the same underlying type. This important restriction does not apply to lists (see below). We can retrieve the type of the elements of a vector via:

```
> typeof(students)
```

```
[1] "character"
```

Try:

```
typeof(seq (from = 1, to = 4, by = 1))
typeof(1:4)
typeof(1i)  # complex number
typeof(TRUE)
```

Finally, one can also create a vector using the function `vector()`. This method is particularly useful if one does not know in advance how many elements will be stored, and so one can create an empty vector.

```
> V <- vector(mode = "integer", length = 5)
> X <- vector()
```

The fisrt command generates an empty vector `V` of type `integer` and containing 5 elements (filled with zero's by default). The second vector creates an empty vector `X` of unknown length. Normally numeric vectors are double precision by default. Integer vectors are only used in special occasions.

A peculiar feature of R is that the elements of a vector can also be given `names`. For example, the coordinates of a point in three dimensional space (x, y, z) could be implemented as:

```
> (point <- c(x = 1, y = 2, z = 3))
```

```
x y z
1 2 3
```

The names of the vector (or any other object) can be interrogated by the `names()` function

```
> names(point)
```

```
[1] "x" "y" "z"
```

The advantage of given names to vector elements is that names are often easier to remember than numeric indices. This option is particularly useful in connection with `data.frames` (see below).

3.4. Calculating with vectors

In R it is as simple to perform calculations with vectors as it is with single numbers. The calculations are performed on an element by element basis. In the next example, all elements of the vector `A` are first multiplied with 2 and then the square root is taken.

```
> A <- c(1, 3, 5, 6)
> sqrt(A * 2)
```

```
[1] 1.414214 2.449490 3.162278 3.464102
```

However, be careful! When you perform operations on two vectors, make sure that they are of the same length. This is because R has a peculiar recycling feature. Adding a vector of length 3 and a vector of length 5 would raise a so-called exception in most programming languages. The language designers would assume the programmer has made an error. However, R allows adding two vectors regardless of their relative lengths. The elements of the shorter vector are recycled as often as necessary to create a vector the length of the longer vector. So when the length of vectors do not match, R automatically recycles the shortest vector untill it matches the longer one, and only then the operation is performed. R does issue a warning, but only when the length of the longer vector is not an integer multiple of the length of the shorter vector. So, for example, adding vectors of lengths 3 and 7 would cause a warning, but adding vectors of length 3 and 6 would not

```
A <- c(1, 3, 5, 6)
B <- c(1, 2, 3)
A + B
```

```
Warning message:
In A + B : longer object length is not a multiple of shorter object length
```

When the length of the longer vector is an exact multiple of the shorter objects length, then no warning will be issued:

```
> A <- c(1, 3, 5, 6)
> D <- c(1, 2)
> A + D
```

```
[1] 2 5 6 8
```

Accordingly, vectors can be used in arithmetic expressions. Built-in mathematical functions can be directly applied to vectors, which leads to concise code. The following command calculates the sine of a sequence and displays the result:

```
> sin( seq(from = 0, to = 2*pi, by = pi/2 ))
```

```
[1]  0.000000e+00  1.000000e+00  1.224606e-16 -1.000000e+00
[5] -2.449213e-16
```

Although we used only one statement, it is more clear if we use two statements:

```
> A <- seq(from = 0, to = 2*pi, by = pi/2 )
> sin(A)
```

```
[1]  0.000000e+00  1.000000e+00  1.224606e-16 -1.000000e+00
[5] -2.449213e-16
```

There also exist operations on vectors that return one single number. The `min()` and `max()` functions select the smallest and largest element in the vector. The `length()` function returns the number of elements, `sum()` provides the sum of all elements, and `prod()` their product.

```
> A <- 1:10
> c(sum (A), mean(A), min(A), max(A))
```

```
[1] 55.0  5.5  1.0 10.0
```

Two important statistical functions are the `mean()` and `var()` function, calculating the sample mean and the sample variance of the data contained in the vector. The `summary` command prints a set of statistics (min, max, mean,...) for a vector:

```
> summary(A)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   1.00    3.25    5.50    5.50    7.75   10.00
```

To compare two vectors one should use the parallel minimum `pmin()` and maximum `pmax()` functions, which compare the two vectors side by side, and select the appropriate element in each position.

```
> A <- c(1,2,3); B <- c(3,2,1)
> pmax(A,B)
```

```
[1] 3 2 3
```

```
> pmin(A^2,B^2)
```

```
[1] 1 4 1
```

Accordingly, these functions return a vector of the same length as the largest input vector.

3.5. Indexing and subsetting of vectors

The elements inside a vector are indexed starting with 1. The elements of a vector can be accessed using the square brackets `[]`. If we want to obtain the second element of vector `point`, we can write:

```
> point[2]
```

```
y
2
```

or, using the name of the element:

```
> point["y"]
```

```
y
2
```

If we want to obtain all *except* the second element of vector `point`, we write:

```
> point[-2]
```

```
x z
1 3
```

We can also use the index to subset a vector (i.e. change the values of specific elements of that vector) :

```
> point["z"] <- 4
> point
```

```
x y z
1 2 4
```

Instead of a single element, one can also select more than one, by using a suitable indexing vector (this is a vector containing integers).

```
> point[1:2]
```

```
x y
1 2
```

```
> point[c(1, 3)]
```

```
x z
1 4
```

Subsetting can be applied to change multiple elements at once:

```
> grades <- c(henk = 8.5, herman = 5, petra = 7, louise = 6.5)
> boys <- c(1, 2)
> grades[boys]
```

```
  henk herman
   8.5    5.0
```

```
> grades[-boys]
```

```
 petra louise
   7.0    6.5
```

```
> grades[c("petra", "herman")]
```

```
 petra herman
     7      5
```

3.6. Logical vectors and conditions

R distinguishes the logical variables `TRUE` and `FALSE`, represented by the integers 1 and 0. The logical operators are `<`, `<=`, `>`, `>=`, `==` for exact equality and `!=` for inequality. The symbol `&` means "and", `|` is "or", `!` is "not". Following help pages list the relational and logical operators available in R.

```
> ?Comparison
> ?Logic
```

Logical expressions are often used to select elements from vectors (and matrices as we see later) that obey certain criteria. Logical vectors (containing `TRUE` or `FALSE` as elements) are created by conditions. For example:

```
> (V <- seq(from = -2, to = 2, by = 0.5))
```

```
[1] -2.0 -1.5 -1.0 -0.5  0.0  0.5  1.0  1.5  2.0
```

```
> (test <- V > 0)
```

```
[1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE
```

The vector `test` is a logical vector of the same length as vector `V`. The elements of `test` are TRUE when the condition is met and FALSE otherwise. These logical vectors can be used as indexing vectors to select part of a vector.

```
> V[test]
```

```
[1] 0.5 1.0 1.5 2.0
```

```
> V[V > 0]
```

```
[1] 0.5 1.0 1.5 2.0
```

will select the positive values from `V`. The statement

```
> V[V > 0] <- 0
```

will zero all positive elements in V, while

```
> sum(V < 0)
```

```
[1] 4
```

will return the number (4) of negative elements. The latter is possible because R represents `TRUE` and `FALSE` by the integers 1 and 0, which can be summed. The statement

```
> V[V != 0]
```

```
[1] -2.0 -1.5 -1.0 -0.5
```

will display all nonzero elements from V ("!" is the "not" operator). Logical tests can also be combined, using | (the "or" operator), and & ("and").

```
> V [V <(-1) | V > 1]
```

```
[1] -2.0 -1.5
```

will display all values from V that are strictly smaller than -1 and strictly larger than 1. Note that we have enclosed "-1" between brackets (can you see why this is necessary?) Finally, the statements

```
> which (V == 0)
```

```
[1] 5 6 7 8 9
```

```
> which.min (V)
```

```
[1] 1
```

will respectively return the index of elements with 0-value, and the index of the element that contains the minimum inside `V`.

3.7. Removing elements

When the index is preceded by a "-", the element is removed.

```
V <- V[-c(1,2)]
V <- V[-which(V >= 0)]
```

will delete the 1^{st} element of V and the positive elements of V (3^{rd} line). For more information, type

```
?Extract
```

3.8. Exercises

Creating and manipulating vectors is essential if we want to use R as a mathematical tool. Although this has been implemented in a consistent way in R, it is not simple for novice users! Practice is the best teacher, so you will get plenty of exercise. Most of the exercises can be answered with one single R-statement. However, as these statements can be quite complicated, it is often simpler to first break them up into smaller parts, after which they are merged into one.

Ex04: The mean of a vector.

Use R -function `mean()` to estimate the mean of two numbers, 9 and 17. (You may notice that this is not as simple as you might think!).

Ex05: Vector `V`

- Create a vector, called `V`, which contains the even numbers between 16 and 56. (Hint: use the R-function `seq()`)
- Display this vector
- What is the sum of all elements of `V`? There exists an R-function that does this in one statement.
- Display the first 4 elements of `V`
- Calculate the product of the first 4 elements of `V` (Hint: use R-function `prod()`).
- Display the 4^{th}, 9^{th} and 11^{th} element of `V`. (Hint: use the concatenation function `c()`).

Ex06: Vector `W`

- Create a new vector `W`, which equals vector `V` multiplied by 3. Display the content of `W`.
- How many elements of `W` are smaller than 100?

  First create a new vector that contains only the elements from `W` < 100 (call it `W100`), then calculate the length of this new vector.
- Now perform the same calculation, in one R statement.

Ex07: Sequences

- Create a sequence that contains the values (1, 1/2, 1/3, 1/4, ..., 1/10).
- Compute the square root of each element.
- Compute the square (^{2}) of each element.
- Create a sequence with values (0/1, 1/2, 2/3, 3/4, ..., 9/10).

Ex08: Vector `U`

- Create a vector, `U`, with 100 random numbers, uniformly distributed between -1 and 1.

  tip: R-statement `runif` generates uniformly distributed random numbers; use `?runif` to see how it works.
- Check the range of `U`; all values should be within -1 and +1.

  tip: there exists an R-function to do that - its name is trivial.
- Calculate the sum and the product of the elements of `U`.
- How many elements of `U` are positive?
- Zero all negative values of `U`.
- Sort `U`.

Ex09: Vectors `x`, `y`

- Create two vectors: vector `x`, with the elements: 2, 9, 0, 2, 7, 4, 0 and vector `y` with the elements 3, 5, 0, 2, 5, 4, 6 (in that order). (tip: use the `c()` function).
- Divide all the elements of `y` by the elements of `x`.
- Type in the following commands; try to understand:

```
x > y
x == 0
```

- Select all values of `y` that are larger than the corresponding values of `x`.

- Select all values of `y` for which the corresponding values of `x` are 0.
- Remove all values of `y` for which the corresponding values of `x` equal 0.
- Zero all elements of `x` that are larger or equal than 7. Show `x`.

4. Lists, data frames and data input/output

Besides vectors (and matrices), R allows the creation of more complex objects such as lists and data frames. These two object types are very important for data storage. When performing simulations or statistical analyses, function calls will often return their results as a list or data frame. Similarly, when building your own functions, it is often useful to return the calculated results as a list or data frame, especially when the function calculates different types of results.

4.1. Lists

Until now, we have used vectors to store data. However, vectors have one important restriction: all elements must be of the same type. This restriction no longer holds for a list. Formally, a *list* is an object, which itself consists of an ordered collection of other objects, referred to as the *components* of the list. The advantage is that the components of a list can be of different types and/or lengths. For example, a personnel database could hold an entry for each employee, specifying the personal details of the employee (e.g. name, number of children, age of the children) :

```
> employee <- list(name = "Wendy", n.child = 2, child.age = c(21, 20))
```

The command `length(employee)` provides the number of (top level) components in the list (three in the above example) while `names(employee)` lists the names of all components (`name`, `n.child` and `child.age`).

The components are always ordered, meaning that they can be referenced by their index in the list. To select a specific component one uses the double square bracket operator `[[ ]]`. To select the first component:

```
> employee[[1]]
```

```
[1] "Wendy"
```

If the component is itself a vector, then one can use the single bracket operator `[ ]` to select a specific element in this vector.

```
> employee[[3]][1]
```

```
[1] 21
```

This last statement selects the first element of the vector, which itself forms the third component of the list. A great advantage is that the components of a list can also be referenced by their name. This can be done by using the `$` notation:

```
> employee$name
```

```
[1] "Wendy"
```

```
> employee$child.age[1]
```

```
[1] 21
```

Additionally, one can also use the name of the list component in double square brackets. This is particularly when the name of the component is stored in another variable

```
> employee[["name"]]
```

```
[1] "Wendy"
```

```
> what.to.select <- "n.child"; employee[[what.to.select]]
```

```
[1] 2
```

This referencing of list components by name greatly improves the readability of R scripts.

To extract information from a list one can use either single brackets or double brackets. It is very important to understand the difference between the commands `employee[2]` and `employee[[2]]`. The operator `[ ]` always returns an object of the same type as the original object.

```
> employee[1]
```

```
$name
[1] "Wendy"
```

```
> employee[[1]]
```

```
[1] "Wendy"
```

Accordingly, in the first case, a new list is created, which only keeps the first component of the original list. In the second case, the value of the first component is returned (without its name). The difference is further illustrated by the function `typeof()`, which shows the type of an object. The type is `"list"` in the first case and `"character"` in the second case.

```
> typeof(employee[1]); typeof(employee[[1]])
```

```
[1] "list"
```

```
[1] "character"
```

Lists can be extended by specifying additional components. This can be done in a number of ways:

```
> employee["gender"] <- "Female"
> employee$working.hours <- 38
> employee <- c(employee,list(employer="NIOZ"))
> employee
```

```
$name
[1] "Wendy"

$n.child
[1] 2

$child.age
[1] 21 20

$gender
[1] "Female"

$working.hours
[1] 38

$employer
[1] "NIOZ"
```

The following example shows how one can create quite complicated data structures by using list objects as components inside a list.

```
spec.list <- vector(mode = "list", length = 3)
names(spec.list) <- c("O2", "H2O", "CO2")
spec.list$O2  <- list(name = "oxygen", mol.weight = 32)
spec.list$H2O <- list(name = "water", mol.weight = 18)
spec.list$CO2 <- list(name = "carbon dioxide", mol.weight = 44)
```

The `names()` command initializes the names of the mother list. Each daughter component of the mother list is in itself again a list object. Print `spec.list` on the screen and try to understand its contents.

4.2. Data frames

Just like a list, a *data frame* can include different data types (e.g. character, logical, numeric), though now arranged in a tabular format. The following table could be set up by a taxonomist specialized in roundworms:

```
> genus <- c("Sabatieria", "Molgolaimus")
> dens  <- c(1, 2)
> Nematode <- data.frame(genus = genus, density = dens)
> Nematode
```

```
        genus density
1  Sabatieria       1
2 Molgolaimus       2
```

In this example, the data frame `Nematode` contains two columns, one containing strings (the genus name) and one containing numeric values (the densities). In essence, a data frame is a special type of list (of the class `data.frame`) consisting of vectors with equal length. For most purposes, a data frame can be regarded as a "generalized" matrix, where each column stores a vector with a different type of elements. Many matrix operations also work on data frames provided the data frame contains a single data type.

The vectors making up the columns of the data frames can be extracted by using `$` and the names of the column, or by using the double square bracket `[[ ]]` operator.

```
> Nematode$density[2]
```

```
[1] 2
```

```
> Nematode[[2,2]]
```

```
[1] 2
```

Just as with lists, one should be careful about the use of single brackets versus double brackets. The object resulting from a selection with single brackets `[ ]`, will be a new data frame or a list. When using double brackets `[[ ]]`, one will obtain a vector. Try:

```
> Nematode$genus
> Nematode[[1]]
> Nematode[1]
```

The first two statements are identical. The third statement will print the two genus names in a different format.

The names of the columns are very convenient when manipulating the data contained within the data frame. For instance:

```
> Nematode$density / sum(Nematode$density)
```

```
[1] 0.3333333 0.6666667
```

will calculate the relative density of each genus in the dataset (i.e. divide the density values by the summed density). Similarly, the following command will calculate the mean density of all nematode genera.

```
> mean(Nematode[ ,2])
```

```
[1] 1.5
```

Adding new columns to a data frame is simple. Here we add a column called `size`

```
> Nematode$size <- c(700, 400)
> Nematode
```

```
        genus density size
1  Sabatieria       1  700
2 Molgolaimus       2  400
```

The addition of new rows is preferably done with the `rbind` command (see chapter on matrices):

```
> NewNem <- data.frame(genus = "Halalaimus", density = 10, size = 1000)
> Nematode <- rbind(Nematode, NewNem)
> Nematode
```

```
        genus density size
1  Sabatieria       1  700
2 Molgolaimus       2  400
3  Halalaimus      10 1000
```

The helpfile `?Extract` explains the various ways in which to extract elements from lists and data frames.

