A Beginners Guide to R's Galaxy

Michal J. Czyz

2018-05-18

Contents

Chapter 1

In the beginning there was only darknes...

R (R Core Team, 2017) is one of the most common used languages in Data Science. It is so called fourth-generation programming language (4GL), meaning it is *user-friendly*, while still quite powerful. **R** is powered by huge open-source oriented community. Thanks to their work, during many years of development, enormous number of *packages* (also incorrectly called *libraries*) were established, making using **R** for common works related to Data Science easy even for Beginners.

The purpose of this document is to familiarize with **R** people who have at least some basics in statistics or modelling and no knowledge on programming. Thus, examples you will find in this book are driven by making life easier for all of those who struggle with data in their work.

To give you an example and how awesome and powerful **R** is, I wrote whole this book in **R** using package **bookdown** (Xie, 2016, 2018). Hoping this short description encouraged you to dive into *World of R*, we can start learning opportunities of this programming language.

Chapter 2

Introduction

2.1 Audience

I wrote this small book for all those who are new to programming, as well for those who are new to **R** language. I tried to cover all the necessary knowledge you need to be familiarize with to start basic work in **R** in a compact way. I also tried to avoid things that may overwhelm or confuse people who are new to programming (like code efficiency). However if you want to play more, with more complex coding you need to read a lot from other sources. You will find some advises what should you do next in last chapter (9) of this book.

2.2 What this book is not about?

Mainly this book not on any advanced R. I do not cover here things like *function factories*, *creating packages* or *S3* classes. It also do not cover specific **R** application in detail, like *statistics*, *predictive modeling* or *text analyses*. Finally this book will not cover programming paradigms such as *Test Driven Development* or *Meta-programming*. There are plenty of well written resources that go really deep in those applications.

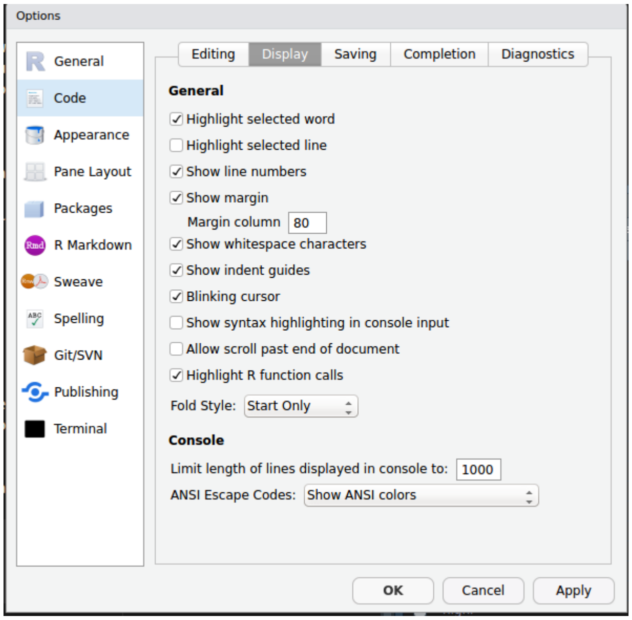
2.3 RStudio

Before we jump into coding, you should first get familiar with **RStudio** (RStudio Team, 2016). It is so called *Integrated Development Environment* (IDE), which has built-in functionalities to make work easier. This IDE is typically used with 4 different windows:

- *Source* - where you can write scripts;
- *Console* - where scripts are executed;
- *'Environmental'* - it's adjustable window, usually containing *Environemnt*, *History* and Version Control panes;
- *'Files'* - also adjustable, usually you will find here *File*, *Packages*, *Help* and *Plots* panes.

2.4 Few tips to make life easier

From menu choose *Tools > Global options*. Now choose *Code* and *Editing* pane, tick box *Insert spaces for Tab* and assure that Tab width is set to 2. Next, in *Display* pane, check following tick-boxes:



General

☑ Highlight selected word
□ Highlight selected line
☑ Show line numbers
☑ Show margin
Margin column 80
☑ Show whitespace characters
☑ Show indent guides
☑ Blinking cursor
□ Show syntax highlighting in console input
□ Allow scroll past end of document
☑ Highlight R function calls
Fold Style: Start Only

Console

Limit length of lines displayed in console to: 1000
ANSI Escape Codes: Show ANSI colors

Figure 2.1: Code display options

- *Highlight selected word*
- *Show line number*
- *Show margin* (and set margin column to 80)
- *Show whitespace characters*
- *Highlite R function calls*

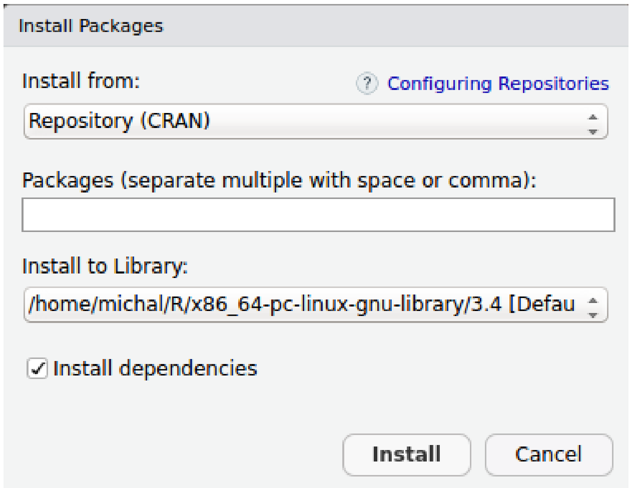
Generally speaking those options, do not influence how your code is performed, but will allow you to write cleaner and read easier. You can also change colors of your environment in *Appearance*.

2.5 Installing packages

In your *'Files'* window, you will find *Packages* pane, which contains *Install* button. You can use it now, to install packages needed to perform exercises from this book. The packages are:

- **deSolve** (Soetaert et al., 2018)
- **fitdistrplus** (Delignette-Muller et al., 2017)
- **mc2d** (Pouillot, 2017)
- **minpack.lm** (Elzhov et al., 2016)
- **nls2** (Grothendieck, 2013)
- **nlsMicrobio** (Baty and Delignette-Muller, 2014)
- **segmented** (Muggeo, 2017)
- **tidyverse** (Wickham, 2017; Wickham et al., 2017; Wickham and Chang, 2016; Wickham and Henry, 2018)

Now every time you need functions from specific package in library you can just tick box next to package name, and RStudio will load it for you.



Install Packages

Install from: Repository (CRAN)

Packages (separate multiple with space or comma):

Install to Library: /home/michal/R/x86_64-pc-linux-gnu-library/3.4 [Defau

☑ Install dependencies

Figure 2.2: Installation window

2.6 Conventions

In this book, we will use following conventions:

- Names of programs and packages are in **Bold**.
- All other names e.g. names of panes menu items as well as things that needs to be stressed are in *italics*.
- Function names and variables are always written in inline code e.g. `t.test()` or `x`.
- File names are written in inline code e.g. `foo.txt`.
- Citations are in APA style, and 'clickable' e.g.click on the name and year of **knitr** package citation (Xie, 2015).
- Code chunks are in blocks and result lines start with ##

```
rnorm(10, 1, 0.5)
```

```
##  [1] 0.9288354 1.1868341 0.9736244 1.4021389 0.7655456 1.4934288 1.7738235
##  [8] 1.6612140 1.7946885 0.4778589
```

- There are no `>` (*prompt*) signs in code chunks.
- Figures are floating - meaning, that they are not always immediately after they are mentioned in text.
- Tables are in *longtable* format (meaning they are not floating and might be multipage) e.g.

```
knitr::kable(
  head(iris, 25), caption = 'Example table',
  booktabs = TRUE, longtable = TRUE
)
```

Table 2.1: Example table

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.0	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa

4.6	3.4	1.4	0.3	setosa
5.0	3.4	1.5	0.2	setosa
4.4	2.9	1.4	0.2	setosa
4.9	3.1	1.5	0.1	setosa
5.4	3.7	1.5	0.2	setosa
4.8	3.4	1.6	0.2	setosa
4.8	3.0	1.4	0.1	setosa
4.3	3.0	1.1	0.1	setosa
5.8	4.0	1.2	0.2	setosa
5.7	4.4	1.5	0.4	setosa
5.4	3.9	1.3	0.4	setosa
5.1	3.5	1.4	0.3	setosa
5.7	3.8	1.7	0.3	setosa
5.1	3.8	1.5	0.3	setosa
5.4	3.4	1.7	0.2	setosa
5.1	3.7	1.5	0.4	setosa
4.6	3.6	1.0	0.2	setosa
5.1	3.3	1.7	0.5	setosa
4.8	3.4	1.9	0.2	setosa

- Tables and figures and references are clickable e.g. see Table 2.1 or see Figure 2.1.

Chapter 3

Basics

In this chapter you will see some short introduction to **R** environment. The basic information will be covered with more concern in following chapters.

3.1 Getting started

3.1.1 Help

There are just few things you really need to remember and follow when you want to start using **R**. First, there is very good *help* build in. To access it, you use `?` sign with name of function: e.g.: `?t.test`. After executing command, in your window with *Help* pane, a page dedicated to this function will pop up. You will get information on syntax, options to use with this function and in most cases some code examples. However, with single question mark you are telling **R** to only look into functions from packages that are *currently loaded* and have this *precise name*. If you want to tell **R** to look for proper function in *all packages* or you are not sure what the exact name is you can use double question mark e.g. `??mutate`. In effect, in the same pane and window as previously you will get a list of results that match your query. Finally, using *Packages* pane, you can click on one of the packages names, to display all of the functions within it. Than, by clicking on the name of functions you are interested in, you will be taken to proper page with description.

3.1.2 Internet is a great source of information

Anytime you feel lost or need help that is beyond the scope of manuals, just ask Google. For instance you can use this query: *how to make density plot in R.* Thanks to huge community you will find a lot of answers. The most reliable ones can be found on *StackOverflow*, *StatsExchange* and *RBloggers*. If you don't know if there is a package to perform particular task, also ask uncle Google. For instance, if you want to use random numbers from Dirichlet distribution, you can use this query: *dirichlet distribution r.*

3.1.3 More on internet sources

A good practice, when you want to learn programming language, is to read what other people do and how they code. In the beginning it might be a bit overwhelming or confusing to read all the stuff. However, reading others work will get you used to syntax and workflow, and will give you great basics to invent your own code. Hopefully, you don't need to spend hours searching for some interesting blogs. There is great blog aggregator R weekly, that gathers in one place the best posts, pod-casts, etc. on **R**, every week.

3.2 Syntax

3.2.1 Common operators

In general **R** syntax resembles syntax of standard math functions $f(x, z) = a*x+b*z$. So in **R** the syntax to call this function would be... `f(x, z)`, which would evaluate (behind scenes) expression `a * x + b * z`. You can think about functions in this language with general pattern `function_name(argument_1, argument_2, ..., argument_n)`. Next thing you should know before you start working is that there are three main *signs* used in **R's** syntax. First two are assignment symbols: `<-` and `=`; for convention we use them in different cases. **You need to use = ONLY for function arguments**. Third one is `#`. It is a symbol used for comments. Everything following this symbol to the end of code line will not be executed. There are also other signs (or symbols) which are building blocks of language, however their use is very precisely defined and reserved for certain events. Below you find a table with reference for most common operators used. You will faster grasp it while you write your own code, than by reading about it. Thus, I suggest we go deeper into variable types in **R** language.

Table 3.1: Common operators in R

sign	type	action
+	maths	addition
-	maths	substraction
*	maths	multiplication
/	maths	division
%%	maths	modulo
^	maths	power
>	relations	left greater
>=	relations	left greater of equal
<	relations	right greater
<=	relations	right greater or equal
==	relations	left equal right
!=	relations	left uneqal right
!	logics	not
&	logics	and
|	logics	or
~	model	left relates to right
<- or ->	assignment	assignes value to variable
$	address	extracts values with 'element name' from variable
:	sequence	creates sequence of numbers from 'left value' to 'right value'
%>% or %<>%	piping	pipes results(from left) as arguments to functionon right

3.2.2 Variables

Concept of variable is crucial for programming. In **R** variables can contain many things: vectors, data frames, results of statistical analyses etc. Each of variables have some characteristic properties. They are defined by *class* of the variable. Thanks to *class* attribute, **R** knows, how to deal with variable – what is the internal structure and what operations can be performed over variable. Data can be stored in variables in different manners. To assign something to variable we use `<-` operator, which tells **R** to store right side of arrow under name on the left side of arrow. The simplest variable is *vector*, which can be of *class*: *character*, *integer*, *numeric* or *logical*. For instance:

```
characterVector <- c('a', 'b', 'c')
class(characterVector)
```

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

```
integerVector <- c(1L, 2L, 3L)
class(integerVector)
```

```
## [1] "integer"
```

```
numericVector <- c(2.5, 3.5, 4.5)
class(numericVector)
```

```
## [1] "numeric"
```

```
logicalVector <- c(TRUE, FALSE)
class(logicalVector)
```

```
## [1] "logical"
```

For more complex data, we have four basic classes: *vectors*, *lists*, *data frames* and *matrices*. *Matrix* is similar to *data frame*. The most obvious difference is that *matrix* contains only one *class* of variables (usually *numeric* or *integer*), while *data frame* can store *numeric*, as well as *characters* and *factors* (for now, you can assume that *factor* class is used to store categorical variables) in separate columns. Also *matrices* are used when programmers want to achieve great speed in mathematical computation. *Data frames* are resembling tables from popular spreadsheet software. Lets look:

```
matrixVariable <- matrix(c(1:10), nrow = 2)
matrixVariable
```

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

```
class(matrixVariable)
```

```
## [1] "matrix"
```

```
dfVariable <- data.frame(x1 = 1:5, x2 = 6:10)
dfVariable
```

```
##   x1 x2
## 1  1  6
## 2  2  7
## 3  3  8
## 4  4  9
## 5  5 10
```

```
class(dfVariable)
```

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

Lists are... lists of variables. Each *list* element can be of different *class* and length. To grasp the idea of *lists* it will be best to present it with example:

```
listVariable <- list(x1 = c("a", "b"), x2 = 1:4, x3 = matrix(c(1:6), nrow = 2))
listVariable
```

```
## $x1
## [1] "a" "b"
##
```

```
## $x2
## [1] 1 2 3 4
##
## $x3
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```
class(listVariable)
```

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

```
class(listVariable$x1)
```

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

```
class(listVariable$x2)
```

```
## [1] "integer"
```

```
class(listVariable$x3)
```

```
## [1] "matrix"
```

There are plenty of other *classes*, e.g. for *time variables*, however mentioned above are the basic ones you will deal mostly. Also because they are so often used, you should learn how to recognize their structure at a glance. Later on, I will present you how (and when) each of this variables types can be used in work.

3.2.2.1 Naming Variables

First of all, all names are *case-sensitive*, which means that **R** recognize variables named `RVariable`, `rVariable` and `Rvariable` as three different objects. Second thing to remember is that variable name *have to* start with a letter and may contain only letters, numbers and symbols: `.` (dot) and _ (underscore). There are also some *good practices* in naming variables (after Hadley Wickham Style guide):

- use lowercase to names variables (and functions)
- use nouns to name variables (and verbs for functions)
- try to be precise when naming
- try to be concise when naming
- use underscore _ to separate words (snake_case) e.g. `first_variable`
- *some other guidelines suggest using camel cases e.g.* `firstVariable`

And the golden rule should be - whatever guideline you follow – be consequent!

3.3 Math operations

In **R** we use standard math operators `+ - * /` to perform addition, subtraction, multiplication and division. Symbol `^` indicates that we want to use power, and `sqrt` to make square root. OK, so whats the name of a function to get nth root? Probably you remember from math lessons that $\sqrt[n]{x} = x^{\frac{1}{n}}$, thus you can just write `x^(1/n)`. To change order of operation (which are following mathematics rules) use brackets `()`. Other important mathematical functions are `%%` for modulo, and `%/%` for integer division.

```
5 + 2
```

```
## [1] 7
```

```
11 - 3
```

```
## [1] 8
```

```
(4+7)/9*2
```

```
## [1] 2.444444
```

```
14 %/% 3 + 1
```

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

```
8^(1/3) + 10%%6
```

```
## [1] 6
```

To calculate logarithms there is *build in* function `log()`. It uses as a base Euler's number by default, however you can override it i.e. `log(10, base = 10)`. You can calculate exponential function using `exp()` function. There are also trigonometric functions in **R**: `sin()`, `cos()`, `tan()`, `asin()`, `acos()`, `atan()`. Angles are used/expressed in radians. To transform values from degrees to radians multiply by `pi` and divide by 180. To transform values from radians to degrees multiply by 180 and divide by `pi`. By the way, `pi` is a constant in **R**, meaning that its value is build in the language (similar as Euler number is `exp(1)`).

```
someArc <- 90*pi/180
sin(someArc)
```

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

```
atanValue <- atan(0.89)
atanValue*180/pi
```

```
## [1] 41.66908
```

3.4 Logics

Logical expression are often used in programming. They compare left side with right side arguments of statement. The result of those comparison might be `TRUE` or `FALSE` (in many other languages those are called *Boolean* values) which belong to *class Logical*. In Table 3.1 you will find list of most common logical operators used to build statements. Here is a small *cheatsheet tables*:

Table 3.2: AND Table

	NA	FALSE	TRUE
NA	NA	FALSE	NA
FALSE	FALSE	FALSE	FALSE
TRUE	NA	FALSE	TRUE

Table 3.3: OR Table

	NA	FALSE	TRUE
NA	NA	NA	TRUE
FALSE	NA	FALSE	TRUE
TRUE	TRUE	TRUE	TRUE

Below you can see them in action:

```
5 >= 1
```

```
## [1] TRUE
```

```
10%%2 == 0
```

```
## [1] TRUE
```

```
!FALSE
```

```
## [1] TRUE
```

```
5L | 11.1 <= 6
```

```
## [1] TRUE
```

3.5 Functions

When writing code we generally want to perform some actions on our variables. There is a lot of *build in functions* in base **R** distribution, and a whole Galaxy of *functions* provided by community. Function can be literally any action performed on variables. For instance, there are some build in statistical functions like `t.test()` or `chisq.test()`. Other function can be use to draw some charts and plots, e.g. `plot()`. It's easy to recognize function, since its structure is *name* followed by parentheses *()*. Inside parentheses user provides arguments and options to function. Lets see how it works with one of *build in* functions:

```
tTestResult <- t.test(numericVector, integerVector)
print(tTestResult)
```

```
##
##  Welch Two Sample t-test
##
## data:  numericVector and integerVector
## t = 1.8371, df = 4, p-value = 0.1401
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.7669579  3.7669579
## sample estimates:
## mean of x mean of y
##       3.5       2.0
```

We used `t.test()` function, with two arguments: `numericVector` and `integerVector`. In second function call, we *ordered* **R** to print out the results of statistical test stored in variable `tTestResult` – which is the argument of this function. I'll show you how to write your own function in chapter 3.5.

Chapter 4

Somwehere between basic and useful

4.1 Addressing

4.1.1 Vectors

When you deal with variables you often will want to use only a part of it in your work. Other times you will want to get rid of some values which follow certain criteria. In order to do it you need to know an 'address' of particular value in a *vector*, *data frame*, *list* etc. From now on, I will use some functions while describing it *at hoc*, since I believe the best way to learn them, is to use them. Lets create a *vector* and see what we can do with it.

```
addressVec <- seq(from = 1, to = 20, by = 2)
addressVec
```

```
##  [1]  1  3  5  7  9 11 13 15 17 19
```

Instead of typing all the numbers by hand, we can use `seq()` function, to generate it automatically. This function takes two arguments `from` and `to`, however additionally we can use option `by` which defines increment of the sequence. As your knowledge is growing I can tell you a secret. Often, there is no need to name all the arguments and options in functions body. In the beginning you should use the names, but the more experienced you get, you will notice that you omit them often. Actually we nearly always omit so called *default* options of a function. Use `?seq` to check the help page for this function. You will see that it can take more options than `from`, `to` and `by`, but since they are predefined we don't need to bother and type them (as long as we are OK with *default* settings). So, lets retype our *variable* definition:

```
addressVec <- seq(1, 20, 2)
addressVec
```

```
##  [1]  1  3  5  7  9 11 13 15 17 19
```

The results are identical. Lets get back to addressing issues. Suppose you want to check the fifth element of our `addressVec` variable. To do it we use variable name followed by element number in brackets (or square brackets, if you will).

```
addressVec[5]
```

```
## [1] 9
```

If you want to get rid of fifth element just precede it with `-`.

```
addressVec[-5]
```

```
## [1]  1  3  5  7 11 13 15 17 19
```

Easy. The thing that is worth to mention right now is that **R Core Team** have reason and dignity of human beings so they start numeration of elements with value 1. In some other languages, because of no sensible reasons, numeration starts with 0 – which is horribly annoying. What to do if we want to extract values of more than one element?

```
addressVec[1,5]
```

```
## Error in addressVec[1, 5]: incorrect number of dimensions
```

```
addressVec[-1,-5,-6]
```

```
## Error in addressVec[-1, -5, -6]: incorrect number of dimensions
```

Error occurs with warning that there is incorrect number of dimensions. That's because *vectors* have only one dimension – length. Within square brackets comma separates dimensions. So when we used command `[1,5]` we told R to look for value that address is number 1 in first dimension and number 5 in second dimension. To correct the mistake, we need to provide a vector of numbers from first dimension. We can use `c()` function to create *ad hoc vector* inside square brackets.

```
addressVec[c(1,5)]
```

```
## [1] 1 9
```

```
addressVec[-c(1,5,6)]
```

```
## [1]  3  5  7 13 15 17 19
```

Now something more complicated. Try to figure out what code below does, and what is the result of `which()` function:

```
addressVec[which(addressVec >= 6)] <- 'o..O'
addressVec
```

```
##  [1] "1"    "3"    "5"    "o..O" "o..O" "o..O" "o..O" "o..O" "o..O" "o..O"
```

If you have problems, you can think of this example as a composition of actions. First there is expression `addressVec >=6`, than we use `which()` function with argument from previous expression. Next the result is passed as an address. Last thing is assignment of new value to...

4.1.2 Data frames (and matrices)

When dealing with *data frames*, addressing is possible in two ways. First is similar to *vector* addressing. However, as *data frames* have two dimensions, you will need to express them like `df[1, 2]` – which tells **R** to look up the cell in row 1, column 2. But what if you want to check all the cells in particular row or column? Omit the number, but not the coma – like this: `df[, 3]`. It will display all row values from column 3. And of course you can still use *vectors* when addressing. This way also works for *matrices*.

```
addressDF <- data.frame(C1 = seq(1,10), C2 = seq(11,20), C3 = seq(21,30))
addressDF[5, 2]
```

```
## [1] 15
```

```
addressDF[6, ]
```

```
##   C1 C2 C3
## 6  6 16 26
```

```
addressDF[c(6, 7), 1:3]
```

```
##   C1 C2 C3
## 6  6 16 26
## 7  7 17 27
```

The second option is to use `$` sign followed with column name (and it works only for *data frames* and *lists*). This way, we tell **R** to look in whole column. If we want to display values from particular cells, we put them in brackets after column name, the same way as we do for *vectors*:

```
addressDF$C2[5]
```

```
## [1] 15
```

```
addressDF$C3[c(6:9)]
```

```
## [1] 26 27 28 29
```

When working with *matrices* you will find a lot of functions that apply to them (check help page for `base` package). The one you should familiarize with, if you plan on working with *matrix* operations are:

- `dim()` – returns dimensions of *matrix*
- `rownames()` and `colnames()` – to specify row and column names
- `cbind()` and `rbind()` – to add columns or rows to a *matrix* (respectively)
- `lower.tri()`, `upper.tri()` – lower and upper triangular part of *matrix*
- `t()` – *matrix* transposition
- `outer()` – outer products of *matrix*
- `rowsums()`, `rowmeans()`. `colsums()` and `colmeans()` – tool to calculate sum and mean of rows or columns
- `%*%` – *matrix* multiplication operator (`matrixOne %*% matrix2`)
- `diag()` – diagonals of a *matrix*

You can also check *help* for `Matrix` package which contains dozens of functions to deal with *matrix* algebra and *matrix* transformations.

4.1.3 Lists

As mentioned before each *list* element can have different structure. Thus, addressing is kind of a mixture of all above. First you need to address element of your *list*. You do it with double brackets containing element number following name of variable. You can also use `$` sign with name of the element. Than you use addressing the same way you address *data frames*, *vectors* or *matrices*.

```
addressList <- list(x1 = c('a', 'b'), x2 = 1:4, x3 = matrix(c(1:6), nrow = 2))
addressList[[2]]
```

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

```
addressList$x2[3]
```

```
## [1] 3
```

```
addressList[[3]]
```

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

```
addressList[[3]][, 3]
```

```
## [1] 5 6
```

OK, now simple task for you. There is a function called `colnames()` which allow us to change names in *matrix* variable. To use it you need to put name of *matrix* variable in the parentheses (like this: `colnames(x)`), and assign value to it (e.g. `colnames(x) <- c('one', 'two')). Now change names of of columns in *matrix* that is element of list above to`I', 'Love', 'R'.