4.3. Data Conversion

Conversion from one class of data structures to another can easily be done, e.g. by:

```
> as.data.frame(M)
> as.vector(A)
```

If unsure about the class, you can write:

```
> is.data.frame(M)
> is.vector(A)
```

Or you can display the data class by:

```
> class(M)
```

4.4. Data import from external sources

In all previous examples, data was entered as input from the keyboard. However, in scientific applications, large data sets usually must be read from external files (e.g. ASCII or csv files). R provides a number of tools to import data from such external sources. For advanced details on importing and exporting data in R , see the R Data Import/Export manual.

The best practise is to structure an save your dataset in a tabular format, so that it can be read directly as a data frame. To read data written in a tabular form from text files, on ecan use the functions `read.table()`, `read.csv()`, or `read.delim()`.

To read an entire data frame directly with `read.table()`, the external file must be formatted following a rather strict template.

- The first line of the file should list the names of all variables.
- Each additional line of the file should have a row label, followed by the values for each variable.

Accordingly, the first line should have on item less than all following lines.

To see how this works, open the file "studentscores.txt" in the **Rstudio** editor window. You will find a table with scores of students for different courses. Each line contains a row label followed information about a specific student (names + scores). All items on a line are separated by a space. The `sep=" "` argument specifies that the separator is a space.

```
> read.table("studentscores.txt",sep=" ")
```

```
    name Physics Geology Biology
1   Fred      15      11      17
2 Ginger      12      14      13
3 Robbie      11       9      12
4   Lucy      12       7       6
```

The same student score information is contained within the file "studentscores.csv", which is now a comma separated file. This format has the advantage that it can easily be generated or read by spreadsheet programs (like Ms Excel). The difference is that there are no row labels, and items are separated by a comma (rather than a space).

```
> read.csv("studentscores.csv")
```

```
    name Physics Geology Biology
1   Fred      15      11      17
2 Ginger      12      14      13
3 Robbie      11       9      12
4   Lucy      12       7       6
```

This `read.csv` statement results in exactly the same data frame as the above `read.table` command.

Note: By default all numeric items (except row labels) are read as numeric variables and non-numeric variables as factors. This can be changed if necessary.

4.5. Exercises

Ex01: North Sea

The following features characterizes the North Sea.

- average depth: 95 m
- surface area: 750 000 km2
- countries: France, U.K., Belgium, The Netherlands

Summarize this information into a list. Determine the length of the countries vector in this list using the function `length()`. Estimate the volume of the North Sea (Answer: 71250 km3).

Ex02: Wadden Sea data set

A marine biologist has determined the density of invertebrates in the sediments of the Wadden Sea and wants to store his data in a systematic way. She creates a separate list for each species she has found. To properly identify a given organism, she uses the AphiaID (a unique identifier) and Name from world register of marine species (WoRMS, see http://www.marinespecies.org/), which stores information of all known marine organimsms. For the common lugworm (a polychaet) the respective values are 129868 and "Arenicola marina (Linnaeus, 1758) ".

- Create a list called "lugworm" that holds the above WoRMS information
- Add a new component this list named Environment with the value "Wadden Sea " (the sampling area)
- Add a new vector as a component to this list. This vector should hold the measured densities (expressed in organsims per square meter) at 10 locations in the sampling area (values: 15, 18, 35, 17, 5, 52, 8, 13, 67, 11).
- Calculate the mean density of lugworms for the first 8 stations in the Wadden Sea dataset (Answer: 20.375).

Ex03: The world's largest rivers

Create a data frame with the following information of the worlds largest rivers: (length in km, discharge in m3/sec):

River	*length*	*discharge*
Nile	6650	5100
Amazon	6400	219000
Yangtze	6300	31900
Mississipi	6375	16200

- Calculate the mean length of the rivers.
- Add an extra column with the drainage area in 1000 km^2, given as: `c(3349, 6915, 1800, 2980)`. Hint: use `cbind()` to do this.
- Add an extra column with the length in miles. One mile = 1.609344 km. Hint: use the existing column containing the river length in metres.
- Add the information for the river Rhine: length = 820 miles, discharge = 2290 $m^3\ s^{-1}$, drainage area = 170000 km^2.

Ex04: Lists as output of a function (*)

The function `diffcoeff` (from the package marelac) calculates the molecular diffusion coefficient in water. What is the difference between following function calls (Hint: store each result in variable called "test" and use `typeof()` to check the type of this new variable).

```
> diffcoeff(S = 35, t = 20, P = 1, species = "O2")[["O2"]]
> diffcoeff(S = 35, t = 20, P = 1, species = "O2")$O2
> diffcoeff(S = 35, t = 20, P = 1, species = "O2")
```

4.6. A real world application: Diversity of deep-sea nematodes

Nematodes are small (< 1 mm) roundworms, which can be very abundant in marine sediments. We will now work on a data set consisting of nematode species densities, found in Mediterranean deep-sea sediments, at depths ranging from 160 m to 1220 m (see (Soetaert, Heip, and Vincx 1991)). The densities are expressed in number of individuals per 10 cm^2 (yes, nematologists use strange units!).

The data were originally present in a database, and have been exported as a table in the comma-separated-values format (creating a csv file). Open the file "nemaspec.csv" in your editor. Check its structure. You may also open the file in your spreadsheet software, but do not forget to close it before proceeding. Excel is rather "territorial", and will not allow R (or any other program) to access a file that is open in Excel.

On the first line is the heading (the names of the stations), the first column contains the species names. Make a script file in which you write the next steps. Submit each line to R to check its correctness.

Read the comma-delimited file, using R-command `read.csv`. Type `?read.csv` if you need help. Specify that the first row is the heading (`header=TRUE`) and the first column contains the rownames (`row.names=1`). Put the data in data.frame `Nemaspec`.

```
Nemaspec <- read.csv("nemaspec.csv", header = TRUE, row.names = 1)
```

Check the contents of `Nemaspec`. As the dataset is quite substantial, it is best to output only the first part (the `head`) of the data:

```
head(Nemaspec)
```

The rest is up to you:

- Select the data from station M160b (the 2^{nd} column of Nemaspec). Put these data in a vector called `dens`. (Hint: to select a complete column, you can leave the row index blanc).
- Remove from vector `dens`, the densities that are 0. Display this vector on the screen. (Answer: [1] 6.580261 5.919719 etc...)
- Calculate N, the total nematode density of this station. The total density is simply the sum of all species densities (i.e. the sum of values in vector `dens`). What is the value of N ? (Answer: 699).
- Divide the values in vector `dens` by the total nematode density `N`. Put the results in vector `p`, which now contains the relative proportions for all species. The sum of all values in p should now equal 1. Verify this.

- Calculate `S`, the number of species: this is simply the length of `p`. (Answer: S=126)

- Estimate the values of the diversity indices $N1$ and $N2$ and Ni, given by the following formulae:

$$N1 = \exp\left(\sum -p_i \cdot \log_e(p_i)\right)$$
$$N2 = 1/\left(\sum p_i^2\right)$$
$$N_\infty = 1/\max(p)$$

  You can calculate each of these values using only one R statement ! (A: 90.15358, 66.77841, 22.56157)

- The 126 nematode species per 10 cm^2 were obtained by looking at all 699 individuals. Of course, the fewer individuals are determined to species level, the fewer species will be encountered. Some researchers determine 100 individuals, other 200 individuals. To standardize their results, the expected number of species in a sample can be recalculated based on a common number of individuals. The expected number of species in a sample with size n, drawn from a population which size N, which has S species is given by:

$$ES(n) = \sum_{i=1}^{S}\left[1 - \frac{\binom{N-Ni}{n}}{\binom{N}{n}}\right]$$

  where Ni is the number of individuals in the i^{th} species in the full sample and is the so-called "binomial coefficient", the number of different sets with size n that can be chosen from a set with total size N.

  In R , binomial coefficients are estimated with statement `choose(N, n)`.

  What is the expected number of species per 100 individuals ? (n=100,N=699). (A: ES(100) = 60.68971).

- Print all diversity indices to the screen, which should look like:

```
        N        N0        N1        N2        Ni       ESS
699.00000 126.00000  90.15358  66.77841  22.56157  60.68971
```

5. Functions and flow control

Computers are very good at performing repetitive tasks. Yet, without proper programming tools, such tasks would require an endless repetition of commands, and so scripts would rapidly become very lengthy. To deal with this problem, R offers *loop control*, and more importantly, it enables that users can write their own functions. The goal of such a *user-defined function* is to put a large set of operations under one single heading, so these operations can be executed by one single statement. The more the same set of operations is repeated, the more it pays off to encapsulate these commands into an R function. As it happens, these user-defined function are one of the most powerful features of R . Therefore, it is extremely important to learn how to correctly create and implement functions in R scripts, as they provide a convenient and powerful tool to write elegant code.

5.1. Functions

Suppose that every time a statitician wants to use the normal distribution, he needs to write down the lengthy formula for the normal probability density distribution. Clearly, this would be rather tiresome. Fortunately, R provides the `dnorm()` function, which offers a very compact way to specify the normal distribution. A user-defined function operates in a similar way as a built-in function like `dnorm()`. The major difference is that one first has to create the function, before one can apply it. The creation of a user-defined function is done by an assignment of the form

```
function.name <- function(arg1, arg2, ...) {
  expression ...
  result <- ...
  return (result)
}
```

The function acts as a processor of information. It requires a given input information, which is specified by the list of *arguments* (`arg1`, `arg2`,...). Based on this input information, the function will calculate a final output (`result`), which is then passed on (via the `return` statement).

After submitting a function to R console, the function is stored as an object in the workspace. Once this has occurred, the function can be used at any point in the script by the function call command:

```
function.name(arg1 = x1, arg2 = x2, ...)
```

In this, `x1` and `x1` provide the actual value for the arguments `arg1` and `arg2`. When initializing the argument of a function, one must always use the "=" operator and not the assignment arrow "<-".

To illustrate this, suppose that we need to calculate the surface area for six spheres, whose radius varies as `R = 1 ,2, 4, 8, 16, 32`. One possibility would be to write a sequence of statements like:

```
R <- 1; 4 * pi * R^2
R <- 2; 4 * pi * R^2
...
```

This repetition of statements makes the script lengthy, and moreover, when copying a formula many times, it's easy to make a copy-and-paste mistake. A far more elegant solution is to create a function:

```
Sphere.area <- function (radius)  {
  area <- 4 * pi * radius^2
  return (area)
}
```

The name of the function is `Sphere.area` and it uses the variable `radius` as its single argument. After submitting this function definition to the R console, we can use it to calculate the surface area of the requested spheres. For the first sphere, this becomes:

```
Sphere.area(radius = 1)
```

```
[1] 12.56637
```

Note how the functions works. First the argument `radius` is initialized with value 1. Subsequently, the function expression is evaluated (the formula for the surface area is applied) and stored in the local object `area`. The content of `area` is returned as the final result.

We can now calculate the surface area for all spheres at once, and assign the result of this calculation to a vector

```
R <- c(1, 2, 4, 8, 16)
Area <- Sphere.area(radius = R)
```

This nicely illustrates how a user-defined function allows to write code that has fewer statements. Moreover, because one has automated the calculation, the end result is also less prone to errors.

When calling a function, it is not strictly needed to explicitly specify the name of the arguments. The following also works:

```
Sphere.area(1)
```

```
[1] 12.56637
```

However, if one does not write the argument names, the order in which the arguments are specified should be respected. This is never a problem when there is only one argument, but when there are multiple arguments, the order of the arguments matters. To illustrate this, let's define a function for the surface area of a cylinder.

```
Cylinder.area <- function (radius, height) {
  area <- pi * radius^2 * height
  return (area)
}
```

Check which of the following statements actually calculates the area of a cylinder with radius of 2 m and a height of 1 m.

```
Cylinder.area(1, 2)
Cylinder.area(2, 1)
Cylinder.area(radius = 1, height = 2)
Cylinder.area(height = 2, radius = 1)
```

The following function exactly mimics the built-in `mean()` function

```
mean.mimic <- function (x) {
  N <- length(x)
  result <- sum (x)/N
  return (result)
}
```

One can verify that `mean()` and `mean.mimic()` indeed provide the same result:

```
mean(0 : 10)
```

```
[1] 5
```

```
mean.mimic(0 : 10)
```

```
[1] 5
```

The curly braces {...} delineates a group of expressions. The value of the group as a whole is the result of the last expression. Such grouped expressions are particularly useful in functions, where only the last result of a whole series of calculations needs to be passed on. To ensure that the proper result is returned, it is advisable that the `return()` statement is explicitly stated. The evalution of the grouped expression inside the function is stopped as soon as `return()` is called. If the end of a function is reached without calling return, the value of the last evaluated expression is returned. For instance, the same function definition can be written in shorthand form as :

```
mean.mimic <- function (x) sum(x)/length(x)
```

The result returned by a function can be a more complicated object than just one element or a vector. One can also return a `list` or `data.frame`. The next function calculates the surface area and the volume of a sphere:

```
Sphere <- function(radius) {
 volume  <- 4/3 * pi * radius^3
 area    <- 4 * pi * radius^2
 result <- list(volume = volume, area = area)
 return (result)
}
```

The earth has an approximate radius of 6371 km, so its volume (km^3) and surface area (km^2) are:

```
Sphere(radius = 6371)
```

```
$volume
[1] 1.083207e+12

$area
[1] 510064472
```

The next statement will only display the volume of a sequence of spheres with radius 1, 2, ... 5

```
Sphere(radius = 1:5)$volume
```

```
[1]   4.18879  33.51032 113.09734 268.08257 523.59878
```

Often it is convenient to provide default values for the input parameters. For instance, the next function estimates the density of "standard mean ocean water" (in $kg\ m^{-3}$), as a function of the temperature T, (assuming salinity = 0 and pressure = 1 atm) (Millero and Poisson 1981). The argument `T` is set by default equal to the value 20 °C:

```
Rho_W <- function(T = 20) {
  dens <- 999.842594 + 0.06793952 * T - 0.00909529 * T^2 +
       0.0001001685 * T^3 - 1.120083e-06 * T^4 + 6.536332e-09 * T^5
  return(dens)
}
```

By ending the first sentence with a "+" we made clear that the statement is not finished and continues on the next line. It would have been wrong to put the + on the next line. Calling this function without specifying the value for the temperature, the function uses the default value $T = 20$:

```
Rho_W()
```

```
[1] 998.2063
```

```
Rho_W(T = 20)
```

```
[1] 998.2063
```

```
Rho_W(0)
```

```
[1] 999.8426
```

```
Rho_W(T = c(5, 10, 15))
```

```
[1] 999.9668 999.7021 999.1016
```

5.2. Control statements

Like all high-level programming languages, R has a number of statements that control the execution flow, that is, the order in which statements are executed. (`!Control` opens a help file on this topic)

Repetitive execution: Loops

If we want a set of commands to be repeated several times, we can use a *loop* statement. The computer will execute the instructions in the loop a specified number of times or until a specific condition is met. Once the loop is complete, the computer moves on to the next line of code immediately following the loop. There are three type of loops: the `for`, the `while` and the `repeat` loops. These loop types are largely equivalent in the sense that a loop constructed using one type could also be constructed by the other two types.

The `for` loop iterates over a specified set of values. In the example below, the loop variable `i` takes on the values `(1,2,3)`:

```
for (i in 1:3)  {
  x <- c(i, 2*i, 3*i)
  print(x)
}
```

```
[1] 1 2 3
[1] 2 4 6
[1] 3 6 9
```

Again, the curly braces {...} group multiple statements that are executed in each iteration.

The `while` and `repeat` loop statements will execute until a specified condition is met.

```
i <- 1
while(i < 4) {
  x <- c(i, 2*i, 3*i)
  print(x)
  i <- i + 1
}
```

```
[1] 1 2 3
[1] 2 4 6
[1] 3 6 9
```

The `break` statement exits the loop and can be used to terminate the loop. This is the only way to terminate repeat loops. The `next` statement stops the current iteration and advances to the next iteration.

```
i <- 1
repeat {
  print(i)
  i <- i + 1
  if (i > 2) break
}
```

```
[1] 1
[1] 2
```

Note: loops are implemented very inefficiently in R and should be avoided as often as possible. Fortunately, R offers many high-level commands that operate on whole vectors and matrices, thus it is often not necessary to use loops.