4.2 Control flow

Controlling flow of function, script or whole program is the essence of programming. In general we can divide procedures in two big categories: **conditional control** and **iteration control**. In the first category we will find `if...else` statements. They change execution of code depending on state of input or previous results. In the second category we will find tools to repeatedly do execute some code. Typically tools to control repeating of code are `for` and `while` loops. However, in R there are special functions, which are basically `for` loops but have slightly better performance, and are shorter to write. We call them `apply` functions family.

4.2.1 `if...else` and `ifelse` statments

The simplest way to control how the code is executed are conditions. In an abstract form it is something like telling your machine: if the state is red make it blue, else make it green. The most general form it is written like below:

```
if (x) { #writting just X without any condtition is evaluated as x == TRUE
  function(x)
}
####rest of the code####

#Or...
if (x) {
  function(x)
} else { #in this example this block will be evaluated if x == FALSE
  function(y)
}
```

As you see, writing conditionals in their basic form is quite easy. If our script can be in just on of two different states while it start evaluating conditions both of above examples will be suitable. In practice it happens quite often that we need to plan different actions because of dichotomy in the results of previous actions. If our code in this place is going to make some simple things you can consider using `ifelse()` shortcut. It works like below:

```
coinToss <- sample(c(0,1), 3, replace = T)
coinToss
```

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

```
ifelse(coinToss > 0, 'Win!', 'Looser!')
```

```
## [1] "Looser!" "Win!"    "Looser!"
```

But what if your script can have more than two states when you start evaluating conditions? You will need to use `else if()` structure. A simple example is below:

```
coinToss <- sample(c(0,1), 3, replace = T)
sumOfCoins <- sum(coinToss)# we can have four states here 0,1,2,3
sumOfCoins
```

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

```
if (sumOfCoins == 0) {
  'Looser'
} else if (sumOfCoins == 1) {
  'Barely alive'
} else if (sumOfCoins >= 2) {
  'Big Winner!!!'
}
```

```
## [1] "Big Winner!!!"
```

```
#Or because the functions realting to particular conditions are simple, you
#can make nested ifelse
ifelse(sum(sumOfCoins) == 0, 'Looser',
       ifelse(sum(sumOfCoins) == 1, 'Barely alive', 'Big Winner!!!'))
```

```
## [1] "Big Winner!!!"
```

4.2.2 for and while loops

In many cases we want to use the same function over data stored in different *vectors*, columns or data sets. Other time we need to make sequence of computations in which every result depends on the previous one (like, for instance, in Fibonacci sequence). In yet another situation we need to run function over data set, but only till we reach specific value. Of course, we can do everything manually, by copy-pasting code, taping *ctrl+enter* several times and so on. In every programming language we will find tools that help us automatize this procedures, and by side effect make them less prone to error. Those are called loops. Namely, the most common loops used are `for` and `while`. The difference between them, is that in first one we define number of iterations and in the second one we define condition that must be met so the loops is executed. In other words `for (i in 1:10) {x + i}` means that numbers from 1 to 10 will be added to x in a sequence. `while (x < 5) {print(x <- x - 1)}` on the other hand will subtract value `1` from `x` as long as `x` is greater than `0`. You thing that is worth to mention is that `while` loops are general form and `for` loops are special cases. It means that it is possible to rewrite any `for` loop into `while` loop, but not the other way.

Also when working in loops you should always preallocate memory for your output. In simple words it means, that if you are writing loop, that grows the vector instead of just substitute values, **R** needs to copy your previous results in each iteration. Thus making a *vector* of length of your number of iterations (or a *list*) before you start a loop will save you some computational time.

To illustrate how it works in practice lets look on some simple examples:

```
exampleDF <- data.frame(norm = rnorm(100, 5, 1.5),
                        chisq = rchisq(100, 3),
                        pois = rpois(100, 3.33))
distroMeans <- vector('list', length(exampleDF)) #prealocate space

for (i in seq_along(exampleDF)) {
  distroMeans[[i]] <- summary(exampleDF[[i]])
}

distroMeans
```

```
## [[1]]
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
##   1.594   4.011   4.849   4.981   5.957  10.057 
## 
## [[2]]
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max. 
##  0.05314  1.00781  2.15244  2.95046  4.09945 11.08640 
## 
## [[3]]
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
##    0.00    2.00    3.00    3.29    4.00   12.00 
```

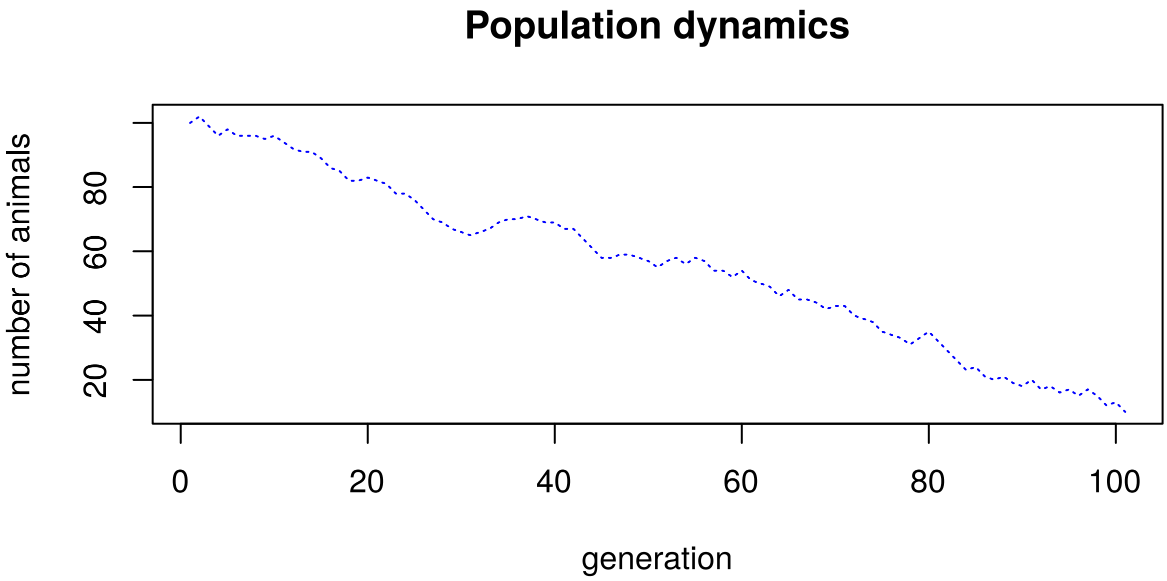


Figure 4.1: Population dynamics

Above example is very simple, and actually can be made without loop. Try to call from your console following command `summary(exampleDF)`. This is called vectorization which will be explained few paragraphs below.

But lets get back to loops and write something that might be useful (yet very simple). Assume that you are investigating animals reproduction. You have a herd that counts 60 animals. The reproduction factor is `1.34`, and survival factor is '0.75'. So the overall growth factor can be calculated as *reproduction factor* * *survival factor*. In each generation there is some noise in reproduction, i.e. from -3 to 2 animals are born with equal probability. This means, that number of animals in each generation directly depends of the outcome of previous generation. We want to make a chart (Fig. 4.1) to visualize population dynamics in `100` generation, thus we decide to use `while (time < 101)` loop.

```
#define initial parameters
N_0   <- 100
rf    <- 1.34
sf    <- 0.75
gf    <- rf*sf
n_gen <- 1:101
#we could repeat this inside loop, however we know the number of iterations
#thus calculating noise vector outside loop is better for performance
noise <- sample(-3:2, 100, replace = T)
t <- 1
geners <- vector('integer', length = length(n_gen))
geners[1] <- N_0
while (t < 101) {
  geners[c(t + 1)] <- floor((geners[t] * gf + noise[t]))
  t <- t + 1
}

plot(n_gen, geners, type = 'l',
     main = 'Population dynamics',
     xlab = 'generation', ylab = 'number of animals',
     col = 'blue', lty = 3)
```

It appears that we found a way to make a species extinct...

4.2.3 `apply` function family

For simple loops there is a shortcut. Namely, there is an `apply` function family that performs defined function over vector, list, matrix etc. Probably your need to use it will be reduced to `lapply()` and `sapply()` function in the beginning, thus I will focus on them here. From my experience it is better to first understand and correctly use `for` and `while` loops, than switch to `lapply()` and `sapply()` and than learn how to correctly use functions like `tapply()`, `mapply()`, `vapply()`. `sapply()` generally is the same as `lapply()`, the only difference is that the later function results in *list* of the same length as first argument. Each element of list contains results of function that was applied to the first argument. Simplified version tries to simplify results to *vector*, *matrix* or *array*. One tricky thing is that we we want to use function that takes more arguments than just our arguments of values over which we want to iterate we need to use a bit different syntax. In general, lets say our list is named as `X`, our function is `SomeFunction`, which takes arguments `A` and `B` (`SomeFunction(A, B)`. Lets asume that each element of`X`is used as an argument`B`in consequent iterations, while`A`is fixed and equals`10`. The correct for of using lapply (or sapply) will be:`lapply(X, SomeFunction, A = 10)‘.

```
testApply  <- list(x = 1:10, b = rep(c(4,5), times = 5), c = rpois(5, 25))
testLapply <- lapply(testApply, mean)
testSapply <- sapply(testApply, mean)

#Here we combine previous results with another lapply,
#and use function with multiple arguments.
lapply(testSapply, rpois, n = 15)
```

```
## $x
##  [1]  4  5  6  5  4  4  5  6  7  6 11  7  4 10  5
##
## $b
##  [1] 5 6 5 7 1 7 1 0 2 6 4 4 4 9 4
##
## $c
##  [1] 23 30 24 31 24 30 34 28 30 37 21 23 30 21 29
```

```
head(sapply(testSapply, rpois, n = 15))
```

```
##      x b  c
## [1,] 6 6 20
## [2,] 5 3 34
## [3,] 8 2 26
## [4,] 5 7 28
## [5,] 7 3 26
## [6,] 1 5 32
```

It is also possible to use so called anonymous functions (simplifying – function that does not have name and exists temporally) as an argument.

```
testApplyFun <- 1:5
lapply(testApplyFun, function(x) x + 0.5)
```

```
## [[1]]
## [1] 1.5
##
## [[2]]
## [1] 2.5
##
## [[3]]
## [1] 3.5
```

```
##
## [[4]]
## [1] 4.5
##
## [[5]]
## [1] 5.5
```

```
sapply(testApplyFun, function(x) x + 0.5)
```

```
## [1] 1.5 2.5 3.5 4.5 5.5
```

On the basis of previous code that calculated population dynamics, we can write general function and use it with `lappy()` to estimate different population dynamics depending on initial population size. Than, by addition of consequent lines representing populations with different starting points we create plot for all 4 populations (Fig. 4.2).

```
popDyn <- function(N_0, rf = 1.34, sf = 0.75, n_gen) {
  gf    <- rf*sf
  noise <- sample(-3:2, 100, replace = T)
  t <- 1
  geners <- vector('integer', length = n_gen)
  geners[1] <- N_0
  while (t < 101) {
  geners[c(t + 1)] <- floor((geners[t] * gf + noise[t]))
  t <- t + 1
  }
  return(geners)
}

N_0   <- c(50, 100, 150, 200, 250)
popSizes <- lapply(N_0, popDyn, n_gen = 101)

n_gen = 101
plot(1:n_gen, popSizes[[5]], type = 'l',
     main = 'Population dynamics',
     xlab = 'generation', ylab = 'number of animals',
     col = 'blue', lty = 3, ylim = c(-10, max(popSizes[[5]])))
lines(1:n_gen, popSizes[[4]], col = 'green')
lines(1:n_gen, popSizes[[3]], col = 'red')
lines(1:n_gen, popSizes[[2]], col = 'yellow')
lines(1:n_gen, popSizes[[1]], col = 'black')
```

4.3 Operation on Vectors

Old proverb:

> *The power of R is vectorization*

Indeed, vectorization is one of the most useful features of **R** environment. In short words, many functions are constructed in such a manner, that one can avoid using loops (since, they are often very slow). When you begin your journey with programming, you will probably not notice the difference in computation speed between *vectorized way* and *loop way*. Nonetheless, what will you find attractive, is that in many cases writing in vectorized form feels more natural. So how it works? Most basic functions are *written and compiled* in very fast, low level languages like **C** or **FORTRAN**. It makes computation way faster, but still we use easy **R** syntax. Instead of running the *precompiled* function on each of the elements of vector we actually can

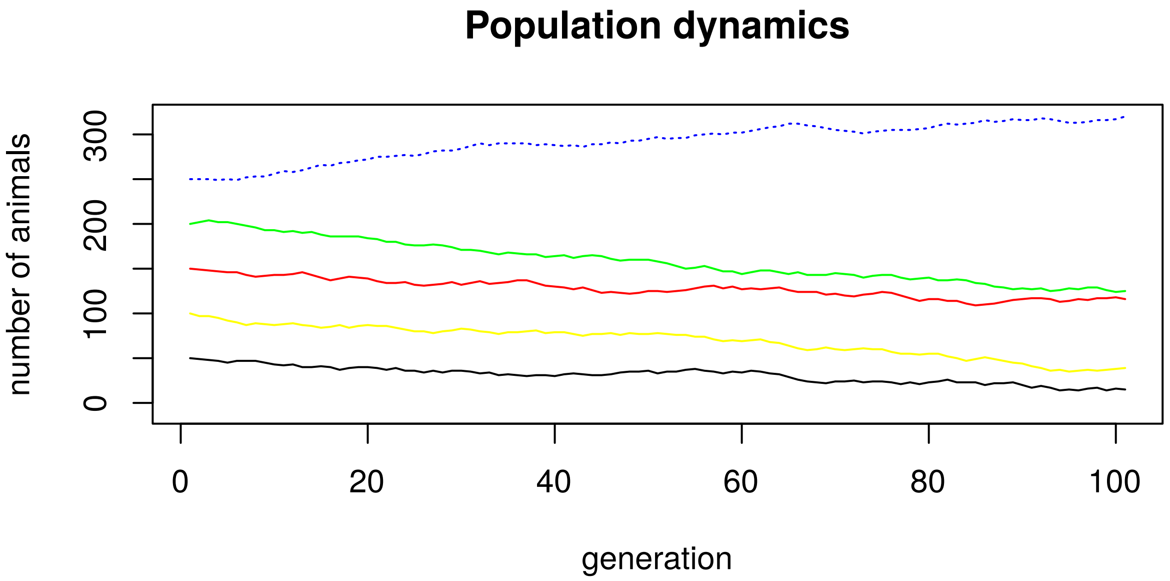


Figure 4.2: Population dynamics with different starting point

pass the vector into function – and **R** will know what to do. The speed is achieved because when we use function on each element, **R** needs to figure out what to do several (or even hundred or thousands of) times. However, when we pass whole vector into function, it needs to find out what to do only once. Sometimes, we need to use some other functions, which are not meant to process vectors (or you want to use some arbitrary functions on *matrix*, *data frame* or *list*). In those cases you will need to parse your data more than once to function. In **R** you can do it by using loops or by using so called `apply` family functions. When I was learning **R** it was most confusing thing to me, however once you get it, they become fairly easy to use. The most important difference for you as a beginner is that when using loops, you should make some memory allocation – in human language, before you run loop, you should create vector which will store results. The loops achieve highest speed when your vector can fit all the values from loop. If you do not know how big your vector will be, you need to relay on **R** ability to re-allocate the memory, which is **S L O W**. `apply` family hopefully takes care of this problem, so every time you know how to – use it. It will save you time and frustration of using loops. I will cover basic use of this functions later on examples.

4.4 Randomization and distribution

Random sampling is quite easy task. There is nice `sample()` function, that allow you to do that. You can sample from *numeric*, *integer* or even *character vectors*. You can even assign probabilities to particular elements of vector (we will not need that at the moment). So lets make some fun and make virtual casino. We will take two dice and will roll it 1000 times. If the sum of each result is odd and smaller than `5` we will earn 30EUR. If it is odd and higher than `5`, we will get 50EUR. However if the sum is even number we loose 70EUR. Lets see if we can beat casino... First lets define our dice and rolls.

```
niceDice <- seq(1,6)
rollDiceOne <- sample(niceDice, 1000, replace = T)
rollDiceTwo <- sample(niceDice, 1000, replace = T)
```

OK, you heard about power of vectorization already, so lets sum up both vectors:

```
sumDiceRolls <- rollDiceOne + rollDiceTwo
```

Next lets make some use of knowledge we got and substitute our results with some cash...

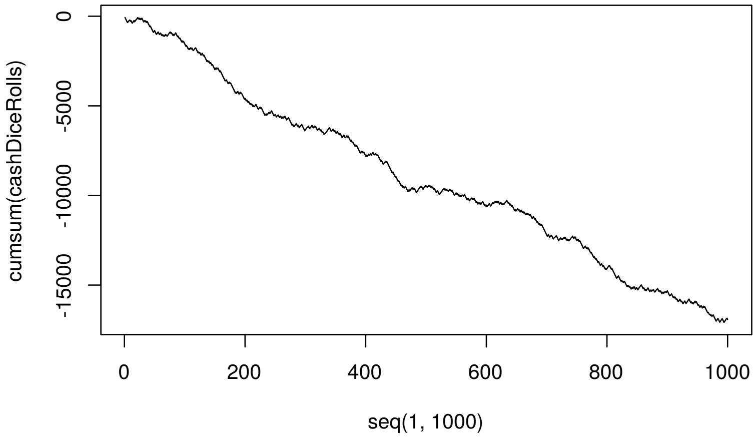


Figure 4.3: How to lose cash in casino

```
cashDiceRolls <- sumDiceRolls
cashDiceRolls[which(cashDiceRolls %% 2 == 1 & cashDiceRolls <= 5)] <- 30
cashDiceRolls[which(cashDiceRolls %% 2 == 1 & cashDiceRolls > 5)] <- 50
cashDiceRolls[which(cashDiceRolls %% 2 == 0)] <- -70
sum(cashDiceRolls)
```

```
## [1] -70000
```

OK. What happened? We put our substitutions in very wrong order... After first two substitutions we have only even numbers in our vector... So how to fix it? Start with assigning the highest absolute value, and after all change it to negative one.

```
cashDiceRolls <- sumDiceRolls
cashDiceRolls[which(cashDiceRolls %% 2 == 0)] <- 70
cashDiceRolls[which(cashDiceRolls %% 2 == 1 & cashDiceRolls > 5)] <- 50
cashDiceRolls[which(cashDiceRolls %% 2 == 1 & cashDiceRolls <= 5)] <- 30
cashDiceRolls[which(cashDiceRolls  == 70)] <- -70
sum(cashDiceRolls)
```

```
## [1] -16920
```

Now, that's not very nice... We lost a lot of cash... But we can also track how our luck changed. Maybe we were above `0` for a while, before things got South? Or maybe we were doomed since the beginning? Lets make a plot (Fig. 4.4) to evaluate our progress.

I think that your love to **R** is being deeper since I showed you how it can save a lot of your money!

In nowadays stochastic analyses are more, and more popular. With **R** it is quite easy to generate random numbers, sample and do all the *mumbo jumbo* on data. There are few functions that cover *classical* distributions: normal, Poisson, binomial, uniform (and few others), that are actually build in base **R** distribution (full list you will find here). Some more sophisticated stuff is usually covered by some packages you will need to download by yourself. You will easily find them by querying Google like this Pearson distribution in R. By using this technique it is also possible to find other useful packages for dealing with distributions (try to search for package that allows you to sample from trimmed distribution). All the distribution packages follow the same schema when naming functions. They use letters: `d`, `p`, `q` and `r` followed by abbreviation of distribution name to generate: density, distribution function, quantile function and random numbers – respectively. For instance, lets generate ten random numbers from *beta distribution*, with parameters `10` and `3`:

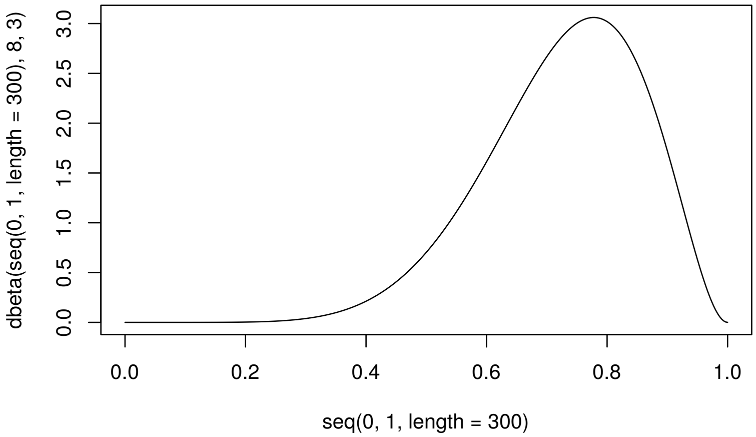


Figure 4.4: Example of beta distribution

```
rbeta(25, 10, 3)
```

```
##  [1] 0.7986224 0.7445944 0.6305413 0.8489741 0.9267320 0.8199201 0.6346696
##  [8] 0.6471016 0.7695083 0.7931333 0.7415597 0.7087800 0.6565048 0.6605388
## [15] 0.7972370 0.8208387 0.8262428 0.6884806 0.6755203 0.6744089 0.6449869
## [22] 0.8355298 0.6800542 0.9447348 0.6337499
```

Or we can make this nice plot (Fig. 4.4) for density of *beta distribution* with parameters `8` and `3`:

4.5 `tidyverse` idea and `dplyr` library

The whole idea of tidy data comes from one of most famous **R** developers – Hadley Wickham. In on of his papers (Wickham, 2014) he described procedures for generating and cleaning data in standardized manner. Many packages right now are designed to work best with data structured according to this publication. Eventually it lead to `tidiverse` – tools tailored for data science with common syntax and philosophy (Wickham, 2017). One of the most useful packages that are included in `tidiverse` (Wickham, 2017) are `tidyr` (Wickham and Henry, 2018) and `dplyr` (Wickham et al., 2017). First helps us to swap from 'wide' to 'long' table format (and back). The second package contains set of tools to easily manipulate rows, columns or even single cells in *data frame*. It is extremely powerful tool, which speeds up work with data sets so much, that after few times dealing with it, you will leave traditional spread sheet forever.

4.5.1 Wide vs. long tables

Lets start with changing wide format table into long format table.

```
wideTable <- data.frame(male = 1:10, female = 3:12)
wideTable
```

```
##    male female
## 1     1      3
## 2     2      4
## 3     3      5
## 4     4      6
## 5     5      7
## 6     6      8
## 7     7      9
## 8     8     10
## 9     9     11
## 10   10     12
```

Than we just make a little *mumbo jumbo* and change it to long format:

```
library('tidyr')
library('dplyr')
longTable <- gather(wideTable, key = sex, value = number)
longTable[7:12, ]
```

```
##       sex number
## 7    male      7
## 8    male      8
## 9    male      9
## 10   male     10
## 11 female      3
## 12 female      4
```

That is how easy it can be done. But what if table is more complicated?

```
wideTable2 <- data.frame(male = 1:10,
                         female = 3:12,
                         type = rep(c('bacteria', 'virus'), times = 5),
                         group = rep(c('a', 'b'), each = 5))
wideTable2
```

```
##    male female     type group
## 1     1      3 bacteria     a
## 2     2      4    virus     a
## 3     3      5 bacteria     a
## 4     4      6    virus     a
## 5     5      7 bacteria     a
## 6     6      8    virus     b
## 7     7      9 bacteria     b
## 8     8     10    virus     b
## 9     9     11 bacteria     b
## 10   10     12    virus     b
```

```
gather(wideTable2, sex, number, -c(3, 4))[7:13, ]
```

```
##        type group    sex number
## 7  bacteria     b   male      7
## 8     virus     b   male      8
## 9  bacteria     b   male      9
## 10    virus     b   male     10
## 11 bacteria     a female      3
## 12    virus     a female      4
## 13 bacteria     a female      5
```

In example above, using expression `-(3, 4)` we indicated that we do not want to gather columns 3 and 4.