Conditional execution: if, else, ifelse constructs

These constructs only will execute statements if some condition is met. Try to understand the following:

```
Number.test <- function (x) {
  if ( x < 0  ) string <- "negative"                  else
  if ( x < 2  ) string <- "positive, smaller than 2"  else
                string <- "larger or equal than 2"
  print(string)
}
```

```
Number.test(-1)
```

```
[1] "negative"
```

```
Number.test(1)
```

```
[1] "positive, smaller than 2"
```

```
Number.test(2)
```

```
[1] "larger or equal than 2"
```

Note that we have specified the `else` clause on the same line as the `if` part so that R knows that the statement is continued on the next line!

The conditions that are used in `if/else` statements must evaluate to a single logical value. The `ifelse(test, yes, no)` construct is a vectorized version (see `?ifelse`).

```
x <- c(-1, 1, 2)
ifelse (x > 0, "strictly positive", "negative")
```

```
[1] "negative"          "strictly positive" "strictly positive"
```

For more information about `if` constructs and loops, check out the help page `?Control`.

5.3. Exercises.

Ex 01: R-function sphere. Extend the `Sphere` function with the circumference of the sphere at the place of maximal radius. The formula for estimating the circumference of a circle with radius r is: $2 \cdot \pi \cdot r$. What is the circumference of the earth near the equator?

Ex 02: Mimicking the var function. Write a function that mimics the effect of the variance function `var()`. The variance is calculated as: $var = \sum (x - \mu)^2/(N - 1)$ where N is the number of datapoints in the dataset and μ is the arithmetic mean. (tip: use the function `mean` to calculate μ). Calculate the variance of the data sequence (1, 2, ..., 9, 10).

(A: 9.166667)

Ex 03: An R-function to estimate saturated oxygen concentrations. The saturated oxygen concentration in water ($\mu mol\ kg^{-1}$), as function of temperature (T), and salinity (S) can be calculated by: $SatOx = e^A$ where :

$$\begin{aligned} A &= -173.9894 + 25559.07/T + 146.4813 * ln(T/100) - 22.204T/100 + \\ &\quad S(-0.037362 + 0.016504T/100 - 0.0020564T/100 * T/100) \end{aligned}$$

and T is temperature in Kelvin (Tkelvin = Tcelsius + 273.15).

Tasks:

- Make a function that implements this formula; the default values for temperature and salinity are 20°C and 35 respectively.
- What is the saturated oxygen concentration at the default conditions? (A: 225.2346)
- Estimate the saturated oxygen concentration for a range of temperatures from 0 to 30 °C, and salinity 35. (Tip: no need to use loops). Plot this function as an x-y plot.

Ex 04: Fibonacci numbers The Fibonacci numbers are calculated by the following relation: $F_n = F_{n-1} + F_{n-2}$

with $F_1 = F_2 = 1$

Tasks:

- Compute the first 50 Fibonacci numbers; store the results in a vector (use the R-command `vector` to create it). You will need to use a loop here.
- For large n, the ratio F_n/F_{n-1} approaches the "golden mean": $(1 + \sqrt{5})/2$
- What is the value of F_{50}/F_{49}; is it equal to the golden mean?
- When is n large enough? (i.e. sufficiently close ($<1e^{-6}$) to the golden mean)

Ex 05: Diversity of deep-sea nematodes for all stations

- Starting from your code to estimate diversity indices for deep-sea station M160b (see chapter 4.6), now write a loop that does so for all the stations in data.frame `Nemaspec`.
- First create a matrix called div, with the number of rows equal to the number of deepsea stations, and with 6 columns, one for each diversity index. This matrix will contain the diversity values.
- The column names of div are: "N", "N0", "N1", "N2", "Ninf", "ESS"

  The row names of matrix div are the station names (= the column names of Nemaspec). Tip: Use R-command `colnames()`, `rownames()`
- Now loop over all columns of data frame Nemaspec, estimate the diversity indices and put the results in the correct row of matrix div:

```
for (i in 1:ncol(Nemaspec))  {
   # you have to write this part of the code
}
```

- Show matrix div to the screen

6. Statistics

R originated as a statistical package, and it is still predominantly used for this purpose.

You can do virtually any statistical analysis in R.

As there exist many documents that may help you with statistical analyses in R, we will not deal with the subject here.

Statistics is used just to show you how to use efficiently use R, in cases where you have no clue where to begin!

6.1. Using R in four steps

Suppose you want to perform a hierarchic clustering and plot the dendrogram of a multivariate data set.

If you have never done that in R here are the steps:

1. Find a function that performs the requested task.

   for instance, use `help.search ("cluster")` to help you.

   Depending on the number of packages that you have installed, R will list a number of possible functions whose help file contains the word "cluster". Use the function from the package **stats** (part of the core of R).

2. Open the help file (?<name-of-the-function>) and look up the syntax for this function. If you have no time to read it completely, at least read (part of) the section "Description", "Usage", and "Examples".

3. Try the examples in the help file. You may:

   - Try them all at once (example(<name-of-the-function>).
   - Alternatively, you may select the statements in the Examples section that look applicable to your problem, copy-paste them into your script file (Ctrl-C / Ctrl-V) and execute them; e.g. line by line. Chances are real that, if they are suited for your case, you will transform them anyway.

4. Transform a promising example so that it suits your problem.

6.2. Exercise: multivariate statistics on the nematode species data.

Use R to perform a multivariate statistical analysis of the nematode data.

Beware: the nematode data have stations as columns and species as rows.

- Perform a hierarchic clustering and plot the dendrogram
- Perform a principal component analysis (PCA) and plot the results; you may also repeat the PCA analysis, with the first two stations removed!

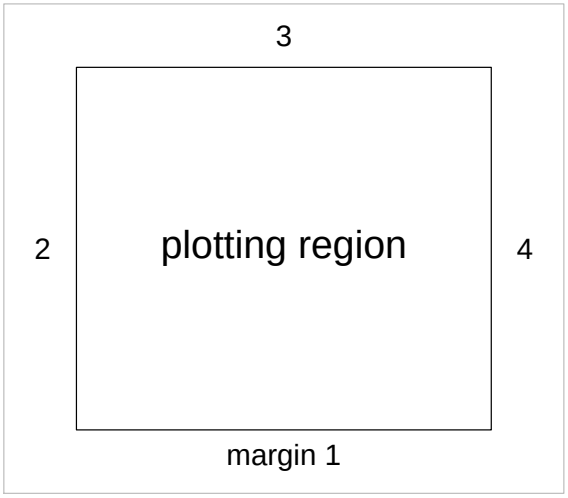


Figure 3: The four (inner) margins of an R figure and the plotting region.

7. Graphics

R has extensive graphical capabilities, and allows making simple (1-D, x-y), image-like (2-D) and perspective (3-D) figures.

Try:

```
> demo(graphics)
> demo(image)
> demo(persp)
```

to obtain a display of R's simple (1-D, x-y), image-like (2-D) and perspective (3-D) capabilities.

Graphics are plotted in the figure window which floats independently from the other windows. If not already present, it is launched by writing (in windows):

```
> windows()
```

or

```
> x11()
```

A figure consists of a plot region surrounded by 4 margins, which are numbered clockwise, from 1 to 4, starting from the bottom (figure 3). R distinguishes between:

1. high-level commands. By default, these create a new figure, e.g.
   - `hist`, `barplot`, `pie`, `boxplot`, ...
   - `plot`, `curve`, `matplot`, `pairs`,... ((x-y)plots)
   - `image`, `contour`, `filled.contour`,... (2-D surface plots)
   - `persp`, `scatterplot3d`,... (3-D plots) [1].
2. low-level commands that add new objects to an existing figure, e.g.
   - `lines`, `points`, `segments`, `polygon`, `rect`, `text`, `arrows`, `legend`, `abline`, `locator`, `rug`, ... These add objects within the plot region
   - `box`, `axis`, `mtext` (text in margin), `title`, ... which add objects in the plot margin
3. graphical parameters that control the appearance of.
   - plotting objects:
     `cex` (size of text and symbols), `col` (colors), `font`, `las` (axis label orientation), `lty` (line type), `lwd` (line width), `pch` (the type of points),...
   - graphic window:
     `mar` (the size of the margins), `mfrow` (the number of figures on a row), `mfcol` (number figures on a column), ...

```
> ?plot.default
> ?par
> ?plot.window
> ?points
```

will open the help files, while

```
> example(plot.default)
> example(points)
```

will run the examples, displaying each new graph, after you have pressed "<ENTER>" (try it!)

7.1. X-Y plots

A circle can be plotted by (x,y) points, where $x = r \cdot \cos(a)$ and $y = r \cdot \sin(a)$, with a the angle, from 0 to 2π, and r the radius. In the following script, we first generate a sequence of angle values, a, from 0 to 2π, comprising 100 values (`length.out`) and then plot a circle with unit radius:

```
a <- seq(from = 0, to = 2*pi, length.out = 100)
plot(x = cos(a), y = sin(a))
```

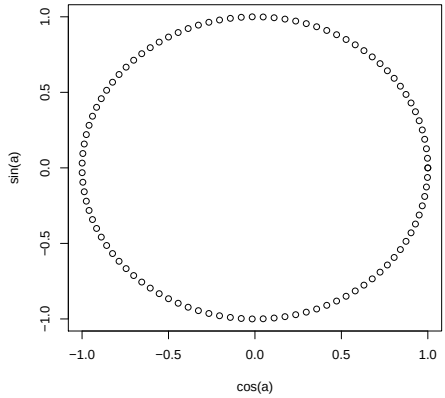


Figure 4: Simple figure with `plot` - see text for R-code

As `plot` is a high-level command, it will start a new figure. By default, R adds axes, and labels, and represents the (x,y) data as small dots (points). Note that the graph is not symmetrical (it is not a circle).

We will now make a more complex figure that resembles a "target face", e.g. for practicing archery or to throw darts. We first use the same command (`plot`) as above, but we add a number of graphical parameters specifying that:

- Rather than dots, the points should be connected by lines (`type`).
- The line should be twice as wide as the default (`lwd`)
- The x- and y-axes labels (`xlab`, `ylab`) have to be empty
- The axes and axes annotations (`axes`) are removed
- The graph has to be symmetrical, i.e. the x/y aspect ratio = 1 (`asp`).

```
plot(cos(a), sin(a), type = "l", lwd = 2, xlab = "", ylab = "",
     axes = FALSE, asp = 1)
```

To this figure, we can now add several low-level objects:

- a series of lines, representing smaller and smaller circles (`lines`).

```
for (i in seq(from = 0.1, to = 0.9, by = 0.1)) lines(i*sin(a), i*cos(a))
```

- an innermost red polygon (`polygon`).

[1] scatterplot3d is in R package **scatterplot3d** which has to be loaded first

```
polygon(sin(a)*0.1, cos(a)*0.1, col = "red")
```

- point marks as text labels, ranging from from 10 to 1 (`text`). The closer to the centre, the higher the score

```
for (i in 1:10) text(x = 0, y = i/10-0.025, labels = 11-i, font = 2)
```

- Now two archers take 10 shots at the target face.

  We mimic their arrows by generating normally distributed (x,y) numbers, with mean=0 (the centre!) and where the experience of the archer is mimicked by the standard deviation. The more experienced, the closer the arrows will be to the centre, i.e. the lower the standard deviation. R-statement `rnorm` generates normally distributed numbers; we need 20 of them, arranged as a matrix with 2 columns (the x and y values).

```
shots1 <- matrix(ncol = 2, data = rnorm(n = 20, sd = 0.2))
shots2 <- matrix(ncol = 2, data = rnorm(n = 20, sd = 0.5))
```

- The shots are added to the plot as points, colored darkblue (experienced archer) and darkgreen (beginners level). Note that we choose a 50% enlarged point size (cex), and we choose a circular shaped point (`pch = 16`)

```
points(shots1, col = "darkblue",  pch = 16, cex = 1.5)
points(shots2, col = "darkgreen", pch = 16, cex = 1.5)
```

- Finally, we add a legend, explaining who has done the shooting:

```
legend("topright", legend = c("Karline", "Filip"), pch = 16,
       col = c("darkblue", "darkgreen"), pt.cex = 1.5)
```

Note that the legend text and the colors are inputted as a vector of strings, using the `c()` function (e.g. `c("A", "B")`).

7.2. X-Y plots; conditional plotting

As a more sophisticated demonstration of the use of symbols in R-graphs, we work on a biological example, from the R data set called "Orange". This data set contains the circumference (in mm, at breast height) measured at different ages for five orange trees. We start by looking at the data (only the first and last part is displayed).

```
head(Orange)
```

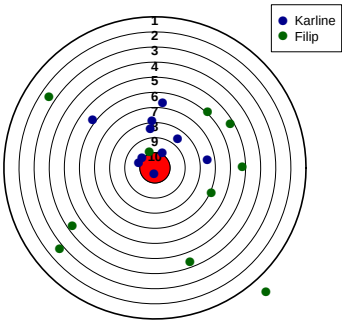


Figure 5: Figure with several low-level objects - see text for R-code

```
  Tree  age circumference
1    1  118            30
2    1  484            58
3    1  664            87
4    1 1004           115
5    1 1231           120
6    1 1372           142
```

```
tail(Orange)
```

```
   Tree  age circumference
30    5  484            49
31    5  664            81
32    5 1004           125
33    5 1231           142
34    5 1372           174
35    5 1582           177
```

and make a rough plot of circumference versus age:

```
plot(x = Orange$age, y = Orange$circumference, xlab = "age, days",
     ylab = "circumference, mm", main =  "Orange tree growth")
```

As Orange is a dataframe, columns can be addressed by their names, `Orange$age` and `Orange$circumference`.

The output (Fig. 6) shows that there is a lot of scatter, which is due to the fact that the five trees did not grow at the same rate.

It is instructive to plot the relationship between circumference and age differently for each tree. In R this is simple: we can make some graphical parameters (symbol types, colors, size,...) conditional to certain "factors".

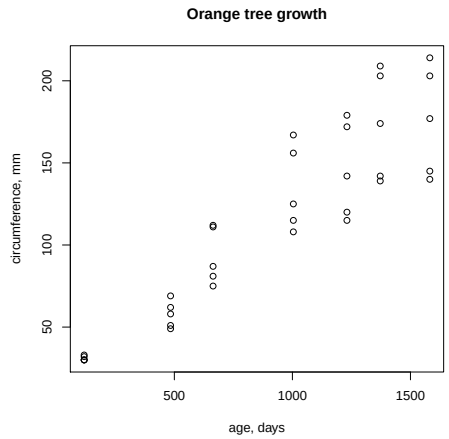


Figure 6: Simple plot of the `orange` dataset - see text for R-code

Factors play a very important part in the statistical applications of R; for our application, it suffices to know that the factors are integers, starting from 1.

In the R statement below, we simply use different symbols (`pch`) and colors (`col`) for each tree: `pch = (15:20)[Orange$Tree]` means that, depending on the value of `Orange$Tree` (i.e. the tree number), the symbol (`pch`) will take on the value 15 (tree 1), 16 (tree 2),... 20 (tree 5). The code `col = (1:5) [Orange$Tree]` does the same for the point color. The final statement adds a `legend`, positioned at the bottom, right.

```
plot(x = Orange$age, y = Orange$circumference, xlab = "age, days",
   ylab = "circumference, mm", main = "Orange tree growth",
   pch = (15:20)[Orange$Tree], col = (1:5)[Orange$Tree], cex = 1.3)
legend("bottomright", pch = (1:5)[order(levels(Orange$Tree))],
   col=(1:5)[order(levels(Orange$Tree))], legend = 1:5)
```

The output (Fig. 7) shows that tree number 5 grows fastest, tree number 1 is slowest growing. (note: it is also instructive to run the examples in the Orange help file.)

7.3. Zooplankton growth rates

`Zoogrowth` from package **marelacTeaching** is a literature data set, compiled by (Hansen, Bjornsen, and Hansen 1997) with measurements of maximal growth rates of zooplankton organisms as a function of body volume.