Even more complicated?

```
wideTable3 <- data.frame(male = rep(1:10, each = 2),
                         female = rep(3:12, times = 2),
                         type = rep(c('bacteria','virus'), times = 10),
                         group = rep(c('a','b'), each = 10),
                         day = 1:10)
wideTable3[7:13, ]
```

```
##    male female     type group day
## 7     4      9 bacteria     a   7
## 8     4     10    virus     a   8
## 9     5     11 bacteria     a   9
## 10    5     12    virus     a  10
## 11    6      3 bacteria     b   1
## 12    6      4    virus     b   2
## 13    7      5 bacteria     b   3
```

```
gather(wideTable3, sex, number, 1:2) %>%
  spread(group, number) %>% .[7:13, ]
```

```
##        type day    sex  a  b
## 7  bacteria   7 female  9  9
## 8  bacteria   7   male  4  9
## 9  bacteria   9 female 11 11
## 10 bacteria   9   male  5 10
## 11    virus   2 female  4  4
## 12    virus   2   male  1  6
## 13    virus   4 female  6  6
```

Here our data set had additional column `group` storing values `a` and `b`. But imagine that `group` is not a variable, but it just stores the names of variables - which are a and b. This somehow might be frustrating, to decide if those are really separate variables or not. You might run into problem, that your column named `environmental factor` contains values: *pH*, *conductivity*, and *oxygen concentration*. This would be straightforward as each of this is different variable and you should spread this column into three different variables. On the other hand you might see a column that contains *minimum temperature* and *maximum temperature*. This would not be as straightforward and decision upon spreading this column would strongly depend on the context. Nonetheless, using `spread()` function we were able to transform values from this column as separate columns. We also used *piping operator* `%>%`. It is a shortcut, which allows us to pass the left side as an argument to the function on the right side of operator. In general it means that writing `a %>% funtion(b)` actually is translated into `function(a, b)`. In the beginning this idea might be not very useful for you, but with time it will get helpful. Mainly because your code gets better structure and you can perform multiple operations without storing it in variables (which you do not want, since it consumes memory).

There is also one more common case that I should shortly mention - compound variable. It is a variable that stores multiple values in a single column. E.g. city and district, age and sex, sex and smoking, blood type and RH, etc. With `tidyr` is very easy to deal with it.

```
wideTable4 <- data.frame(type = rep(c('bacteria.a', 'virus.a'), times = 5))
wideTable4
```

```
##          type
## 1  bacteria.a
## 2     virus.a
## 3  bacteria.a
## 4     virus.a
## 5  bacteria.a
## 6     virus.a
## 7  bacteria.a
## 8     virus.a
## 9  bacteria.a
## 10    virus.a
```

```
wideTable4 %>% separate(type, c('organism', 'type'), sep = '\\.')
```

```
##    organism type
## 1  bacteria    a
## 2     virus    a
## 3  bacteria    a
## 4     virus    a
## 5  bacteria    a
## 6     virus    a
## 7  bacteria    a
## 8     virus    a
## 9  bacteria    a
## 10    virus    a
```

4.5.2 World of dplyr

dplyr is a library containing several useful functions, designed to ease all kinds of transformations, selections and filtering of your data frame. As the number and possibilities of functions are really huge, here I will concentrate only on some mostly used ones. Of course there is a well described documentation if you want to get deeper.

4.5.3 select columns and filter rows

Those are the most basic operations on *data frames*. select() function allows you to choose which columns you use. The biggest advantage of using this function instead of simple addressing, is that you can use some special functions inside it: starts_with(), ends_with(), contains(), matches(), num_range() which are helpful when using large data sets with numerous columns. There is also similar function rename() which can be used to change variable names, but in results (contrary to select()) it keeps all the variables. Lets look how the thing works on some simple examples:

```
head(select(wideTable3, starts_with('gr')), 3)
```

```
##   group
## 1     a
## 2     a
## 3     a
```

```
head(select(wideTable3, -starts_with('gr')), 3)
```

```
##   male female     type day
## 1    1      3 bacteria   1
## 2    1      4    virus   2
## 3    2      5 bacteria   3
```

```
head(select(wideTable3, contains('ale')), 3)
```

```
##   male female
## 1    1      3
## 2    1      4
## 3    2      5
```

```
head(rename(wideTable3, Male = male), 3)
```

```
##   Male female     type group day
## 1    1      3 bacteria     a   1
## 2    1      4    virus     a   2
## 3    2      5 bacteria     a   3
```

Filtering rows is also straightforward. Inside function `filter()` you can use logical operators (`&`, `|`, `xor`, `!`), comparisons (e.g. `==` or `>=`), or functions (`is.na()`, `between()`, `near()`).

```
filter(wideTable3, group == 'a')
```

```
##    male female     type group day
## 1     1      3 bacteria     a   1
## 2     1      4    virus     a   2
## 3     2      5 bacteria     a   3
## 4     2      6    virus     a   4
## 5     3      7 bacteria     a   5
## 6     3      8    virus     a   6
## 7     4      9 bacteria     a   7
## 8     4     10    virus     a   8
## 9     5     11 bacteria     a   9
## 10    5     12    virus     a  10
```

```
filter(wideTable3, type != 'bacteria')
```

```
##    male female  type group day
## 1     1      4 virus     a   2
## 2     2      6 virus     a   4
## 3     3      8 virus     a   6
## 4     4     10 virus     a   8
## 5     5     12 virus     a  10
## 6     6      4 virus     b   2
## 7     7      6 virus     b   4
## 8     8      8 virus     b   6
## 9     9     10 virus     b   8
## 10   10     12 virus     b  10
```

```
filter(wideTable3, near(female, 5))
```

```
##   male female     type group day
## 1    2      5 bacteria     a   3
## 2    7      5 bacteria     b   3
```

4.5.4 mutate and transmute

Both functions are widely used in data manipulation in **R**. Their main purpose is to create new variable (usually from existing ones) in data frame. Lets look on examples:

```
select(wideTable3, male) %>%
  mutate(cumulativeMaleSum = cumsum(male)) %>%
  head(5)
```

```
##   male cumulativeMaleSum
## 1    1                 1
## 2    1                 2
## 3    2                 4
## 4    2                 6
## 5    3                 9
```

```
select(wideTable3, male, female) %>%
  mutate(cumulativeMaleSum = cumsum(male),
         femaleLog = log(female),
         cumSumFemaleLog = cumsum(femaleLog)) %>%
  head(5)
```

```
##   male female cumulativeMaleSum femaleLog cumSumFemaleLog
## 1    1      3                 1  1.098612        1.098612
## 2    1      4                 2  1.386294        2.484907
## 3    2      5                 4  1.609438        4.094345
## 4    2      6                 6  1.791759        5.886104
## 5    3      7                 9  1.945910        7.832014
```

As you can see in second example, when we use `mutate()` function, the newly created variables (in this example `femaleLog`) are available immediately so we can use them to create another variable (in this case `cumSumFemaleLog`) within one function call. In this example, I also used piping operator, because calculating intermediate steps and storing them as a result, which can be used later is useless, as we are interested only in the final outcome. The biggest advantage of this procedure is that we keep our *Environment* clean and preserve memory – which is very important in long and memory consuming projects. Last but not least, the difference between `mutate()` and `transmute()` is that the latter do not preserve all variables, only the ones you created.

4.5.5 group_by and summarise

`summarise()` is commonly used function on grouped data. It allows to calculate many typical descriptive statistics (like `mean()` or `quantile()`) for particular groups in you data set, as well as it can count number of observations (`n()` function), or number of unique observations (`distinct()`). To see how it works in practice lets look back on our example and calculate mean value for `males` and `female`, as well as day range and number of cases for groups derived from `type`.

```
wideTable3 %>%
  group_by(type) %>%
  summarise(meanF = mean(female),
            meanM = mean(male),
            numberOfDays = (range(day)[2]-range(day)[1]),
            numberOfCases = n())
```

```
## # A tibble: 2 x 5
##   type     meanF meanM numberOfDays numberOfCases
##   <fct>    <dbl> <dbl>        <int>         <int>
## 1 bacteria     7   5.5            8            10
## 2 virus        8   5.5            8            10
```

Easy.

4.6 Is there anything more in dplyr library?

Yes. Actually I presented here only very very tiny fracture of `dplyr` possibilities just to familiarize you with syntax and using piping operator. When you go deeper into world of data frames transformation, you will find other commonly used functions like: different kinds of joining data frames (similar to **SQL** joining tables), conditional selecting, filtering, renaming rows and columns, extracting values or arranging your data frame. Thankfully this procedures are so common that even if you won't grasp it immediately from functions description, you will still find hundreds of tutorials on the web, or help in *StackOverflow*.

Chapter 5

Lets do some math!

5.1 Models

Simple statistical model

OK. There is no such thing as simple statistical model. However there are lots of packages that will make you suffer less. In fact this is one of biggest **R** advantages, that you can make even very sophisticated statistical modelling without any knowledge on programming since you use *black boxes.* When you are dealing with classic statistical models many of them are included in **base R** distribution – like linear or generalized additive models. You probably need to use mixed effect models, at some point. Good news is that there is a very nice and quite straightforward to use package called `lme4`. Every time you run into problem, and you do not know what to use for statistical modelling, or how to perform full procedure, just query Google. There are hundreds of blogs, web pages and *Stack Overflow* discussion on it.

* Other models

Besides *simple statistical models*, there is a lot of other models. The scope of this book is not to discuss dichotomies in classification of models. To keep things simple, we will just stick to *models* that can be described with at least one of the following adjectives: *stochastic*, *mechanistic* or belong to *differential equations systems.*

5.2 Packages

Other, usually dynamic models, require more knowledge, experience and some packages. Before you start looking for more tailored solutions, install following packages: `deSolve`, `fitdistrplus`, `rriskDistributions` and `truncdist`. First one contains tools for solving differential equations sets, second and third provides you tools to deal with distributions (such as comparing distributions or estimating its parameters), and the last one allows you to use truncated distributions.

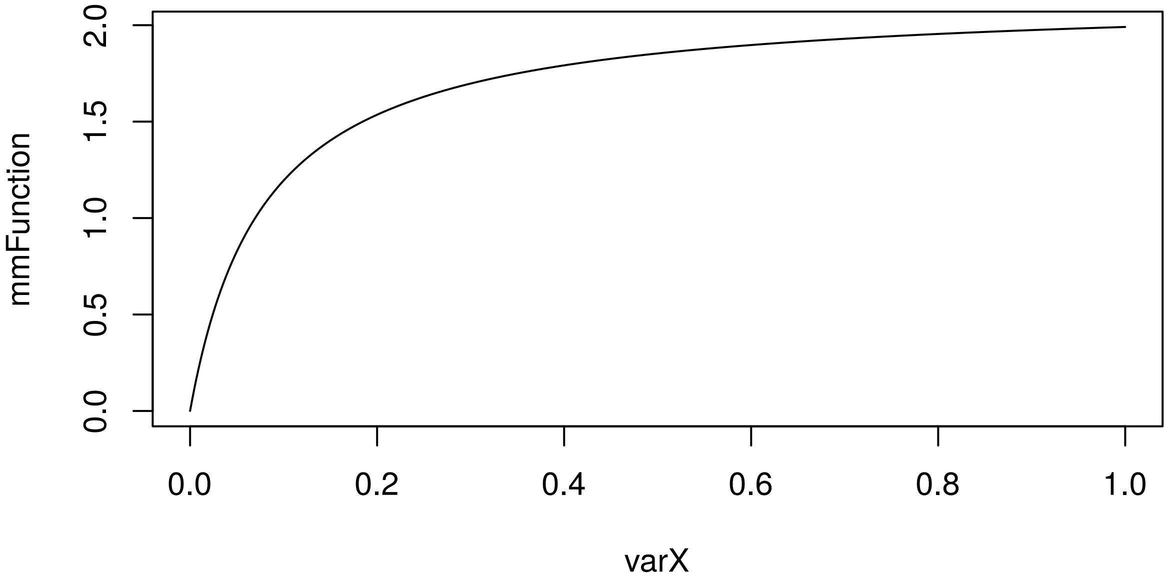


Figure 5.1: Simple Michaelis-Menten function

5.3 Simple mechanistic model and noise

To show some basic mechanistic model, we will use the well known Michaelis-Menten function: $f(x) = \frac{ax}{b+x}$, with *parameter* $a = 2.15$ and *parameter* $b = 0.08$.

```
parA <- 2.15
parB <- 0.08
varX <- seq(0,1, length.out = 1000)
mmFunction <- parA * varX/(parB + varX)
plot(mmFunction~varX, type = 'l')
```

As you can see, as long as you have simple equation it is very straightforward to translate it into R. But we want something more useful, for instance adding noise or uncertainty to our model. Lets assume, that we are not sure what is the value of *parameter a*. Lets say that from previous research and expert knowledge we assume that it can be as high 2.5, but never is lower than 1.8. Also it usually takes value of 2.15. Knowing this we can use PERT distribution to include uncertainty in model.

```
library('mc2d')
sParA <- rpert(1000, 1.8, 2.15, 2.5)
parB <- 0.08
mmSAFunction <- sParA * varX/(parB + varX)
```

We added some uncertainty to our function. But what with *parameter b*, how our uncertainty on its value affects the outcome? Lets assume that it's value is normally distributed with $mean = 0.8$ and $sd = 0.23$.

```
sParB <- rnorm(1000, 0.08, 0.023)
mmSABFunction <- sParA * varX/(sParB + varX)
```

Let me put this two functions on plots side by side:

```
par(mfrow = c(1,2))
plot(mmSAFunction~varX, type = 'l')
plot(mmSABFunction~varX, type = 'l')
par(mfrow = c(1,1))
```

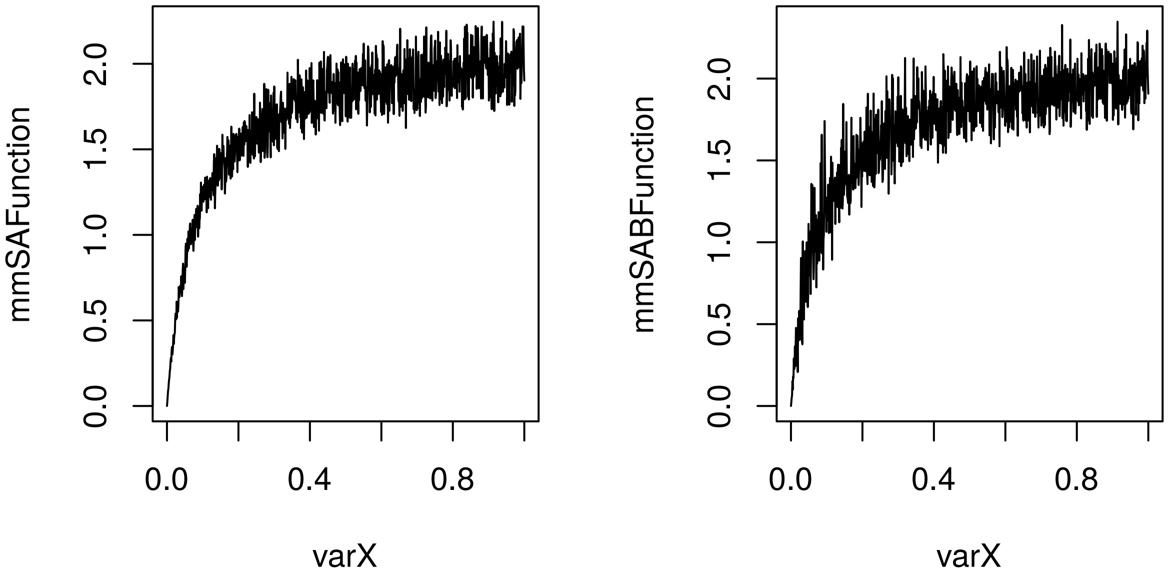


Figure 5.2: Michaelis-Menten function with stochastic parameter A (left) or A and B (right)

It seems that inclusion of uncertainty on value of *parameter b* does change our outcome. Lets also include some variability into the outcome. We can assume that for some reasons value of function will vary more than just our uncertainty about value of both *parameters*. And usually it varies from $f(x)$ by some value from uniform distribution. The 25^{th} and 75^{th} percentile are given by: `-0.13` and `0.17`. To calculate parameters of this distribution we will use `rriskDistributions` package.

```
library('rriskDistributions')
unifParams <- get.unif.par(p = c(0.25, 0.75), q = c(-0.13, 0.17), plot = F)
fVariability <- runif(1000, min = unifParams[1], max = unifParams[2])
mmFSFunction <- ((sParA * varX/(sParB + varX)) + fVariability)
plot(mmFSFunction~varX, type = 'l')
```

It seems that there are differences between those three plots. So lets overlay them one on another to inspect visually differences.

```
plot(varX, mmFunction, type = 'l',
     col = 'black', ylim = c(0, 3),
     main = 'Michaelis-Menten function', xlab = 'x', ylab = 'f(x) = a*x/(b+x)')
lines(varX, mmSAFunction, col = 'blue')
lines(varX, mmSABFunction, col = 'green', lty = 'dashed')
lines(varX, mmFSFunction, col = 'red', lty = 'dotted')
```

5.4 Solving differential equations

5.4.1 The simple epidemiological

SIR (Susceptible, Infectious, Recovered) is a typical compartmental model for epidemiology. Since it has only 3 compartments, it is hard to find an easier one to model with differential equations. In our example we will use slightly more complicated example with four compartments – SEIR (Susceptible, Exposed, Infectious, Recovered). We begin with defining parameters of initial population. We need to define: birth and death rate, transmission (β), γ ($\frac{1}{\gamma}$ defines infectious period) and α ($\frac{1}{\alpha}$ defines latent period). Than we define time and initial conditions – number of all specimens, population size, number of exposed, number of

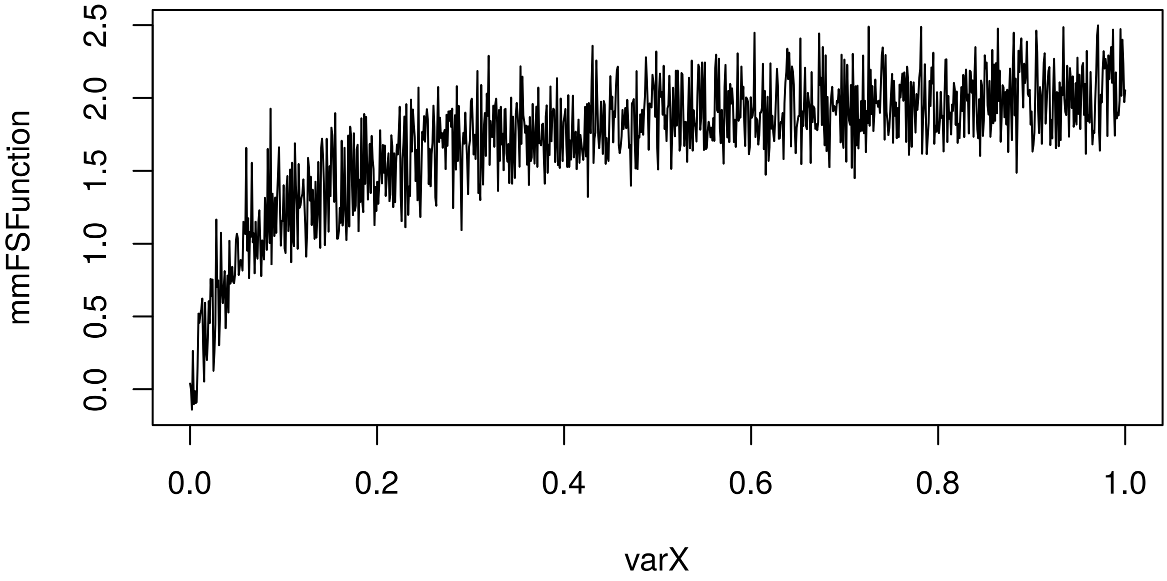


Figure 5.3: Michaelis-Menten function fully stochastic

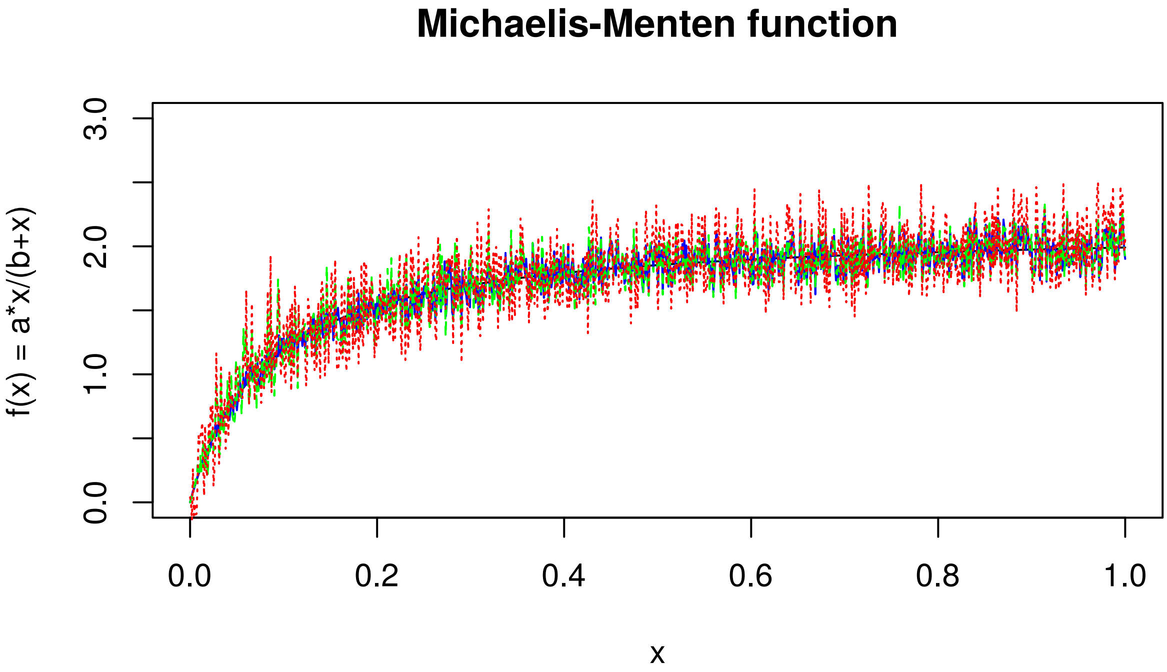


Figure 5.4: Michaelis-Menten function with different levels of stochasticity

initially infected, number of initially recovered, which we combine in `initial` values vector.
Main step of this procedure is to define function which will be used to solve *Ordinary Differential Equations (ODE)*. This function takes three parameter – t (which is time sequence), state (which are our initial conditions) and parameters. In the body of the function we define differential equations which connect our compartments. To solve the system of differential equations we use `ode()` function from `deSolve` package and save our results in `modelOutput` variable, which we will later use to make a plot.

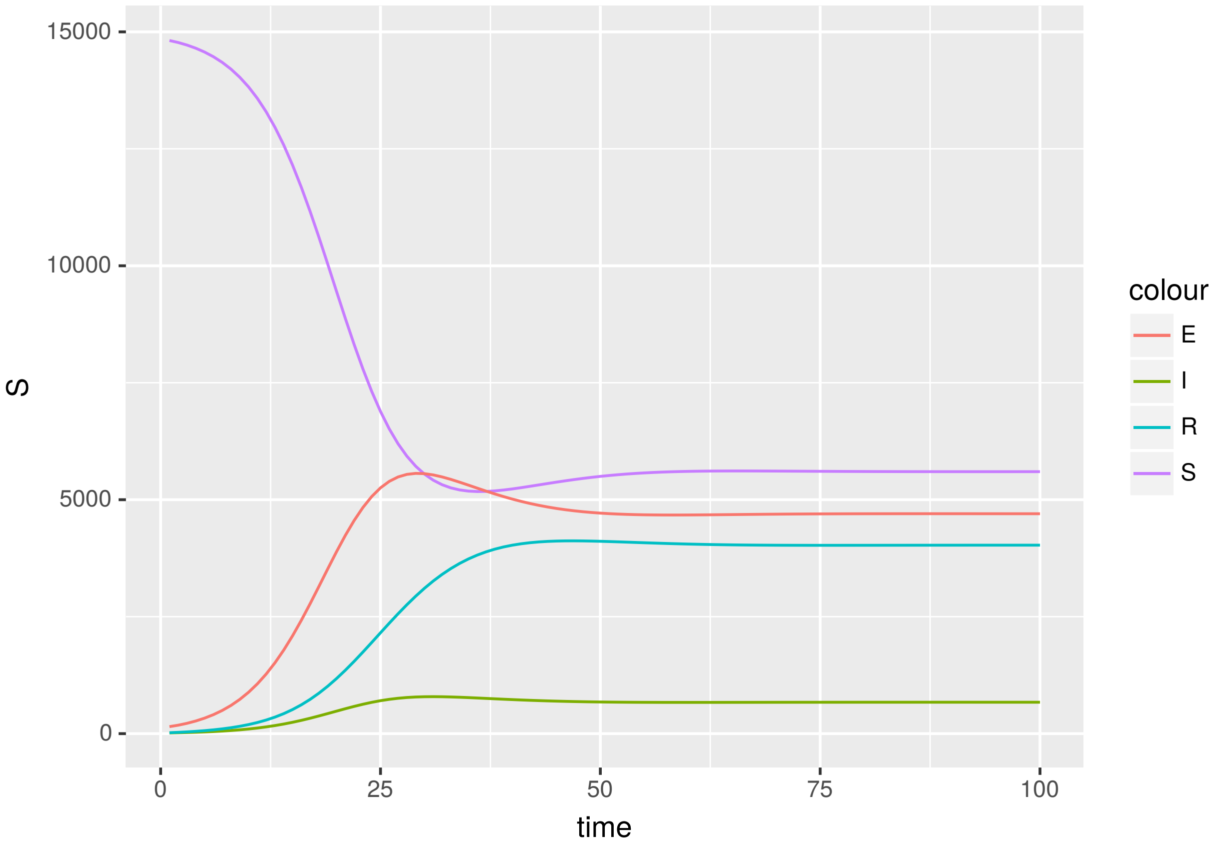
```
parameters <- c(b     = 0.1,        # birth rate
                d     = 0.1,        # death rate
                beta  = 0.00025,    # transmission parameter
                gamma = 0.6,        # 1/gamma = infectious period
                a     = 0.1        # 1/a = latent period
                )

time <- 1:100

# Initial conditions
N = 15000    # population size
E0 = 150     # initialy exposed
I0 = 15      # initialy infected
R0 = 20      # initialy recovered

initial <- c(S = N - (E0 + I0 + R0), E = E0, I= I0, R = R0)

# ODE Model
fSEIR <- function(t, state, parameters) {
  with(as.list(c(state, parameters)), {
    dS = b * (S + E + I + R) - beta * S * I - d * S
    dE = beta * S * I - a * E - d * E
    dI = a * E - gamma * I - d * I
    dR = gamma * I - d * R
    return(list(c(dS, dE, dI, dR)))
  })
}

modelOutput <- deSolve::ode(y = initial, func = fSEIR, times = time, parms = parameters)
head(modelOutput)
```

```
##      time        S        E        I        R
## [1,]    1 14815.00 150.0000 15.00000 20.00000
## [2,]    2 14771.93 180.7236 19.41006 27.94129
## [3,]    3 14717.22 220.8724 24.19132 37.72072
## [4,]    4 14649.90 270.7314 29.83648 49.53296
## [5,]    5 14567.85 331.7169 36.66138 63.77320
## [6,]    6 14468.27 405.8173 44.94943 80.96618
```