Run the example for this data set (you will need to load package **marelacTeaching** first):

```
> require(marelacTeaching)
> example(Zoogrowth)
```

7.4. Images and contour plots

R has some very powerful functions to create images and add contours. For example, the

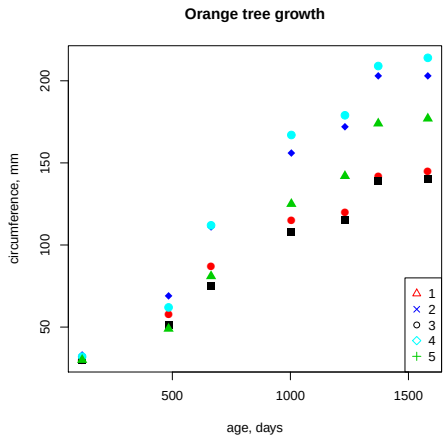


Figure 7: Improved plot of the `orange` dataset - see text for R-code

data set `Bathymetry` from the **marelac** package can be used to generate the bathymetry (and hypsometry) of the world oceans (and land):

```
require(marelac)
image(Bathymetry$x, Bathymetry$y, Bathymetry$z, col = femmecol(100),
      asp = TRUE, xlab = "", ylab = "")
contour(Bathymetry$x, Bathymetry$y, Bathymetry$z, add = TRUE)
```

Note the use of `asp = TRUE`, which maintains the aspect ratio.

7.5. Plotting a mathematical function

Plot curves for mathematical functions are quickly generated with R-command "curve":

```
> curve(expr = sin(3*pi*x))

> curve(expr = sin(3*pi*x), from = 0, to = 2, col = "blue",
      xlab = "x", ylab = "f(x)", main = "curve")
> curve(expr = cos(3*pi*x), add = TRUE, col = "red", lty = 2)
> abline(h = 0, lty = 2)
> legend("bottomleft", c("sin", "cos"),
  text.col = c("blue", "red"), lty = 1:2)
```

The first command will plot the curve for $y = sin(3 \cdot \pi \cdot x)$, using the default settings (Fig. 9 left), while the next commands first draw a graph of $y = sin(3 \cdot \pi \cdot x)$ in blue (`col`), and for x values ranging between 0 and 2 (`from`, `to`), adding a main title (`main`) and x- and y-axis labels (`xlab`, `ylab`) (1^{st} sentence).

The 2^{nd} R-sentence adds the function $y = cos(3 \cdot \pi \cdot x)$, as a red (`col`) dashed line (`lty`). Note the use of parameter `add = TRUE`, as by default curve creates a new plot.

The final statements adds the x-axis, i.e. a horizontal, dashed (`lty = 2`), line (`abline`) at y = 0 and a legend.

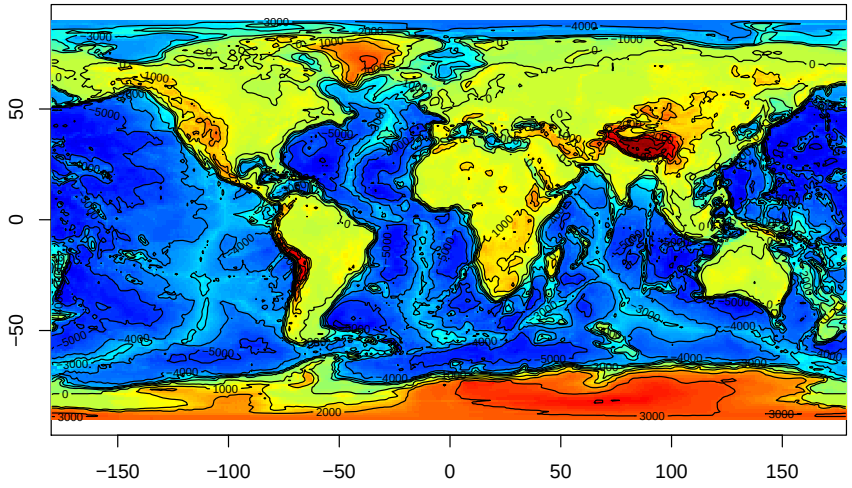


Figure 8: Image plot of ocean bathymetry - see text for R-code

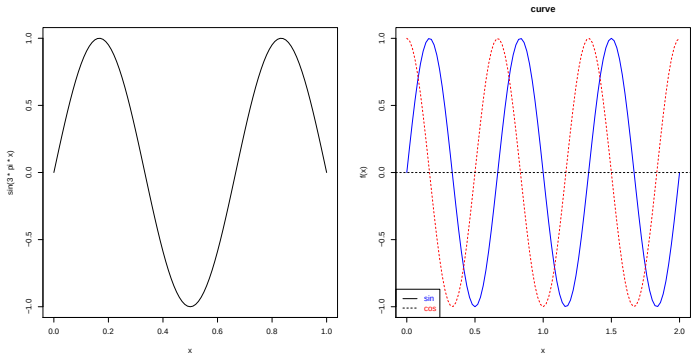


Figure 9: Plotting a mathematical function using `curve` - see text for R-code

7.6. Multiple figures

There are several ways in which to arrange multiple figures on a plot. The simplest is by specifying the number of figures on a row (`mfrow`) and on a column (`mfcol`):

```
> par(mfrow = c(3, 2))
```

will arrange the next plots in 3 rows, 2 columns. Graphs will be plotted row-wise.

```
> par(mfcol = c(3, 2))
```

will arrange the plots in 3 columns, 2 rows, in a columnwise sequence. Note that both `mfrow` and `mfcol` must be inputted as a vector. Try:

```
> par(mfrow = c(2, 2))
> for ( i in 1:4) curve(sin(i*pi*x), 0, 1, main = i)
```

The R-function `layout` allows much more complex plot arrangements. But it is significantly more difficult to use.

7.7. Histograms, boxplots

The data set `morley` contains the results of a classical experiment of Michaelson and Morley on the speed of light. The data comprise five experiments, each consisting of 20 consecutive "runs". The response is the speed of light measurement.

We first look at the data:

```
head(morley)
```

```
    Expt Run Speed
001    1   1   850
002    1   2   740
003    1   3   900
004    1   4  1070
005    1   5   930
006    1   6   850
```

We will plot 4 figures, arranged in two rows:

```
par(mfrow = c(2, 2))
```

Then, we plot a histogram of the given measurements:

```
hist(morley$Speed, xlab = "speed of light",
     main = "Michaelson-Morley")
```

The same data can also be visualised as a "density plot", or a Box-and-whisker plot:

```
plot(density(morley$Speed), xlab = "speed of light",
     main = "Michaelson-Morley")
boxplot(morley$Speed, ylab = "speed of light",
     main = "Michaelson-Morley")
```

Box-and-whisker plots are also convenient ways to see whether a certain treatment has an impact. To make such a Box-and-whisker plot, we need to use the "formula" representation that is often used in R. In this representation, e.g. `y~group` means that `y` should be expressed as a function of the groups in x.

For instance, to visualise whether the experiment had an effect on the speed-of-light measurements, we make a boxplot depicting the speed as a function of the experiment (`Expt`):

```
boxplot(Speed ~ Expt, data = morley, ylab = "speed of light",
     main = "Speed of Light", xlab = "Experiment No.")
```

7.8. Exercises

Ex01: Plotting data, manual input The following oxygen concentrations were measured at hourly intervals, starting at 8 o'clock, from the jetty near our institute:

(210, 250, 260, 289, 280, 260, 270, 260).

Make a plot that displays these data. Use large symbols, and label the axes appropriately.

Ex02: Human population growth The human population (N, millions of people) at a certain time t, can be described as a function of time (t), the initial population density at $t = t0$ (denoted by "N_{t0}"), the carrying capacity "K" and the rate of increase "a" by the following equation [2]:

$$N(t) = \frac{K}{1 + [\frac{K - N_{t0}}{N_{t0}}]e^{-a \cdot (t - t0)}}$$

For the US, the population density in 1900 (N_{t0}) was 76.1 million; the population growth can be described with parameter values: a=0.02 yr^{-1}, K = 500 million people.

Actual population values are:

1900	1910	1920	1930	1940	1950	1960	1970	1980
76.1	92.4	106.5	123.1	132.6	152.3	180.7	204.9	226.5

Tasks:

1. Plot the population density curve as a thick line, using the US parameter values.
2. Add the measured population values as points. Finish the graph with titles, labels etc...

Ex03: Simple curves

[2] This is the solution of a so-called logistic differential equation (Verhulst 1838)

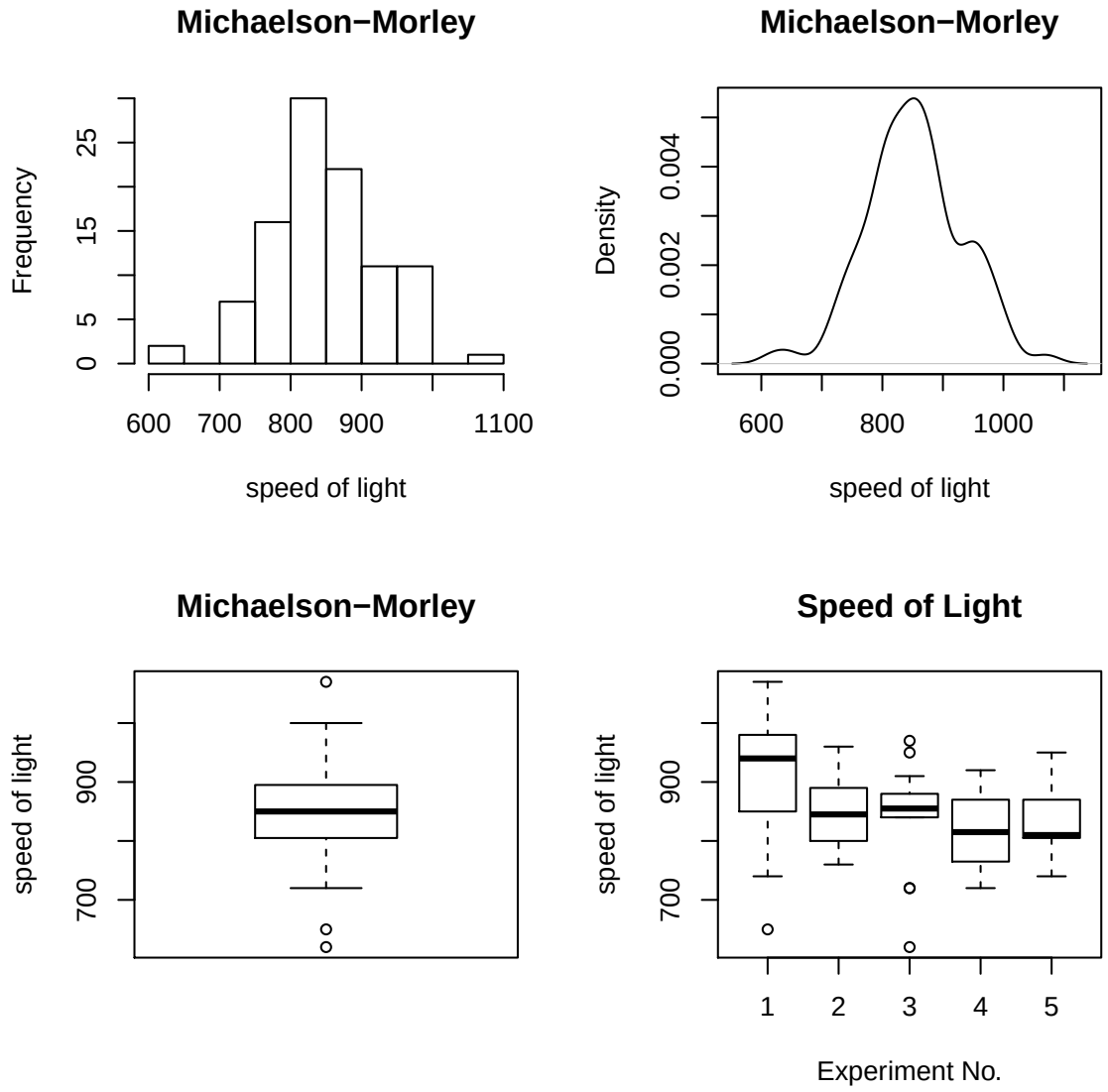


Figure 10: Four ways to look at the same data.

- Create a script file which draws a curve of the function $y = x^3 sin^2(3\pi x)$ in the interval [-2, 2].
- Make a curve of the function $y = 1/cos(1 + x^2)$ in the interval [-5,5].

Ex04: Toxic ammonia Ammonia nitrogen is present in two forms: the ammonium ion (NH_4^+) and unionized ammonia (NH_3). As ammonia can be toxic at sufficiently high levels, it is often desirable to know its concentration.

The relative importance of ammonia, (the contribution of ammonia to total ammonia nitrogen, $NH_3/(NH_3 + NH_4^+)$, is a function of the proton concentration $[H^+]$ and a parameter K_N, the so-called stoichiometric equilibrium constant:

$$p_{[NH_3]} = \frac{K_N}{K_N + [H^+]}$$

Tasks:

- Plot the relative fraction of toxic ammonia to the total ammonia concentration as a function of pH, where $pH = -log_{10}([H^+])$ and for a temperature of 30°C. Use a range of pH from 4 to 9.

  The value of K_N is 8 10^{-10} at a temperature of 30°C.
- Add to this plot the relative fraction of ammonia at 0°C; the value of K_N at that temperature is 8 10^{-11} $mol\ kg^{-1}$.

Ex05: The iris data set A famous data set that is part of R is the "iris" data set (Fisher, 1936), which we will explore next.

It gives measurements, in centimeters for sepal length and width and petal length and width, respectively, for 50 flowers of the species *Iris setosa*, *Iris versicolor* and *Iris virginica*.

Tasks:

- Have a look at the data:
- What is the class of the data set? why?
- What are the dimensions of the data set? (number of rows, columns)
- Produce a scatter plot of petal length against petal width; produce an informative title and labels of the two axes.
- Repeat the same graph, using different symbol colours for the three species.
- Add a legend to the graph. Copy-paste the result to a WORD document. If you do not have WORD, make a PDF file of the graph.
- Create a box-and-whisker plot for sepal length where the data values are split into species groups; use as template the first example in the "boxplot" help file.
- Now produce a similar box-and-whisker plot for all four morphological measurements, arranged in two rows and two columns. First specify the graphical parameter that arranges the plots two by two.

Ex06: Estuarine morphology

The Westerschelde estuary has a trumpet-shaped morphology, i.e. its cross-sectional area increases in a sigmoidal fashion from Rupelmonde near the river towards Vlissingen near the sea.

The estuary is 100 km long, and the cross-sectional surface, $A(x)$, in m^2, can be approximated with the following equation:

$$A(x) = A_r + dA * \frac{x^p}{x^p + {k_s}^p}$$

where $dA = A_s - A_r$, p = 5, ks = 50000 m, $A_r = 4000m^2$, $A_s = 76000m^2$. Here A_r and A_s are the cross-sectional surfaces at the boundary with the `r`iver and the `s`ea respectively. Plot the cross-surface area of the Scheldt estuary, as a function of distance from the river

Ex07: North American Rivers The data set `rivers` gives the length, in miles, of 141 "major" rivers in North America, as compiled by the US Geological Survey.

Check the contents of `rivers`. Create a new vector, called `rivers.km`, which has the length of the rivers, in kilometres. One mile = 1.609344 km.

Make a histogram of the length of the rivers; use 20 cells for the histogram.

The rivers Scheldt and Seine are respectively 350 and 777 km long, while the Amazon river is 4000 miles long. Add this information to the previous histogram. You can use a symbol at the correct position (`points`), or write the name of the river at the correct position (`text`). Use your imagination!

Ex08: CO2 and grasses

The data set `CO2` lists the CO2 uptake rates from a grass species as a function of ambient carbon dioxide concentration, and for two different temperature treatments.

Look at the contents of this data set.

Use a boxplot to visually inspect whether there is an effect of (1) ambient concentration and (2) temperature treatment on the CO2 uptake rate.

8. Matrix algebra

A matrix is a combination of vectors with the same length.

8.1. Creating a matrix

Matrices can be created in several ways:

- By means of the R function `matrix`.
- By means of the R function `diag` which constructs a diagonal matrix.
- The functions `cbind` and `rbind` add columns and rows to an existing matrix, or to another vector.