5.4.2 Ploting SERI model

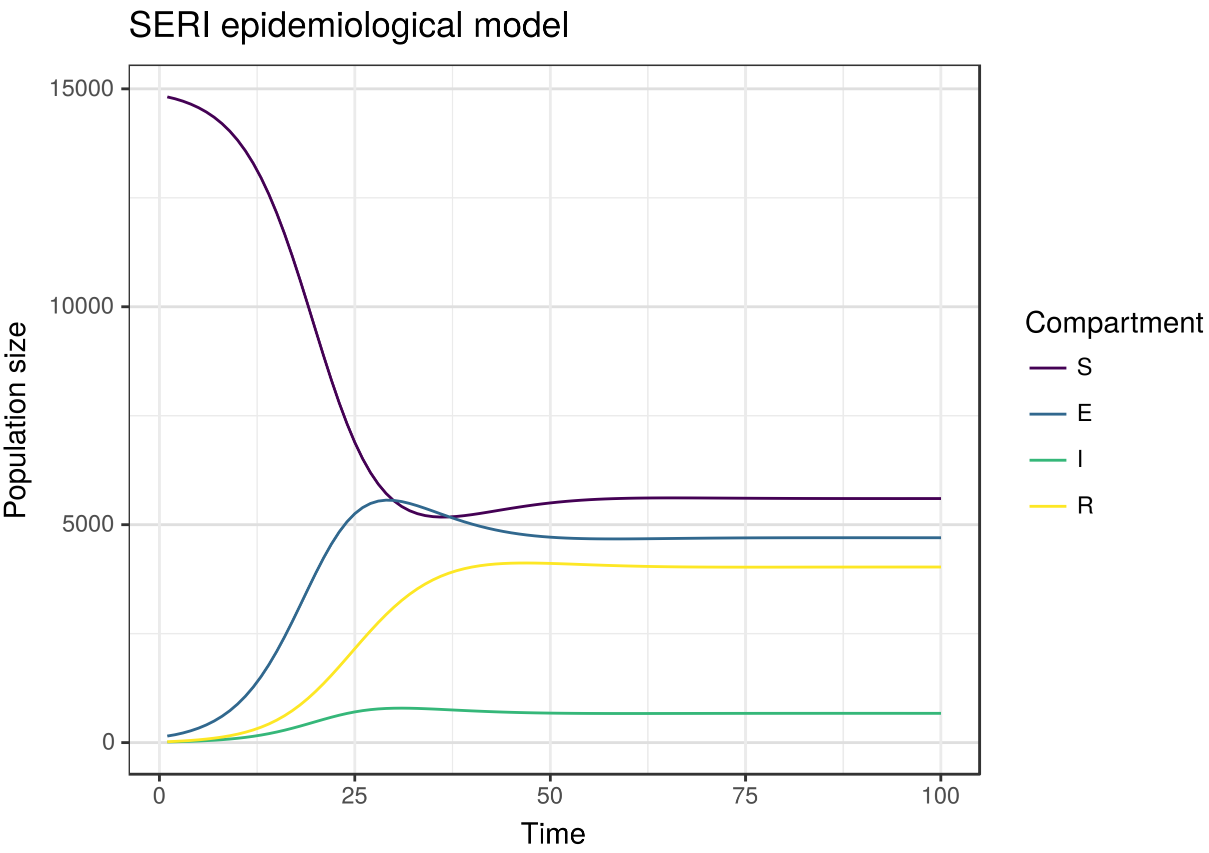
To create a plot using our favorite library `ggplot2` (which will be described with more details in chapter 7) we first need to convert our modelOutput to object of class *data frame*. Than we can use code below to plot our model.

```
modelOutput <- as.data.frame(modelOutput)
ggplot(modelOutput, aes(x = time)) +
  geom_line(aes(y = S, colour = "S")) +
  geom_line(aes(y = E, colour = "E")) +
  geom_line(aes(y = I, colour = "I")) +
  geom_line(aes(y = R, colour = "R"))
```



However, the element `geom_line` is repeated four times, which does not look nice. We can change it with little effort, and also define some better aesthetics:

```
modelOutput <- as.data.frame(modelOutput) %>%
  gather(key = compartment, value = value, -time) %>%
  left_join(data.frame(compartment = c("S", "E", "I", "R"),
                       ordered = 1:4,
                       stringsAsFactors = F), by = 'compartment')
ggplot(modelOutput, aes(x = time, y = value,
                        colour = reorder(compartment, ordered))) +
  geom_line() +
  viridis::scale_colour_viridis(discrete = T) +
  theme_bw() +
  theme(panel.grid.major.y = element_line(colour = "#dfdfdf")) +
  labs(colour = 'Compartment',
       x = 'Time',
       y = 'Population size',
       title = 'SERI epidemiological model')
```



You might wonder why we added additional column to `modelOutput` data frame. The thing is that `ggplot` automatically maps some aesthetics to groups using alphabetical order. Created column in which we store numbers that refer to compartment order in *SERI* acronym. Than we use it as argument to `reorder` function inside `ggplot`, which under the hood transforms variable to *factor* and change *level* orders. Using this approach, we don't need to change all the orderings of variables and aesthetics manually. I also used color palette from *`viridis` package, which contains four nice looking color palettes, which are colorblind safe (at least in theory).

Chapter 6

Functions

In short, function is a piece of code that takes some arguments, makes *mumbo jumbo* and returns result. All the time, through this book we were using functions that are built in **base R** or comes from additional packages (like `dplyr`). So you are now quite familiar whit the syntax that resembles typical mathematical syntax – *name of a function followed by arguments in brackets - like f(x).* From time to time you will need to do something in your code for few times with different arguments. In order to not repeat yourself and not *copy – paste* your code multiple times you can wrap your procedure in a function. The best way (as always) to understand how to do it, is to do it. To begin with something simple we will start with making basic math functions which later will lead us to simple calculator.

6.1 Simple math functions.

Simple math operations include: *adding*, *substracting*, *dividing* and *multipling.* To make our calculator slightly less boring, we can also add *powers* and *nth rooting.* To not to complicate to much things in the beginning, lets say that we want our function to take two and only two arguments. Lets look below how to code our functions:

```
addF <- function(x,y) {
  return(x+y)
}
subF <- function(x,y) {
  return(x-y)
}
divF <- function(x,y) {
  return(x/y)
}
mulF <- function(x,y) {
  return(x*y)
}
powF <- function(x,y) {
  return(x:y)
}
ntrF <- function(x,y) {
  return(x^(1/y))
}
```

6.2 Building your own calculator

Above example is of course useless and boring. So lets quickly get to making calculator, that would use one of above functions, or return results of all of them. So actually this time we will make so called *wraper* around previously made functions. We will use *switch* functionality, so besides our two parameters: `x` and `y` we will also tell our function which of the results we are interested in.

```r
simpleCalculator <- function(x, y, mathType) {
  addF <- function(x, y) {
  return(x+y)
  }
  subF <- function(x, y) {
  return(x-y)
  }
  divF <- function(x, y) {
  return(x/y)
  }
  mulF <- function(x, y) {
  return(x*y)
  }
  powF <- function(x, y) {
  return(x^y)
  }
  ntrF <- function(x, y) {
  return(x^(1/y))
  }
  switch(mathType,
         add = addF(x, y),
         substract = subF(x, y),
         multiple = mulF(x, y),
         divide = divF(x, y),
         power = powF(x, y),
         root = ntrF(x, y),
         all = cat('The result of mathematical opereators on two numbers:',
                      paste(x, 'and', y), 'are:', '\naddition:', addF(x, y),
                      '\nsubstraction:', subF(x, y), '\nmultiplication:', mulF(x, y),
                      '\ndvision:', divF(x, y), '\nWhat is more the', y,
                      'th power of', x, 'is', powF(x, y), 'and', x, y,
                      'th root is', ntrF(x, y)))
}

simpleCalculator(25,5, 'root')
```

```
## [1] 1.903654
```

```r
simpleCalculator(16,2,'all')
```

```
## The result of mathematical opereators on two numbers: 16 and 2 are:
## addition: 18
## substraction: 14
## multiplication: 32
## dvision: 8
## What is more the 2 th power of 16 is 256 and 16 2 th root is 4
```

6.3 It is not over yet... *Calculator shouldn't divide be 0*!

We know that division by 0 is not the best idea in the world, thus we should stop users (or ourselves) from doing it. Thus we will add an `if...else` statement to our function. Next time when you use 0 as a second argument you will see an error. Also, we declare `mathType = 'all'` as a default value, so if we omit this parameter, function will evaluate anyway.

```
simpleCalculator <- function(x, y, mathType = 'all') {
  if (mathType == 'divide' & y == 0) {
    return('You cannot divide by 0, please change y value.')
  } else if (mathType == 'root' & y == 0) {
    return('Root denominator is 0, cannot perform operation, please change y value.')
  } else if (mathType == 'all' & y == 0) {
    return('Y value needs to be different from 0 to make division and nth root.')
  }
  addF <- function(x, y) {
  return(x+y)
  }
  subF <- function(x, y) {
  return(x-y)
  }
  divF <- function(x, y) {
  return(x/y)
  }
  mulF <- function(x, y) {
  return(x*y)
  }
  powF <- function(x, y) {
  return(x^y)
  }
  ntrF <- function(x, y) {
  return(x^(1/y))
  }
  switch(mathType,
         add = addF(x, y),
         substract = subF(x, y),
         multiple = mulF(x, y),
         divide = divF(x, y),
         power = powF(x, y),
         root = ntrF(x, y),
         all = cat('The result of mathematical opereators on two numbers:',
                      paste(x, 'and', y), 'are:', '\naddition:', addF(x, y),
                      '\nsubstraction:', subF(x, y), '\nmultiplication:', mulF(x, y),
                      '\ndvision:', divF(x, y), '\nWhat is more the', y,
                      'th power of', x, 'is', powF(x, y), 'and', x, y,
                      'th root is', ntrF(x, y)))

}
simpleCalculator(25,0)
```

```
## [1] "Y value needs to be different from 0 to make division and nth root."
```

```
simpleCalculator(25,0, 'root')
```

```
## [1] "Root denominator is 0, cannot perform operation, please change y value."
```

```
simpleCalculator(25,0, 'divide')
```

```
## [1] "You cannot divide by 0, please change y value."
```

```
simpleCalculator(25,5)
```

```
## The result of mathematical opereators on two numbers: 25 and 5 are:
## addition: 30
## substraction: 20
## multiplication: 125
## dvision: 5
## What is more the 5 th power of 25 is 9765625 and 25 5 th root is 1.903654
```

Chapter 7

Graphics

What would be our work value without visualization? Not much. **R** provides use with some tools to make plots, charts and other visual stuff. However base version has very limited graphic design by default and making it pretty needs a lot of time and code. Nowadays, however, in `tidyverse` there is a very powerful library with dozens of extensions - `ggplot2`. *GG* stands for *Gramar of Graphics* and in practice it means that we build our visualization layer after layer. The vast spectrum of `ggplot` functions and its extensions is out of the scope of this book. Here we will focus on the basic and most useful things as well as how to find not so common functionalities.