Consider the matrix

$$\begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}$$

To enter this matrix in R, we first write it as a single vector, going down each column, i.e., the data are: `c(1, 2, 3, 4, 5, 6, 7, 8, 9)`. The matrix has 3 rows (`nrow`) and 3 columns (`ncol`). We can create the matrix, and put the result in variable `MAT` as follows:

```
MAT <- matrix(nrow = 3, ncol = 3, data = 1:9)
```

Built-in functions operate on matrices in an element-wise fashion (like they operate on vectors). The next two statements create a matrix `A`, display the matrix followed by the square root of its elements:

```
A <- matrix(nrow = 2, data = 1:4)
A
```

```
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

```
sqrt(A)
```

```
         [,1]     [,2]
[1,] 1.000000 1.732051
[2,] 1.414214 2.000000
```

By default, R fills a matrix column-wise (see the examples above). However, this can easily be overruled, using the argument `byrow`:

```
(M <- matrix(nrow = 4, ncol = 3, byrow = TRUE, data = 1 : 12))
```

```
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
[4,]   10   11   12
```

The unity matrix (I) is created using the R-function `diag`:

```
diag(1, nrow = 2)
```

```
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

The *names* of columns and rows are set as follows:

```
rownames(A) <- c("x", "y")
colnames(A) <- c("c", "b")
A
```

```
  c b
x 1 3
y 2 4
```

Note that we also use the `c()` function to create these names ! Row names and column names are in fact vectors containing "strings".

Matrices can also be created by combining (binding) vectors, e.g. rowwise:

```
V <- 0.5 : 5.5
sqrt.V <- sqrt(V)
rbind(V, sqrt.V)
```

```
            [,1]     [,2]     [,3]     [,4]    [,5]     [,6]
V      0.5000000 1.500000 2.500000 3.500000 4.50000 5.500000
sqrt.V 0.7071068 1.224745 1.581139 1.870829 2.12132 2.345208
```

As one can see, the names of the vectors are automatically assigned as row names.

8.2. Arrays

Arrays are multidimensional generalizations of matrices. We will not often use them, and so they are given here only for completeness. A multi-dimensional array is created as follows:

```
AR <-array(dim = c(2, 3, 2), data = 1)
```

In this case `AR` is a 2*3*2 array, and its elements are all 1.

8.3. Dimensions

The commands

```
> length(V)
> dim(A)
> ncol(M)
> nrow(M)
```

will return the dimension of the matrix or array `A`, and the number of columns and rows of matrix `M` respectively.

8.4. Selecting and extracting elements

To select subsets of a matrix, we can either

- specify the numbers of the elements that we want (simple indexing)
- specify a vector of logical values (TRUE/FALSE) to indicate which elements to include (TRUE) and which not to include (FALSE). This uses logical expressions

As was the case with vectors, the elements of a matrix or array are indexed using the square brackets `[]` operator. Accordingly, individual elements can be extracted from a matrix `A` by using `A[i,j]`, which extracts the element in the i^{th} row and j^{th} column of `A`.

```
M <- matrix(nrow = 4, ncol = 3, byrow = TRUE, data = 1:12)
M[1, 2]
```

```
[1] 2
```

Sometimes we want to extract an entire column or entire row from a matrix. Assume that we want to see the entire second column of matrix `M`. One way to request that information is:

```
M[1:4, 2]
```

```
[1]  2  5  8 11
```

But it is very easy to make mistakes this way. A much more elegant way is to leave the row indices blank, which tells R that it needs to select all elements.

```
M[  , 2]
```

The following statement selects the first and third row of `M`:

```
M[c(1, 3), ]
```

```
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    7    8    9
```

and the entire last column of `M`.

```
M[ ,ncol(M)]
```

```
[1]  3  6  9 12
```

If an index is omitted, then all the rows (1^{st} index omitted) or columns (2^{nd} index omitted) are selected. In the following:

```
M[   ,2] <- 0
M[1:3, ] <- M[1:3, ] * 2
```

first all the elements on the 2^{nd} column (1^{st} line) of `M` are zeroed and then the elements on the first three rows of `M` multiplied with 2 (2^{nd} line).

8.5. Calculations with matrices

Matrix algebra is very simple in R . Practically everything is possible! Here are the most important R functions that operate on matrices:

- `%*%` Matrix multiplication
- `t(A)` transpose of `A`
- `diag(A)` diagonal of `A`
- `solve(A)` inverse of `A`
- `solve(A,B)` solving $Ax = B$ for x
- `eigen(A)` eigenvalues and eigenvectors of `A`
- `det(A)` determinant of `A`

For instance the following first inverts matrix `A` (`solve(A)`), and then multiplies the inverse with `A` , giving the unity matrix:

```
(A <-matrix(nrow = 2, data = c(1, 2, 3, 4)))
```

```
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

```
solve(A) %*% A
```

```
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

whilst `t(A)` will transpose matrix `A` (interchange rows and columns).

```
t(A)
```

```
     [,1] [,2]
[1,]    1    2
[2,]    3    4
```

The next set of statements will solve the linear system $Ax = b$ for the unknown vector x:

```
b <- c(5, 6)
solve(A, b)
```

```
[1] -1  2
```

Finally, the eigenvalues and eigenvectors of `A` are estimated using the R function `eigen`. This function returns a list that contains both the eigenvalues (`$values`) and the eigenvectors (`$vectors`), (the columns).

```
eigen(A)
```

```
$values
[1]  5.3722813 -0.3722813

$vectors
           [,1]       [,2]
[1,] -0.5657675 -0.9093767
[2,] -0.8245648  0.4159736
```

8.6. Exercises

Matrix algebra exercise 1

- Use the R function `matrix` to create a matrix with the following contents:

$$\begin{bmatrix} 3 & 9 \\ 7 & 4 \end{bmatrix}$$

- display it to the screen
- Use R function `matrix` to create a matrix called "`A`":

$$\begin{bmatrix} 1 & 1/2 & 1/3 \\ 1/4 & 1/5 & 1/6 \\ 1/7 & 1/8 & 1/9 \end{bmatrix}$$

  - Take the transpose of `A`.
  - Create a new matrix, `B`, by extracting the first two rows and first two columns of `A`. Display it to the screen.

- Use `diag` to create the following matrix, called "`D`":

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{bmatrix}$$

- Use `cbind` and `rbind` to augment this matrix, such that you obtain:

$$\begin{bmatrix} 1 & 0 & 0 & 4 \\ 0 & 2 & 0 & 4 \\ 0 & 0 & 3 & 4 \\ 5 & 5 & 5 & 5 \end{bmatrix}$$

  It is simplest to do this in two statements (but it can be done in one!)

- Remove the second row and second column of the previous matrix

Matrix algebra exercise 2

- Use the R function `matrix` to create the matrices called `A` and `B`:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 6 & 4 & 1 \\ -2 & 1 & -1 \end{bmatrix}, B = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

- Take the inverse of `A` and the transpose of `A`.
- Multiply `A` with `B`.
- Estimate the eigenvalues and eigenvectors of A.
- For a matrix `A`, x is an eigenvector, and λ the eigenvalue of a matrix `A`, if $A \cdot x = \lambda \cdot x$. Test it!

Matrix algebra exercise 3

- Create a matrix, called `P`:

$$\begin{bmatrix} 0 & 0.0043 & 0.1132 & 0 \\ 0.9775 & 0.9111 & 0 & 0 \\ 0 & 0.0736 & 0.9534 & 0 \\ 0 & 0 & 0.0452 & 0.9804 \end{bmatrix}$$

- What is the value of the largest eigenvalue (the so-called dominant eigenvalue) and the corresponding eigenvector?.
- Create a new matrix, `T`, which equals `P`, except for the first row, where the elements are 0.
- Now estimate $N = (I - T)^{-1}$, where I is the identity matrix [3].

System of linear equations

- Solve the following system of linear equations for the unknown x_i:

```
3x1 + 4x2 + 5x3 = 0
6x1 + 2x2 + 7x3 = 5
7x1 +  x2       = 6
```

- You will first need to rewrite this problem in the form: $Ax = B$, where A contains the coefficients, x the unknowns, and B the right-hand side values. Then you solve the system, using R-function `solve`
- Check the results (i.e. is $Ax = B$?)

[3]Note: this is a stage-model of a killer whale (Caswell 2001). The eigenvalue-eigenvectors estimate the rate of increase and stable age distribution, the matrix N contains the mean time spent in each stage.

9. Roots of functions

9.1. The root-finding problem

The goal of a root-finding procedure is to find a value for x such that

$$f(x) = 0$$

for a given function f. This value of x is called a "root" of the function f. For simple functions, it is often possible to find the root in an analytical way. For example, one can easily see that the root of the function $f(x) = 3-x^2$ must be equal to $x = \sqrt{3}$ or $x = -\sqrt{3}$. If we plug one of these two values for x in the formula, say $x = \sqrt{3}$, we obtain $f(3) = 3-(\sqrt{3})^2 = 0$. Mathematically, the root finding problem is equivalent to the solution of an algebraic equation. In the example above, we need to find the value of x for which the non-linear equation $3 - x^2 = 0$ holds.

In the case of simple functions, one is often able to provide an exact and explicit expression for the value of the root. However, for more complex and especially non-linear functions, such an analytical solution procedure is not possible, and hence, one must turn to numerical root solving methods. Suppose we need to determine the root of the non-linear function

$$f(x) = \cos(x) - 2x$$

Graphically, the root represents the location where the function $f(x)$ intersects the x-axis. So to start, it is always a good idea to plot the function over the interval where one expects the root to be found:

```
curve(expr = cos(x)-2*x, from = -2, to = 2)
abline(h = 0, lty = 2)
```

The first statement creates the plot by means of the `curve()` command (see Graphics chapter), while the second line adds the x-axis to the plot. This figure shows that there indeed exists a single root within the [-2, 2] interval located approximately at $x = 0.5$.

There are various algorithms for numerical root-finding. A common feature is that they all work "iteratively": they start from an initial guess, and subsequently they produce a sequence of values that (hopefully) converges towards the root (within some tolerance - see below). In each iteration step, a new and better estimate for the root is computed based on the previously obtained value, thus moving closer and closer to the root.

A number of different algorithms have been developed for root-finding, and each comes with its own intricacies and application domain. The best known methods are the bisection, secant and Newton-Raphson methods.

9.2. The bisection method

The bisection method is a basic root-finding method. It is simple and robust, but also rather slow. In real life problems, other methods will be preferred that are numerically more efficient. However, the bisection method nicely illustrates how a root algotithm works, and for this reason, we discuss it here. We start from an interval $[a, b]$ for which we know that it contains a root. The bisection method then divides this interval in two and selects the subinterval in

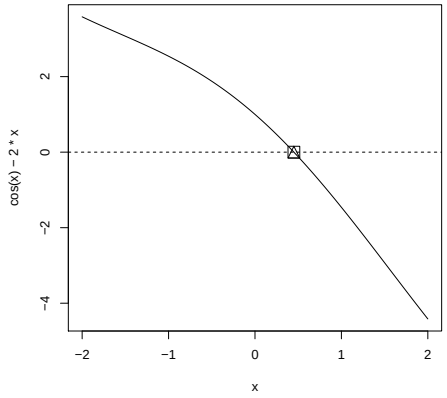


Figure 11: Function drawn with `curve` and the root of the function plotted - see text for R-code

which a root must lie. This procedure is repeated until the interval is sufficiently small (i.e. smaller than some preset tolerance). The bisection method can be implented by following function:

```
bisection <- function(f, interval, tol=.Machine$double.eps^0.25, maxiter=1000)
{
a <- interval[1]
b <- interval[2]

if (a >= b) stop("startpoint a  must be smaller than endpoint b")
if (f(a)*f(b) >= 0) stop("f(a) and f(b) must have a different sign")

i <- 1
while (i <= maxiter) #limit iterations to prevent infinite loop
  {
    mid <- (a + b)/2 # new midpoint
    estim.prec <- (b - a)/2 # estimated precision
    if (f(mid) == 0 | estim.prec < tol) # root found
      {
        if (f(mid) == 0) estim.prec <- 0
        result <- list(root=mid,f.root=f(mid),iter=i,estim.prec=estim.prec)
        return(result)
      }
    i <- i + 1 # step counter increases
    if (sign(f(mid)) == sign(f(a))) a <- mid else b <- mid
  }
stop(paste("Bisection method failed: max number of steps",maxiter,"exceeded"))
}
```

The bisection method requires four different types of input:

- `f`: the function for which the root needs to be found

- `interval`: the interval specified by the start point `a` and the end point `b`. The start point must be smaller than the end point, and the function f needs to have an opposite sign at these two points.
- `tol`: the requested tolerance on the root. If no value is specified, we use a default value based on the machine precision `.Machine$double.eps`
- `maxiter`: the maximum number of iterations. This criterion is specified to prevent that the algorithm gets stuck in an infinite loop.

At each step the bisection method calculates the midpoint of the interval and the function value at that midpoint. Unless the midpoint is itself a root (which is very unlikely, but possible) there are two possibilities: either f(a) and f(mid) have opposite signs and so the interval [a,mid] brackets the root, or f(mid) and f(b) have opposite signs and the interval [mid,b] brackets the root. The method selects the new interval to be used in the next step, so the interval that contains the root is halved.

It should be emphasized that the value of the root returned by a numerical root-finding method is always an approximation up to a certain accuracy. This accuracy, or equally the "error" tolerated, must be specified *a priori* by the user. In `bisection`, this is done via the facultative `tol` argument. The root finding iteration is stopped when successive changes of x are smaller than `tol`.

The bisection method generates four different types of output:

- `root`: the estimated value of the root
- `f.root`: the function value evaluated at the root
- `iter`: the number of iterations used
- `estim.prec`: an estimate of the precision for the root

```
test.f <- function(x) return(cos(x)-2*x)
(result <- bisection(f = test.f, interval = c(-2, 2)))
```

```
$root
[1] 0.4501343

$f.root
[1] 0.0001201335

$iter
[1] 16

$estim.prec
[1] 6.103516e-05
```

In theory, the function value `f.root` should be zero at the root, but since we are employing a numerical procedure, such absolute accuracy is not possible. Accordingly, the `$f.root` value should be sufficiently close to zero (compare `$f.root` to the function values at the start

and end of the search interval). The `$iter` value gives the number of iterations (or steps) that were needed to find the root, while `$estim.prec` provides an estimate of the precision with which the root is found. Accordingly, the last two decimal digits printed for `$root` are not meaningful, and so, the final outcome of the root calculation should be presented as 0.45013 ± 0.00006.

For most purposes one should not bother about the value of the tolerance `tol`, and use the default value based on the machine precision. However, for some applications, it can be useful to be less accurate, and hence, obtain a faster result with less iterations. Try the following.

```
test.f <- function(x) return(cos(x) - 2*x)
bisection(f = test.f, interval = c(-2, 2), tol = 0.01)
```

How do the root value and the required number of iterations change?

9.3. Roots of one-dimensional functions: uniroot

As noted above the bisection method is not very numerically efficient. In general, there is a trade-off between robustness (making sure that the algorithm converges towards the root) and the speed (miminizing the number of iterations and calculations to obtain the root). Robust methods are typically slow, while for fast methods, one is less sure that they will converge.

A more sophisticated root-finding method is implemented in R by the function `uniroot` from the **stats** package. This method will find roots for one-dimensional functions. For reference, the algorithm underlying `uniroot()` is Brent's method, which is a popular but complicated root-finding algorithm combining the bisection method, the secant method and inverse quadratic interpolation (Brent, 1973). In this hybrid method, the idea is to use the secant method or inverse quadratic interpolation if possible, because they converge faster, but to fall back to the more robust bisection method if necessary.

The statement below applies the `uniroot` method to our problem. As arguments one should provide an R function for which the root needs to be determined and the `interval` in which the root should be found. Here this function is called `test.f` (see Chapter on R functions). For the method to work, there should be at least one root in the interval.

```
test.f <- function(x) return(cos(x) - 2*x)
(result <- uniroot(f = test.f, interval = c(-2, 2)))
```

```
$root
[1] 0.4501842

$f.root
[1] -1.472704e-06

$iter
[1] 6

$estim.prec
[1] 6.103516e-05
```

The `uniroot` function returns a list with several output values in a similar way as teh `bisection` function. The most important one is the root itself (`$root`), which is 0.4501842. The value `$f.root` provides the function value at the root, which is $-1.47e^{-6}$. Note that the number of iterations (6) is much smaller when compared to the bisection method.

9.4. Multiple roots

The `uniroot` function only gives an unambiguous result when there is a only single root within the given interval. When there are more roots, the answer can be ambiguous. To see this, try:

```
f.root <- function (x)
  return( x^3 - 2*x )

uniroot(f = f.root, interval = c(-2, 2))$root
uniroot(f = f.root, interval = c(-10, 2))$root
uniroot(f = f.root, interval = c(-5, 2))$root
```

The polynomial function $f(x) = x^3 - 2 * x$ has three roots: $-\sqrt{2}$, 0 and $+\sqrt{2}$. Depending on the search interval that one chooses, a different root is selected by `uniroot`. Accordingly, when dealing with functions that have multiple roots, such as polynomials, one must be careful. There exist specific root-finding algorithms that determine all roots of a polynomial, which are not further discussed here. In R , this is done by the `polyroot()` function - consult its help file for more information. It is simpler to use R function `uniroot.all` from the R package **rootSolve**:

```
f.root <- function (x)
  return( x^3 - 2*x )
uniroot.all(f = f.root, interval = c(-2, 2))
```

```
[1]  0.000000 -1.414214  1.414214
```

9.5. Roots in multi-dimensional problems

For root finding in multi-dimensional problems, one should consult the methods in the package **rootSolve**.

9.6. A real world application: the pH of natural waters

The pH is a master variable for the chemical composition of natural waters. The pH of natural waters is most influenced by the carbonate buffer system, which protects the water against large pH changes. In ocean acidification studies, it is crucial to fully determine the speciation of the water, that is, the concentration of all seven chemical species involved in the carbonate buffer system. This means that one needs to determine the concentration of dissolved carbon dioxide ($[CO_2]$), bicarbonate ($[HCO_3^-]$), carbonate ($[CO_3^{2-}]$) and protons ($[H^+]$), as well the dissolved inorganic carbon (DIC) concentration, the titration alkalinity (TA) and the pH. The latter three quantities are defined as:

$$\mathrm{DIC} = [\mathrm{CO_2}] + [\mathrm{HCO_3^-}] + [\mathrm{CO_3^{2-}}]$$
$$\mathrm{TA} = [\mathrm{HCO_3^-}] + 2[\mathrm{CO_3^{2-}}] - [\mathrm{H^+}]$$
$$\mathrm{pH} = -\log([\mathrm{H^+}])$$

From the theory of aquatic chemistry, it is known that two out of the total set of seven variables need to be known to fully determine the speciation. When the proton and DIC concentrations are known, one can directly calculate the concentration of the proton and carbonate species, and subsequently, the titration alkalinity, by using the relations (Zeebe and Wolf-Gladrow 2003)

$$[\mathrm{CO_2}] = \frac{[H^+] \cdot [H^+]}{[H^+] \cdot [H^+] + K_{C1} \cdot [H^+] + K_{C1} \cdot K_{C2}} \cdot DIC$$
$$[\mathrm{HCO_3^-}] = \frac{K_{C1} \cdot [H^+]}{[H^+] \cdot [H^+] + K_{C1} \cdot [H^+] + K_{C1} \cdot K_{C2}} \cdot DIC$$
$$[\mathrm{CO_3^{2-}}] = \frac{K_{C1} \cdot K_{C2}}{[H^+] \cdot [H^+] + K_{C1} \cdot [H^+] + K_{C1} \cdot K_{C2}} \cdot DIC$$

However, in chemical oceanographic studies, one usually determines the DIC and TA, because these are more acccurately measured. Accordingly, the challenge is to calculate the other species concentrations from DIC and TA. This speciation problem essentially comes down to a root-finding problem, where one needs to solve for the proton concentration $[H^+]$ (or the pH value). The trick is to estimate the alkalinity based on the known DIC concentration and a guess of the proton concentration, and compare that with the measured value for the alkalinity. If both are equal within a preset tolerance level, the correct proton concentration has been found.

In the implementation below, the dissociation constants of the carbonate system (`kc1`, `kc2`) are first calculated as a function of temperature and salinity. This is done using the R package **AquaEnv**, which is dedicated to aquatic chemistry and has to be loaded first (`require`). All concentrations are in $\mu mol\ kg^{-1}$, while the dissociation constants are in $mol\ kg^{-1}$, so we need to introduce a conversion factor of 10^6.

We then define a function (`pHfunction`) for which the root has to be found. In this function we estimate the alkalinity, based on the guess of pH, the dissociation constants (`kc1`, `kc2`) and the measured DIC concentration. The difference between this calculated alkalinity (`Estimated.TA`) and the measured alkalinity is then returned. If the pH is correctly estimated, then the estimated and measured alkalinity should be equal. So, to find the pH, we need to find the root of the `pHfunction` within an interval between 0 and 12 (which covers the whole range found in nature).

```
DIC.measured <- 2130e-6
TA.measured <- 2350e-6
require(AquaEnv)
K1 <- K_CO2(S = 35, t = 20, p = 0)[1]
K2 <- K_HCO3(S = 35, t = 20, p = 0)[1]
pHfunction <- function(pH, K1, K2, DIC, TA)
{
```

```
   H    <- 10^(-pH)
   HCO3 <- H*K1  /(H*K1 + H*H + K1*K2)*DIC
   CO3  <- K1*K2 /(H*K1 + H*H + K1*K2)*DIC

   Estimated.TA  <- HCO3 + 2*CO3 - H
   return (Estimated.TA  - TA)
}
```

```
uniroot(pHfunction, interval = c(0, 12),
        K1 = K1, K2 = K2, DIC = DIC.measured, TA = TA.measured)
```

```
$root
[1] 8.191469

$f.root
[1] 4.151739e-10

$iter
[1] 7

$estim.prec
[1] 6.103516e-05
```

9.7. Exercises

Ex 01: Roots of simple functions

Use the function `uniroot` to find the roots of:

- the function $f(x) = x^2 - \sqrt(x) + 0.1$ in the interval [0.5, 2]. (Answer: 0.9298 +/- 0.0005)
- the equation $e^x = 4x^2$ in the interval [0, 1]. (Answer: 0.7148 +/- 0.0005)

Each time, first draw the function curve. First, use a tolerance of 0.001. Afterwards, calculate the root once more with the higher tolerance of 0.00001. How strongly does its value change?

Ex 02: Function evaluation

Find the value of x for which the function $f(x) = x^2 - \sqrt(x)$ becomes equal to 2. Use `uniroot` and explore the interval [0.5, 2.5]. (Answer: 1.83118 +/- 0.00006)

Ex 03: Dangerous global warming

Under the bussiness as usual scenario, the predicted increase in the mean global temperature over the 21 th century is given by the function

$\Delta T = a * (t - t_{ref}) + b * (t - t_{ref})^2$

where the $a = 0.0264$ and $b = 2.1E-4$. The year 1990 is the reference time t_{ref} of the Kyoto protocol. When do we exceed the level of 2 degree warming, after which climate change is considered dangerous? (Answer: the year 2043).

Ex 04: The Platt function

The response of photosynthesis to different light intensities is a critical component of ecosystem models. This relation is given by the PI curve (photosynthesis-irradiance), which is an empirical relationship that predicts the rate of photosynthesis P (in mg C (mg Chl)-1 h-1) as a function of the solar irradiance I (in μmol photons m-2 s-1). An often used PI curve is the so-called Platt model

$P(I) = \left(1 - \exp\left(\frac{-\alpha I}{P_{max}}\right)\right) \exp\left(\frac{-\beta I}{P_{max}}\right)$

The quantities α, β, and P_{max} are parameters. Typical values for these these parameters are $\alpha = 0.02$, $\beta = 0.01$, and $P_{max} = 5$

Tasks:

- Create an R function "Platt" that implements the above formula. Use the above parameter values as default arguments in this R function.
- Check your formula. For I = 100 and using the default values of the parameters, you should obtain the value of P = 0.2699191.
- Make a plot of the P-I curve over the range or I from 0 to 500.
- Find the irradiance for which the rate of photosynthesis is equal to 0.2 (Answer: 64.6)

Ex 05: A two-dimensional problem

Solve the equations $1000 = y * (3 + x) * (1 + y)^4$ for y and with x varying over the range from 1 to 100. Plot the root as a function of x.

Tip: first make a sequence of x-values, then loop over each "x" value, each time estimating the root and putting it in a vector.

Ex 06: Ocean acidification

The pH of natural waters can be estimated based on the measured alkalinity and pCO_2, that is the partial pressure of CO_2. To solve this equation, the simplest solution is to use another (equivalent) way to write the relationships between the DIC species:

$$[HCO_3^-] = K_{C1} \cdot \frac{[CO_2]}{[H^+]}$$

$$[CO_3^{2-}] = K_{C2} \cdot \frac{[HCO_3^-]}{[H^+]}$$

pCO_2 relates to $[CO_2]$ through Henrys constant, Kh, which can also be estimated as a function of salinity, temperature and pressure, using R-package **seacarb**:

$$pCO_2 = \frac{[CO_2]}{Kh}$$

- Estimate the pH at equilibrium with alkalinity 2300 $\mu mol\ kg^{-1}$ and the current pCO_2 of 360 ppm.

  Use package **seacarb** to estimate the dissociation constants and Henrys constants at temperature 20°C, salinity 0, and pressure 0. (A: pH = 8.19)

- The Intergovernmental Panel on Climate Change predicts for 2100 an atmospheric CO_2 concentration ranging between 490 and 1250 ppmv, depending on the socio-economic scenario (IPCC, 2007). These increases of pCO_2 make the water more acid. Make a plot of pH as a function of these increased atmospheric pCO_2 levels. (Assume that the pCO_2 of the ocean is at equilibrium with the atmospheric pCO_2). What is the maximal drop of pH ? (A: at pCO_2 of 1250 ppmv, pH = 7.68).

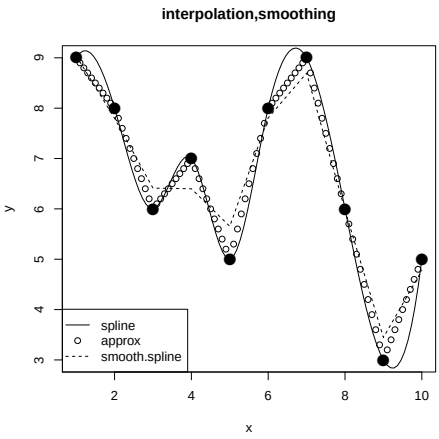


Figure 12: smoothing and interpolation - see text for R-code

10. Interpolation, smoothing and curve fitting

10.1. Interpolation and smoothing

Interpolating and smoothing in R can be done in several ways:

- `approx` linearly interpolates through points
- `spline` uses spline interpolation, which is smoother
- `smooth.spline` smoothens data sets; this means that it does not connect the original points.

The use of these functions is exemplified in the following script and corresponding output:

```
x <- 1 : 10
y <- c(9, 8, 6, 7, 5, 8, 9, 6, 3, 5)
```

```
plot(x, y, pch = 16, cex = 2, main = "interpolation,smoothing")
lines (spline(x, y, n = 100), lty = 1)
points(approx(x, y, xout = seq(from = 1, to = 10, by = 0.1)), pch = 1)
lines (smooth.spline(x, y), lty = 2)
legend("bottomleft", lty = c(1, NA, 2), pch = c(NA, 1, NA),
       legend = c("spline", "approx", "smooth.spline"))
```

10.2. Curve fitting

R also has several curve fitting procedures. Depending on whether the function to be fitted is linear, or non-linear, you may use:

- `lm` and `glm` for fitting linear models and generalised linear models
- `nls`, `nlm`, `optim`, `constrOptim` for nonlinear models.

As an example, we now fit the US population density values, at 10-year intervals, with the logistic growth model (see previous chapter). The model was:

$$N(t) = \frac{K}{1 + [\frac{K-N_{t0}}{N_{t0}}] \cdot e^{-a \cdot (t-t_0)}},$$

and the data:

1900	1910	1920	1930	1940	1950	1960	1970	1980
76.1	92.4	106.5	123.1	132.6	152.3	180.7	204.9	226.5

We start by inputting the data.

```
year <- seq(from = 1900, to = 1980, by = 10)
pop  <- c(76.1, 92.4, 106.5, 123.1, 132.6, 152.3, 180.7, 204.9, 226.5)
```

The simplest method for non-linear curve fitting is by using R function `nls`.

This functions requires as input the formula (`y ~ f (x,parameters)`) and starting values of the parameters.

In the example, `y` are the population values, `f` is the logistic growth formulation.

As starting conditions, we use: $K = 500, N0 = 76.1, a = 0.02$.

```
fit <- nls(pop ~ K/(1+(K-N0)/N0 * exp(-a*(year-1900))),
           start = list(K = 500, N0 = 76.1, a = 0.02))
```

We end the fitting by printing a summary of the fitting parameters, which shows the estimates of the parameters and their standard errors. Clearly, it is not possible to obtain reliable estimates of the value of K based on the data.

```
summary(fit)
```

```
Formula: pop ~ K/(1 + (K - N0)/N0 * exp(-a * (year - 1900)))

Parameters:
    Estimate Std. Error t value Pr(>|t|)
K  1.008e+03  8.932e+02   1.129  0.30210
N0 7.866e+01  2.531e+00  31.084 7.36e-08 ***
a  1.550e-02  2.505e-03   6.188  0.00082 ***
---
Signif. codes:  0 Ś***Š 0.001 Ś**Š 0.01 Ś*Š 0.05 Ś.Š 0.1 Ś Š 1

Residual standard error: 3.685 on 6 degrees of freedom

Number of iterations to convergence: 6
Achieved convergence tolerance: 4.301e-06
```

The values of the coefficients themselves are retrieved using R-function `coef`.

```
coef(fit)
```

```
          K           N0            a
1.008227e+03 7.866365e+01 1.550343e-02
```

10.3. Exercises

Smoothing

An anemometer measures wind-velocity at three hourly intervals. On a certain day, these velocities are: 5, 6, 7, 9, 4, 6, 3, 7, 9 at time 0, 3, ... 24 o'clock respectively. In order to estimate air-sea exchange, we need hourly measures.

Tasks:

- Interpolate the three-hourly measurements to hourly measurements.
- Make a plot of the interpolated values

Fitting

Primary production is measured by ^{14}C incubations from phytoplankton samples, at different light intensities. The data are:

```
ll <- c(0., 1, 10, 20, 40, 80, 120, 160, 300, 480, 700)
pp <- c(0., 1, 3, 4, 6, 8, 10, 11, 10, 9, 8)
```

Fit the resulting production estimates (pp), as a function of light intensity (ll) with the 3-parameter Eilers-Peeters equation. The primary production is calculated as:

$$pp = p\max \cdot \frac{2 \cdot (1+\beta) \cdot {}^{I}/_{Iopt}}{({}^{I}/_{Iopt})^2 + 2 \cdot \beta \cdot {}^{I}/_{Iopt} + 1}$$

where I is light and pmax, β and Iopt are parameters.

- First plot primary production (pp) versus light (ll). Use large symbols.
- Then use R-function `nls` to fit the model to the data
- Add the best-fit line to the graph. (note: use `coef` to retrieve the best parameter values).

11. Differential equations

11.1. Dynamic models

Model formulation in the natural sciences

In the natural sciences, the challenge is often to describe or predict how a given environmental variable will evolve over time. Depending on the problem at hand, this variable can have many different meanings. For example, one could be interested in future evolution of the temperature of the atmosphere (as in the IPCC models of global warming), the seasonal fluctuation in the oxygen concentration in a lake (as described in biogeochemical ecosystem models), or the year-to-year fluctuations in the density of lions in the Serengeti National park (as in the population models used by conservation biologists). The mathematical models that describe such changes in environmental variable are called dynamic models, because they have the time t as the independent variable. The models themselves are based on differential equations, which are typically of the form:

$$\frac{dC}{dt} = f\left(a, b(t), C\right) = f\left(C, t\right)$$

In this, t represents the time and C is the state variable. Furthermore, a is a parameter that remains constant in time, while $b(t)$ is a function that varies with time (a so-called forcing function). This equation is termed a differential equation because it contains the derivative dC/dt, which represents the so-called rate of change of the state variable C. The specific form of the function f depends on the physical, chemical and biological processes that influence C.

The determination of how the function f should look like, is the subject of the process of model development. In essence, one or more suitable differential equations need to be derived from the general balance equations of mass, momentum and/or energy, which form the theoretical pillars of any model in the natural sciences. How this is exactly done will not be considered here, but is treated in detail in the course on Environmental Modelling.

An example: Transformation of an organic pollutant in a lake

For now, we can assume that the model has already been developed, and so, we know how the differential equation(s) will look like. For example, imagine that we want to study how a lake responds to the input of an organic pollutant. This pollutant is brought in by the river, and decays within the lake (for example, under the influence of UV radation). Based on mass balance considerations, we know that the time evolution of the pollutant concentration C can be described by the differential equation

$$V \cdot \frac{dC}{dt} = F(t) \cdot (C_{in} - C) - k \cdot V \cdot C$$

In this, V represents the volume of the lake (m^3), C_{in} is the concentration of the pollutant in the inflowing river water ($mol\ m^{-3}$), and k is the decay rate constant of the pollutant (s^{-1}). The quantity $F(t)$ is the flow rate of the river that discharges into the lake upstream ($m^3\ s^{-1}$). It is made dependent on time as this flow rate strongly varies over the year. After dividing both sides by the volume V, this equation can be brought in the same form as the

differential equation above:

$$\frac{dC}{dt} = D(t) \cdot (C_{in} - C) - k \cdot C$$

where $D = F/V$ is called the dilution rate of the lake. In modelling theory, the following terminology is used:

- C is the dependent variable (or state variable),
- t is the independent variable,
- $\frac{dC}{dt}$ is the rate of change (the derivative of the dependent variable with respect to the independent variable),
- k and C_{in} are parameters (do not vary with time, the independent variable),
- $D(t)$ is a forcing function (varies with time).

The specification of the differential equation by itself is not enough to arrive at a complete model statement. We also need to specify the initial conditions. In our case, this is the pollutant concentration C_0 in the lake at some initial time t_0.

$$C(t = t_0) = C_0$$

The model formulation is now complete. The goal is now to find a suitable expression for the pollutant concentration as a function of time, i.e., $C(t)$. This requires the solution of the above differential equation given the above initial condition.

11.2. Types of differential equations

Autonomous versus non-autonomous differential equations

When the function f does not depend on the time t, the differential equation is called autonomous. In other words, an autonomous differential equation only features constant parameters, and does not contain forcing functions. To illustrate this, consider following two different models of the same lake:

$$\begin{aligned}\frac{dC}{dt} &= D_0 \left(C_{in} - C\right) - k \cdot C \\ \frac{dC}{dt} &= \left(D_0 + D_{fluc} \sin\left(2\pi \tfrac{t}{\tau}\right)\right) \left(C_{in} - C\right) - k \cdot C\end{aligned}$$

The first model describes a lake with a constant inflow, and the associated differential equation becomes autonomous. The second model describes a lake where the inflow varies with a regular seasonal cycle. The parameter D_{fluc} describes the magnitude of the discharge fluctuation, and τ the time period of the cycle. The associated differential equation explicitly features the time t and so it is no longer autonomous.

Linear versus non-linear differential equations

When the function f is a linear function of the variable C, the differential equation is termed linear. Linear dynamic models have the general form:

$$\frac{dC}{dt} = a_1(t) \cdot C + a_0(t)$$

When $a_0(t)$ and $a_1(t)$ are constant (independent of time), the equation is said to be autonomous (as already noted above). When $a_0(t) = 0$, the equation is said to be homogeneous. For example, consider following variants of our lake model, which differ in the consumption term of the pollutant:

$$\begin{aligned}
\frac{dC}{dt} &= D \cdot (C_{in} - C) - k \cdot C \\
\frac{dC}{dt} &= D \cdot (C_{in} - C) - k \cdot C^2 \\
\frac{dC}{dt} &= D \cdot (C_{in} - C) - R \cdot \exp(-r \cdot t)
\end{aligned}$$

The first differential equation is autonomous and linear. The second equation is autonomous and non-linear (because it features the quadratic consumption term $k \cdot C^2$). The third equation is again linear but is no longer autonomous as the consumption term now features the time t (note that the linearity only applies to the dependent variable C and not the independent variable t). The (non)-linearity of a dynamic model depends on the processes that are playing and the level of detail with which these processes are modelled. Linear, autonomous differential equations always have an analytical solution (see below). Non-linear or non-autonomous differential equations sometimes allow an analytical solution, but this is only rarely the case. If not, then a numerical solution must be found.

Ordinary versus partial differential equations

When the function f only contains the time t and the state variable C, the resulting equation is called an ordinary differential equation (ODE). When the function f also contains one or more differentials of the state variable C with respect to a spatial coordinate, the resulting equation is called a partial differential equation (PDE).

The use of ODE's versus PDE's to describe a natural system depends on the level of detail one wants to include in the model. For example, the concentration in a lake can be described with varying spatial detail. If the lake is strongly mixed, concentrations will not exhibit spatial gradients, and so the lake can be considered as one homogeneously mixed volume. In this case, the lake can be described by a single concentration $C(t)$, which only depends on time. However, when the lake is not fully mixed, more complex descriptions of the concentration in the lake are needed. Following cases are possible:

- 0D-model: $C(t)$ only dependent on time (ODE case),
- 1D-model: $C(t, z)$ depends on time and the spatial coordinate (e.g. z = depth in a lake),
- 2D-model: $C(t, x, z)$ depends on time and two spatial coordinates (e.g. z = depth, x = distance along the axis of a river)
- 3D-model: $C(t, x, y, z)$ depends on time and three spatial coordinates (e.g. full three-dimensional model of a lake)

The following equations describe the exactly same lake, but with a different level of detail:

$$\frac{dC}{dt} = D \cdot (C_{in} - C) - k \cdot C$$
$$\frac{dC}{dt} = K_z \frac{d^2C}{dz^2} - k \cdot C$$
$$\frac{dC}{dt} = K_x \frac{d^2C}{dx^2} + K_y \frac{d^2C}{dy^2} + K_z \frac{d^2C}{dz^2} - k \cdot C$$

The first equation is an ODE or ordinary differential equation and describes how the average concentration in the lake will change through time ($0D$-model). The second equation comprises a PDE or partial differential equation, and describes how the concentration will change with time as well as with the depth of the lake ($1D$-model). The third equation is also a PDE, but now describes the concentration varies in all three directions (full $3D$-model).

Note that for the solution of a PDE problem, one needs a set of boundary conditions in addition to the initial conditions. Here, we only focus on the solution of ODE problems. Accordingly, the implementation of boundary conditions and the solution of PDEs is for a more advanced course. Intrinsically however, it is similar to the solution of ODEs.

11.3. Time-dependent solution: Analytical approach

The solution of a dynamic model comes down to the determination of how the state variable C will evolve as a function of the time t. As already noted above, the solution of a dynamic model comes down to the solution of a given differential equation combined with some suitable initial conditions. The time-dependent solution of a differential solution is also termed the transient solution.

When the differential equation is relatively simple (such as in the case of a linear ODE), one can find a so-called analytical solution. This means that an explicit mathematical expression is available for the state variable $C(t)$. The theory of differential calculus provides a whole series of methods to find such analytical solutions (this is beyond the scope of our course here). So for our purposes here, an analytical solution is one that can be found in mathematical textbooks. Let's consider again pour lake model,

$$\frac{dC}{dt} = D \cdot (C_{in} - C) - k \cdot C$$
$$C(t = t_0) = C_0$$

This is a linear ODE with constant parameters, and for this specific type of ODEs, one can prove that an analytical solution must always exist. The specific analytical solution for the above problem is:

$$C(t) = \frac{D}{D+k} C_{in} + \left(C_0 - \frac{D}{D+k} C_{in} \right) \cdot \exp\left(-(D+k) \cdot t\right)$$

One can indeed verify that this expression is a true solution for the above model (to this end, substitute the analytical solution into the differential equation and the initial condition). To see how the concentration will evolve over time, one simply needs to plot the analytical solution in a graph.

```
# Parameters
D<-2; C_in<-1; k<-1; C_0<-2
# Time sequence vector
```

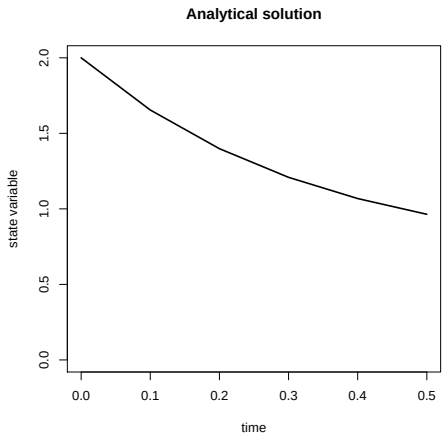


Figure 13: Plot of analytical solution of an ODE

```
time.seq <- seq(from = 0, to = 0.5, by = 0.1)
# Analytical solution as a vector expression
C.star <- D/(D+k)*C_in
C.an <- C.star + (C_0 - C.star )*exp(-(D+k)*time.seq)
# Plot solution
plot(time.seq, C.an, type="l", ylim=c(0,2), lwd =2, cex = 2,
xlab="time", ylab="state variable", main = "Analytical solution")
```

11.4. Time-dependent solution: Numerical approach

It is often difficult to find the analytic, or exact, solution to a differential equation. This may be because the equation is non-linear or because it has parameters that vary with time. Moreover, even for linear ODEs with constant parameters it can be useful to have a numerical solution because the solution can be automated for many different inputs and/or initial conditions. There are many methods for finding a numerical solution to differential equations. These methods are referred to by a variety of different names: numerical methods, numerical integration, approximate solution. In this section, we will introduce a basic method (the Euler method). Afterwards, we will show how to use more sophisticated integration methods in R .

A first point to note is that numerical methods do not generate exact solutions, but only approximate ones. Because these methods are based on computation, the output of a numerical method approximates the solution to the differential equations (and they may provide an extremely poor approximation when not used carefully). A second important point is that numerical metods only calculate the solution at certain discrete time intervals. Typically, the solution is calculated at the time t_0 of the initial conditions, and subsequently, for every time interval Δt thereafter (i.e., at $t = t_0 + \Delta t$, $t = t_0 + 2\Delta t$, ... , $t = t_0 + n\Delta t$).

Euler integration

The simplest method of numerical integration is Euler's method, which is presented now.

Consider the generic dynamic model:

$$\frac{dC}{dt} = f\,(C,t)$$
$$C(t=t_0) = C_0$$

Starting at the initial time t_0, we would like to calculate a new value for the state variable C after the time interval Δt. To this end, we can approximate the differential of the concentration by its linear difference

$$\frac{dC}{dt} = \lim_{t\to 0}\left(\frac{C(t+\Delta t)-C(t)}{\Delta t}\right) \approx \frac{C(t+\Delta t)-C(t)}{\Delta t}$$

If we re-arrange this formula, we can calculate an new approximate value C^* at the time $t = t_0 + \Delta t$ as a function of what we know at the time $t = t_0$

$$C^*\,(t_0+\Delta t) = C(t_0) + \Delta t \cdot \left(\frac{dC}{dt}\right)_{t=t_0}$$

Furthermore, as we now from the ODE in the model formulation, the derivative of C with respect to the time t at $t = t_0$ is given by the function value of f at $t = t_0$

$$C^*\,(t_0+\Delta t) = C(t_0) + \Delta t \cdot f\,(C_0, t_0)$$

This immediately suggests a sequential procedure for calculating the numerical solution of an ODE. Using the previously computed approximation, we can get the approximation at the next time step. The following sequential scheme illustrates this procedure:

$$C_0^* = C^*(t_0) = C_0$$
$$C_1^* = C^*\,(t_0+\Delta t) = C_0^* + \Delta t \cdot f\,(C_0^*, t_0)$$
$$C_2^* = C^*\,(t_0+2\Delta t) = C_1^* + \Delta t \cdot f\,(C_1^*, t_0+\Delta t)$$
$$...$$

This way, we obtain a sequence of numbers C_1^*, C_2^*, C_3^*, ..., C_n^* that will approximate the solution of the differential equation at t_1 , t_2 , t_3, ..., t_n. This sequential procedure is the basis of the Euler integration method.

So, how do we implement the Euler Method in an R script? It is fairly simple. We first define the integrator function f, which specifies the right-hand side of the ODEs. This function is called `model` in the R script, and has a quite elaborate form. We introduce this specific form here for later consistency with other integrator methods (but admittedly, for the Euler method, it is a bit overcomplicated). The `model` function has three different arguments as input: the actual time (`t`), the values of the state variables (`state`) and the values of the parameters (`parameters`).

```
model <- function(t, state, parameters)
{
with (as.list(c(state, parameters)),
 {
 dC <- D*(C_in-C)-k*C
 return (list(c(dC)))
```

```
 })
}
```

Subsequently we must specify the parameters and the initial conditions.

```
# Parameters
parameters <- c(D=2,C_in=1,k=1)
# Initial conditions
C_0 <- 2
```

As noted above, any numerical integration procedure calculates the solution only at specific points in time. Therefore, we need to explicitly specify a vector that contains the different times where an approximate solution should be calculated.

```
# Step size and number of steps
time.seq <- seq(from = 0, to = 0.5, by = 0.1)
n <- length(time.seq)
```

Finally, the sequential time-stepping of the Euler procedure can be implemented as

```
C  <- vector(length = length(time.seq))
dC_dt  <- vector(length = length(time.seq))
C[1] <- C_0

for (i in 1:(n-1))
{
  # Model function call: Calculate the rate of change at the present time point
  dC_dt[i] <- model(t=time.seq[i],state=c(C=C[i]),parameters)[[1]]

  # Calculate the time jump
  Delta_t <- time.seq[i+1] - time.seq[i]

  # Integration step: Calculate the concentration at next time point
  C[i+1] <- C[i] + Delta_t*dC_dt[i]

}
```

As you can see on the graph, the numerical solution slightly deviates from the exact analytical solution. This is because we have used a rather large time interval $\Delta t = 0.2$. If we would make this time interval smaller, the numerical approximation would improve (see exercises).

11.5. Numerical solution: implementation

The Euler method only provides a very crude and basic approach to numerical integration. There are more sophisticated integration methods available, which are contained within the

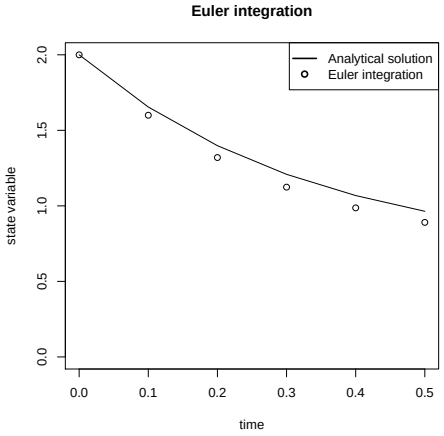


Figure 14: Comparison of numerical solution via Euler method (points) with the analytical solutions (line)

deSolve package (differential equations Solving). Here, we will illustrate the general procedure for the time-dependent numerical solution of ODEs in R . As an example, consider the following set of two differential equations:

$$\frac{dA}{dt} = r \cdot (x - A) - k \cdot A \cdot B$$
$$\frac{dB}{dt} = r \cdot (y - B) + k \cdot A \cdot B$$

In this, A and B are the state variables, while r, x, y and k are constant parameters.

The first step is to reformulate the model so that the right hand sides only features the differentials (this is already done above). Then, we can define the integrator function (again called `model`), which specifies the right-hand side of the ODEs. As noted before, this function has three different arguments as input: the actual time (`t`), the values of the state variables (`state`) and the values of the parameters (`parameters`).

```
model <- function(t, state, parameters)
{
with (as.list(c(state, parameters)),
{
 dA <- r*(x-A) - k*A*B
 dB <- r*(y-B) + k*A*B
 return (list(c(dA, dB)))
 })
}
```

The above function can be used as a blueprint. This function simply calculates the rate of change of the state variables (`dA` and `dB`) and returns those as a list. The R-statement `with (as.list (c(state,parameters)),` ensures that the state variables and parameters can be addressed by their names.

Before we can actually solve this ODE model, we need to:

- give values to the parameters (`parameters`):
- assign initial conditions to the state variables (`state`)
- generate a sequence of time values at which we want output (`time.seq`),

```
parameters <- c(x = 1, y = 0.1, k = 0.05, r = 0.05)
state <- c(A = 1, B = 1)
time.seq <- seq(from = 0, to = 300, by = 1)
```

The model can now be solved. To do so, we use the integration routine `ode`, which can be found in R package **deSolve**. This package is loaded first.

```
require(deSolve)
```

Just like the Euler method did above, the routine `ode` will calculate an approximate value for the state variables `A` and `B` at each time value specified in the vector `time.seq`. Accordingly, the actual numerical solution of our ODE model is doen within the following single statement:

```
out <- ode(y=state,times=time.seq,func=model,parms=parameters)
```

The output is stored in a matrix, called `out`. All we need to do now is to plot this model output. Before we do so, we can have a look at the output matrix `out`:

```
 head(out)
```

```
     time         A         B
[1,]    0 1.0000000 1.0000000
[2,]    1 0.9523189 1.0037869
[3,]    2 0.9090687 1.0052854
[4,]    3 0.8699226 1.0047151
[5,]    4 0.8345728 1.0022854
[6,]    5 0.8027203 0.9982009
```

The data are arranged in three columns: first the time stamp, then values of the state variables A and B. We can extract the data using the column names (`out[,"time"]`, `out[,"A"]`, `out[,"B"]`).

Before plotting the model output, the range of concentrations of substances A and B is estimated. This is used to set the limits of the y-axis (`ylim`). The R -function `plot` creates a new plot; `lines` adds a line to this plot; `lty` selects a line type; `lwd=2` makes the lines twice as thick as the default. Finally a `legend` is added.

```
ylim   <- range(c(out[,"A"],out[,"A"]))
plot(out, which = "A", xlab = "time", ylab = "concentration",
      lwd = 2, type = "l", ylim = ylim, main = "model")
lines(out[,"time"], out[,"B"], lwd = 2, lty = 2)
legend("topright", legend = c("A", "B"),lwd = 2, lty = c(1, 2))
```

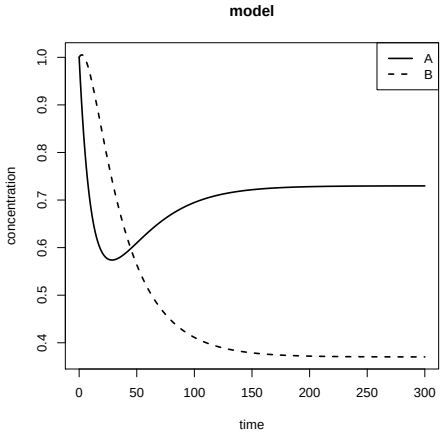


Figure 15: ode model - see text for R-code

11.6. Steady-state model solution

Of particular interest is the behaviour of dynamic models for very long times. There are four possibilities for the long term behaviour of a dynamic model:

- The model eventually reaches a state where the state variables no longer change with time. This is called a steady state.
- The model solution goes to infinity. It is said that the model solution diverges.
- The model solution eventually reaches a periodic regime, where the solution repeats itself after a certain period.
- The model solution does not diverge, but also never tends to a periodic or staedy regime. It keeps on going in a rather unpredictable fashion. This is called a chaotic solution.

Here, we will be mainly concerned with systems that reach a steady state. There are three alternative ways to find this steady state.

Direct analytical calculation of steady-state

In the direct procedure, one takes advantage of the definition of the steady state. According to its definition, the steady state is a particular state where the rate of change of all state variables vanishes. As a result, we can set the dC/dt terms on the left hand side of the differential equation to zero:

$$\frac{dC}{dt} = 0 = D \cdot (C_{in} - C) - k \cdot C$$

Because, we have removed the dC/dt terms, we arrive at an algebraic equation on the right hand side (this equation no longer contains any differentials). By solving this algenraic equation, we find that the steady state concentration is given by:

$$C^* = \frac{D}{D+k} \cdot C_{in}$$

One can verify in Figure (**??**) that this is indeed the long-term behaviour of the model. When the time becomes large, the solution asymtotically approaches C_{in}.

Direct numerical calculation of steady-state

This is exactly the same procedure as above. One sets the dC/dt terms to zero to arrive at an algebraic equation. However, sometimes this algebraic equation does not allow the analytical calculation of the root. In this case, one can use a numerical root-solving procedure to approximately calculate the staedy state value to within some level of tolerance. See Chapter 9 on root finding.

Time dependent numerical calculation of steady state

Here we use another aspect of the steady state. As noted above, the model will eventually reach the steady state when the model runs long enough. Accordingly, we can apply the time-dependent numerical solution procedure as discussed above, and let it run for a long time interval. Subsequently, we can check if the rate of change terms dC/dt are small enough. If so, we have reached the steady state. If not, we need to prolong the integration interval, and repeat the procedure (run the model and check for steady state).

11.7. Exercises

Exercise 01: Numerical integration

Find the numerical solution of following dynamic problems by adapting the blueprint script for numerical integration with `ode` function.

Dynamic problem 1a:

$$\begin{aligned} &\frac{dC}{dt} = -b \cdot C + a \\ &C(t = t_0) = 0 \end{aligned}$$

Parameter values are $a = 1$ and $b = 2$. The integration period is 50 time units with output every time unit.

Dynamic problem 1b:

$$\begin{aligned} &\frac{dC}{dt} = f \cdot \exp(-d \cdot t) \cdot C^2 - b \cdot \sqrt{C} + a \\ &C(t = t_0) = 0 \end{aligned}$$

Parameter values are $a = 1$, $b = 0.2$, $d = 0.1$ and $f = 0.01$. The integration period runs over 10 time units with output every 0.1 time unit. Afterwards perform a new simulation where the intergration period is increased to 50 days (store the results in a new object called `out2`). Plot both solutions side by side in a two panel plot.

Are the above problems linear, autonomous and/or homogeneous?

Exercise 02: Oxygen dynamics in a small pond

In a small pond, the oxygen concentration fluctuates daily by means of production during the day (as a result of photosynthesis) and consumption at night (due to respiration). The daily variation of the net oxygen production rate R (photosynthesis minus respiration) is given by:

$$R = R_0 \cdot \sin\left(2\pi \frac{t}{\tau}\right)$$

The parameter R_0 is 100 $mmol\ m^{-3}\ d^{-1}$ and the fluctuation period τ is 1 day. The dynamics of the oxygen in the pond is further described by the following model:

$$\frac{dC}{dt} = \frac{k_d}{L} \cdot (C_{atm} - C) + R$$
$$C(t = t_0) = C_0$$

The parameter k_d is the piston velocity (5 $m\ d^{-1}$) which regulates the exchange with the atmosphere, and L is the depth of the pond (1 m). The parameter C_{atm} represents the oxygen concentration that is in equilibrium with the atmosphere (250 $mmol\ m^{-3}$). The initial concentration of oxygen is 100 $mmol\ m^{-3}$ The integration period runs over 10 days with output every 0.1 day.

Find the numerical solution and make a plot (black line). Does the oxygen concentration reach a steady state? Determine a second model solution with no net O2 production $R_0 = 0$, all other parameters being the same. Plot this solution in a blue line on the same graph.

Exercise 03: Logistic growth model

A realistic model of population growth is given by the logistic growth equation.

$$\frac{dP}{dt} = r\left(1 - \frac{P}{K}\right) P$$
$$P(t = t_0) = P_0$$

In this, P_0 is the initial population, r is the intrinsic growth rate (growth rate that will occur in the absence of any limiting factors), and K is called the carrying capacity.

- Write a script file that solves the model numerically using the `ode` function. The parameter values are $r = 0.5$ and $K = 10$, while teh initial population size is $P_0 = 2$. Simulate the model over a time period of 10 time units.
- Does the model reach a steady state over this period? Check this by running the model over a longer time period (e.g. 100 time units). What is the population size P after such a long time?
- Determine the steady state solution using the direct analytical calculation of steady-state. Is this the same as the population size P after 100 time units?

Exercise 04: Steady state calculations

Determine the steady state solutions of following ODEs.

$$\frac{dC}{dt} = 2 \cdot (6 - C)$$
$$\frac{dC}{dt} = 4 - 2 \cdot \exp(C)$$
$$\frac{dC}{dt} = 7 - 2 \cdot \ln(2C + 0.1)$$

Do this via the method of direct analytical calculation as well as the method of time dependent numerical calculation. Check that both solutions are identical.

Exercise 05: Euler method Consider the following dynamic problem:

$$\frac{dC}{dt} = 2 - 2 \cdot C - exp(-4t)$$
$$C(t = t_0) = 1$$

This is a fairly simple linear differential equation, for which the analytical solution is given by:

$$C(t) = 1 + 0.5exp(-4t) - 0.5exp(-2t)$$

Use the Euler Method with a step size of $\Delta t = 0.1$ to find approximate values of the solution at t = 0.1, 0.2, 0.3, 0.4, and 0.5. Compare them to the exact values of the analytical solution at these points. Quantify the error between numerical and exact solution as:

$$\epsilon = 100 \left(\frac{C_{num} - C_{an}}{C_{an}} \right)$$

The maximum error in the approximations from this example is around 5%. This is not too bad, but also not all the great of an approximation. This kind of error is generally unacceptable in real applications however. So, how can we get better approximations? Try the same Euler method, though now with a reduced time intervals of $\Delta t = 0.05$ and $\Delta t = 0.01$. Calculate once more the relative error and summarize all information in one matrix table (time, exact, Dt=0.1, Dt=0.05, Dt=0.01).

Exercise 06: Lotka-volterra model The Lotka-Volterra model is a famous model that either describes predator-prey interactions or competitive interactions between two species. A.J. Lotka and V. Volterra formulated the model almost simultaneously in the 1920's ((Lotka 1925), (Volterra 1926)).

- Write a script file that solves the Lotka-Volterra model:

$$\frac{dx}{dt} = a \cdot x \cdot (1 - \frac{x}{K}) - b \cdot x \cdot y$$
$$\frac{dy}{dt} = g \cdot b \cdot x \cdot y - e \cdot y$$

  for initial values x=300,y=10 and parameter values: a=0.05, K=500, b=0.0002, g=0.8, e=0.03

- Make three plots, one for x and one for y as a function of time, and one plot expressing y as a function of x (this is called a phase-plane plot). Arrange these plots in 2 rows and 2 columns.

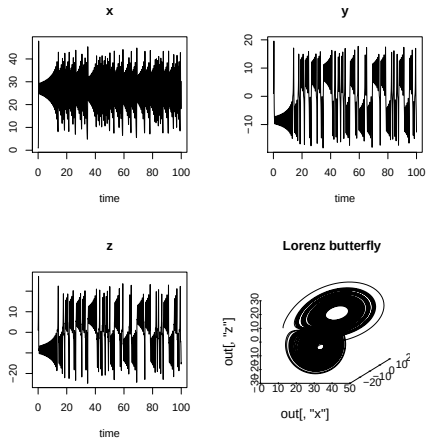
- Now run the model with other initial values (x=200, y=50); add the (x,y) trajectories to the phase-plane plot

Exercise 07: Butterfly The Lorenz equations (Lorenz 1963) represents the fisrt set of of differential equations in which chaotic behaviour was discovered. These three differential equations represent an idealized model for the circulation of air within the atmosphere of the earth.

$$\frac{dx}{dt} = -\frac{8}{3} \cdot x + y \cdot z$$
$$\frac{dy}{dt} = -10 \cdot (y - z)$$
$$\frac{dz}{dt} = -x \cdot y + 28y - z$$

- It takes about 10 lines of R-code to generate the solutions and plot them.

- Function `scatterplot3d` from the package **scatterplot3d** generates 3-D scatterplots. Can you recreate the following "butterfly" ? Use as initial conditions x = y = z = 1; create output for a time sequence ranging from 0 to 100, and with a time step of 0.005.


x
y
z
Lorenz butterfly
time
time
time
out[, "z"]
out[, "x"]

12. Final remarks

12.1. The questions

These lecture notes have been generated with LaTeX and making use of R package **Sweave** (Leisch 2002), which allows to merge LaTeX with R-code.

12.2. The answers

The answers to the questions in this course are present as an R-vignette in package **marelacTeaching**. From within R , type:

```
> vignette("Answers")
```

Or, you can find the file "Answers.pdf" in the /inst/doc subdirectory of package **marelacTeaching**.

12.3. If you use tinn-R and it does not automatically connect with R

You connect Tinn-R to the R console by navigating to `Options`, `Main` and `Application`. Go to the `R` tab, and click the `Path to your preferred Rgui` button. Locate the `Rgui.exe` file in the `bin` directory of your specific R installation.

If you have installed the latest tinn-R version, you need to change an input file to make tinn-R communicate properly with R . Find the place where R has been installed (in the `etc` subdirectory) and locate the file "Rprofile.site". Open this file in the tinn-R editor and add the following:

```
.trPaths <- paste(paste(Sys.getenv('APPDATA'),'\\Tinn-R\\tmp\\', sep=''),
c('', 'search.txt', 'objects.txt', 'file.r', 'selection.r', 'block.r', 'lines.r'), sep='')
```

References

Caswell H (2001). *Matrix population models: construction, analysis, and interpretation*. Sinauer, Sunderland, second edition edition.

Hansen P, Bjornsen P, Hansen B (1997). "Zooplankton grazing and growth: Scaling within the 2-2,000-mu m body size range." *Limnology and Oceanograpy*, **42**, 687–704.

Hofmann AF, Soetaert K, Middelburg JJ, Meysman FJR (2010). "AquaEnv - An Aquatic Acid-Base Modelling Environment in R." *Aquatic Geochemistry*, **DOI 10.1007/s10498-009-9084-1**.

Kuhnert P, Venables W (2005). *An introduction to R: software for statistical modelling & computing*. URL `www.r-project.org`.

Lavigne H, Gattuso JP, Epitalon JM, Gentili B, Hofmann A, Orr J, Proye A, Soetaert K (2010). *seacarb: Calculates parameters of the seawater carbonate system*. R package version 2.3.2, URL http://CRAN.R-project.org/package=seacarb.

Leisch F (2002). "Sweave: Dynamic Generation of Statistical Reports Using Literate Data Analysis." In W Härdle, B Rönz (eds.), "Compstat 2002 - Proceedings in Computational Statistics," pp. 575–580. Physica Verlag, Heidelberg. ISBN 3-7908-1517-9, URL http://www.stat.uni-muenchen.de/~leisch/Sweave.

Ligges U, Machler M (2003). "Scatterplot3d - an R Package for Visualizing Multivariate Data." *Journal of Statistical Software*, **8(11)**, 1–20.

Lorenz E (1963). "Deterministic non-periodic flows." *J. Atmos. Sci*, **20**, 130–141.

Lotka AJ (1925). *Elements of Physical Biology*. Williams & Wilkins Co., Baltimore.

Millero F, Poisson A (1981). "International one-atmosphere equation of state for seawater." *Deep-Sea Research*, **28(6)**, 625–629.

R Development Core Team (2011). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org.

Soetaert K (2009). *rootSolve: Nonlinear root finding, equilibrium and steady-state analysis of ordinary differential equations*. R package version 1.4.

Soetaert K, Heip C, Vincx M (1991). "Diversity of nematode assemblages along a mediterranean deep-sea transect." *Marine Ecology Progress Series*, **75**, 275–282.

Soetaert K, Herman PMJ (2009). *A Practical Guide to Ecological Modelling. Using R as a Simulation Platform*. Springer. ISBN 978-1-4020-8623-6.

Soetaert K, Meysman F (2009). *marelacTeaching: Datasets and tutorials for use in the MArine, Riverine, Estuarine, LAcustrine and Coastal sciences*. R package version 1.1.

Soetaert K, Petzoldt T, Meysman F (2009). *marelac: Constants, conversion factors, utilities for the MArine, Riverine, Estuarine, LAcustrine and Coastal sciences*. R package version 2.0.

Soetaert K, Petzoldt T, Setzer RW (2010). "Solving Differential Equations in R: Package deSolve." *Journal of Statistical Software*, **33**(9), 1–25. ISSN 1548-7660. URL http://www.jstatsoft.org/v33/i09.

Verhulst PF (1838). "Notice sur la loi que la population pursuit dans son accroissement." *Correspondance mathematique et physique*, **10**, 113–121.

Volterra V (1926). "Variazioni e fluttuazioni del numero d'individui in specie animali conviventi." *Mem. R. Accad. Naz. dei Lincei. Ser. VI*, **2**, 31–113.

Zeebe R, Wolf-Gladrow D (2003). *CO2 in Seawater: Equilibrium, kinetics, isotopes*. Elsevier.

Affiliation:

Karline Soetaert
Royal Netherlands Institute of Sea Research (NIOZ)
4401 NT Yerseke, Netherlands E-mail: karline.soetaert@nioz.nl
URL: http://www.nioz.nl

Filip Meysman
Royal Netherlands Institute of Sea Research (NIOZ)
4401 NT Yerseke, Netherlands
E-mail: filip.meysman@nioz.nl