7.1 Base plots

Before we get to `ggplot2`, we should begin with base graphic functions. It is true that the do not look as pretty as plots from dedicated libraries, but they are extremely useful in quick checking of our workflow. Not to mention, that many packages are still operating with old fashioned base graphics.

7.2 How to make plot?

In previous chapters we made some plots, but we did not cover the details. Most important thing is to acknowledge that we can make plots with `hist()`, `plot()`, `barplot()` and `boxplot()` functions (to mention only the most common). There are of course other types of plots you might need, however those three are the *classic* ones. If you are looking for more examples of base plots, you should visit help page of `graphics` package. To change the look of your plot, just provide proper arguments to this function, like `col` for color of line/points or, `xlab` and `ylab` to change names of axis labels. Unfortunately a list of arguments that you can control and adjust is long. The good news is that all of them have their default value, so you need to change only the ones you want, without thinking about rest. Full list of parameter you can change you will find after executing: `?par` command. As this help page contains many different kinds of parameters you should scroll to find *Graphical Parameters* chapter.

7.3 Can we actually do something?

Yes we can. But as I said earlier scope of this book is not graphics. Thus I will cover here only the basic things you should know, or at least know where to find them. Bellow, I will present only code and its output (Fig. 7.1). After what you read, you should be perfectly fine with figuring out what is going on. You can

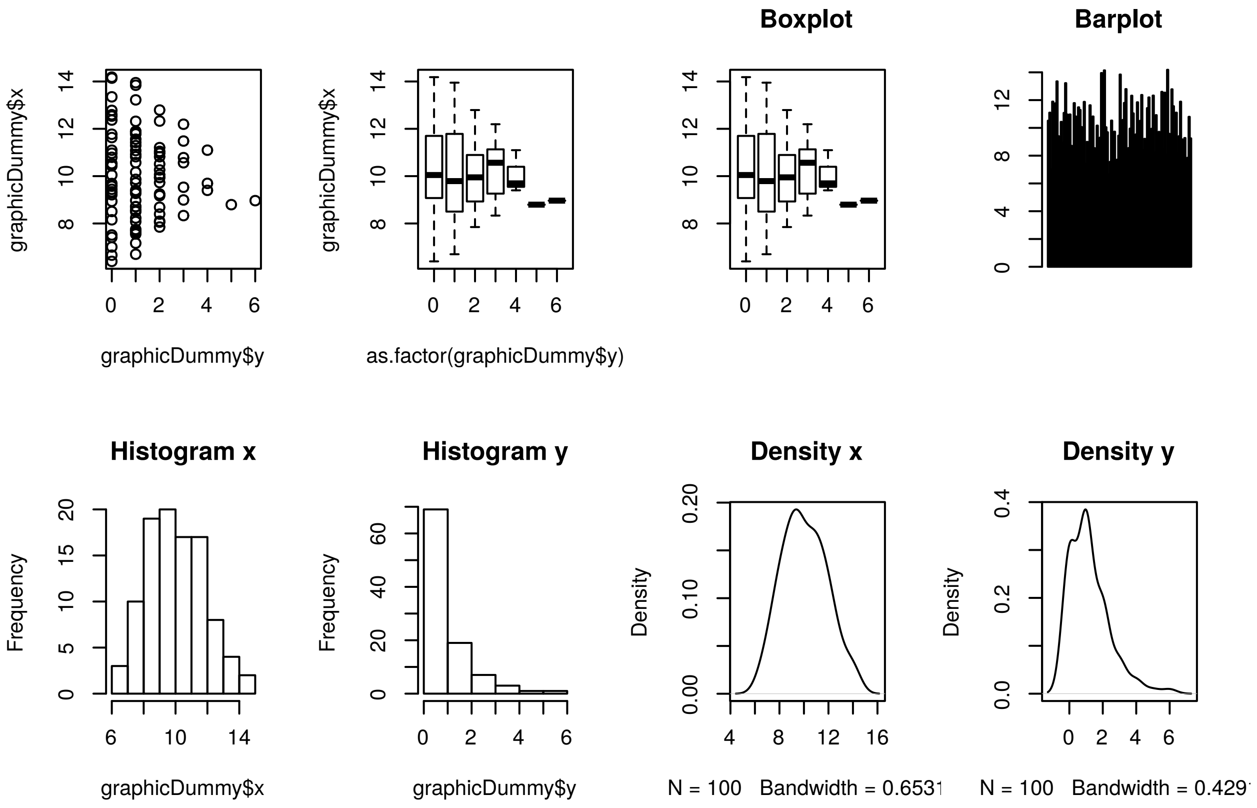


Figure 7.1: Base plots and histograms

also copy - paste the code in your script or console, edit it and learn how it works by doing. Even the best book is just a book, and nothing will substitute the knowledge you gain by experiencing.

```
graphicDummy <- data.frame(x = rnorm(100, 10, 2), y = rpois(100, 1.2))
par(mfrow = c(2,4))
plot(graphicDummy$x~graphicDummy$y)
plot(graphicDummy$x~as.factor(graphicDummy$y))
boxplot(graphicDummy$x~graphicDummy$y, main = 'Boxplot')
barplot(graphicDummy$x, main = 'Barplot')
hist(graphicDummy$x, main = 'Histogram x')
hist(graphicDummy$y, main = 'Histogram y')
plot(density(graphicDummy$x), main = 'Density x')
plot(density(graphicDummy$y), main = 'Density y')
```

7.4 ggplot2 - Graphics taken into other dimension

This package use different approach to make plots and charts (however it is not limited to only those). The philosophy it follows is called Grammar of Graphics (Wilkinson, 2005; Wickham, 2010). Within this workframe, we build whole plot (chart, etc.) by adding layer after layer of visual options. *With great power comes great responsibility* – ggplot2 as well as it's extensions are highly customisable, thus if you want to use non-default settings it may take some time before you figure out how to do it. Here we will briefly cover the most basic stuff, but there is very good book that goes deeper into manipulation of visual effects, with lots of examples – R Graphics Cookbook (Chang, 2013) which is also available on line.

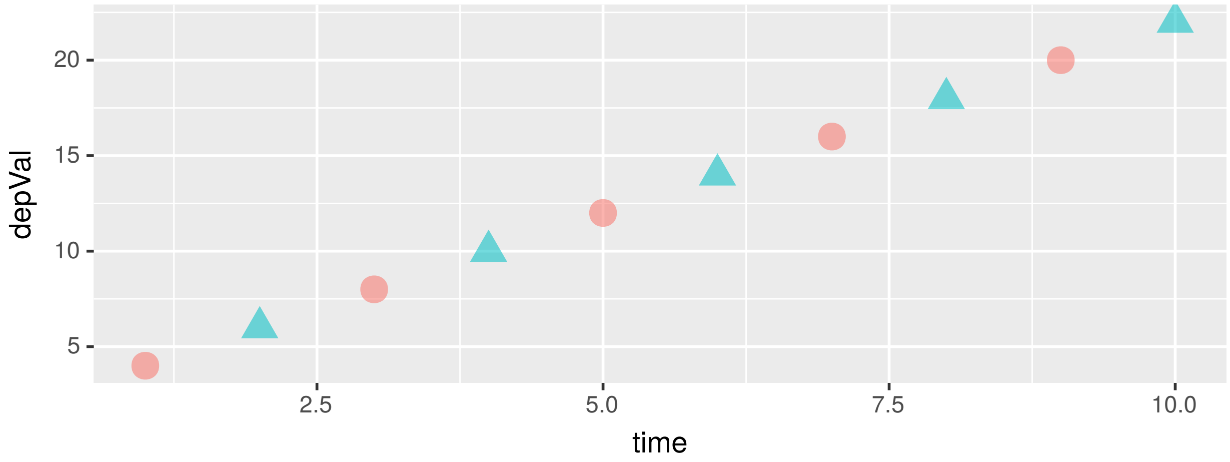


Figure 7.2: Basic scatterplot

7.4.1 Basic plot in ggplot2

Making basic scatterplot (Fig. 7.2) requires defining 2 layers:

1. Data
2. Type of plot/chart

```
plotTable <- data.frame(time = 1:10,
                        depVal = c((1:10)*2)+2,
                        typeDep = rep(c('first', 'second'), times = 5))
basePlot <- ggplot(plotTable, aes(x = time, y = depVal)) +
  geom_point(aes(colour = typeDep, alpha = 0.8, shape = typeDep, size = 3.5),
             show.legend = FALSE)
basePlot
```

In the function call (1st layer), we define data set (`plotTable`) and aesthetics (`x` and `y` values). In following layer, we define 3 aesthetics – `colour`, `alpha`, `shape` of points and their `size`. We also tell `ggplot` to not to make legend. Unfortunately, when you first start working with `ggplot` (and even when you have some experience), you will often run into small problems that will need some adjustments in your code. Hopefully, this package is so broadly used that you will easily find a lot of solution of your problems on *Stack Overflow*, as well as many extensions with predefined functions to make sophisticated graphics.

Lets take a little deeper look into layers and aesthetics. Default look of plot is little bit boring, and usually unsuitable for publications. In code below we will use another layer – `theme` – to define plot background, plot name and axis names (Fig. 7.3). Of course, this still is unsuitable for publication, however it looks more fancy and can be used in presentation, poster or on blog.

```
basePlot +
  labs(title = 'Basic dot plot', x = 'Time', y = 'Dependent variable') +
  theme(axis.title.x     = element_text(size = rel(0.7), colour = 'red'),
        axis.title.y     = element_text(size = rel(1.1)),
        plot.background  = element_rect(fill = 'yellow'),
        panel.background = element_rect(fill = 'pink'),
        panel.grid       = element_blank(),
        title            = element_text(size = rel(1.3),
                                        colour = 'gray50',
                                        face = 'bold'))
```

Beautiful!

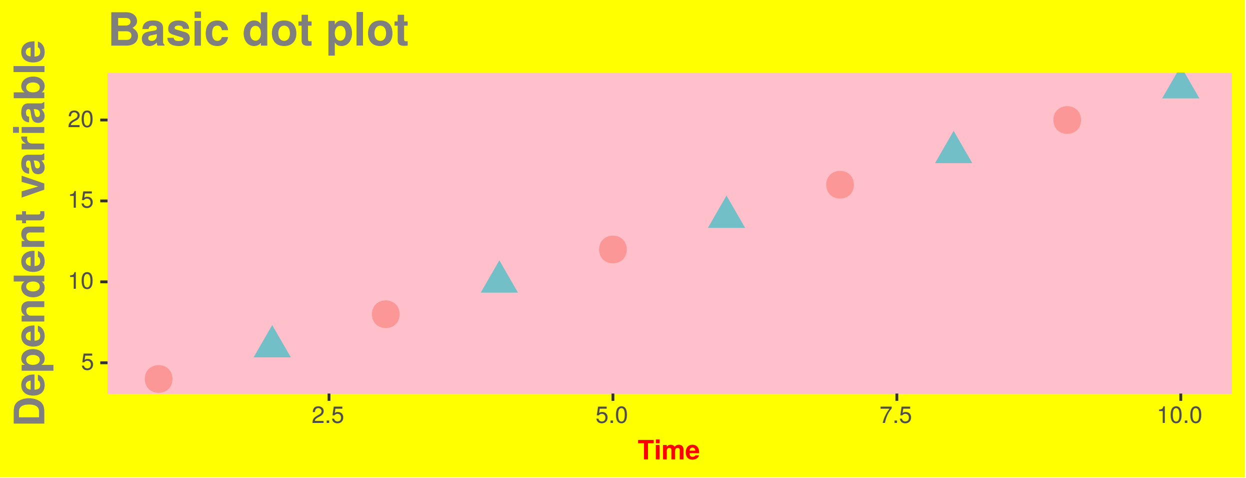


Figure 7.3: Scatterplot with additional aesthetics

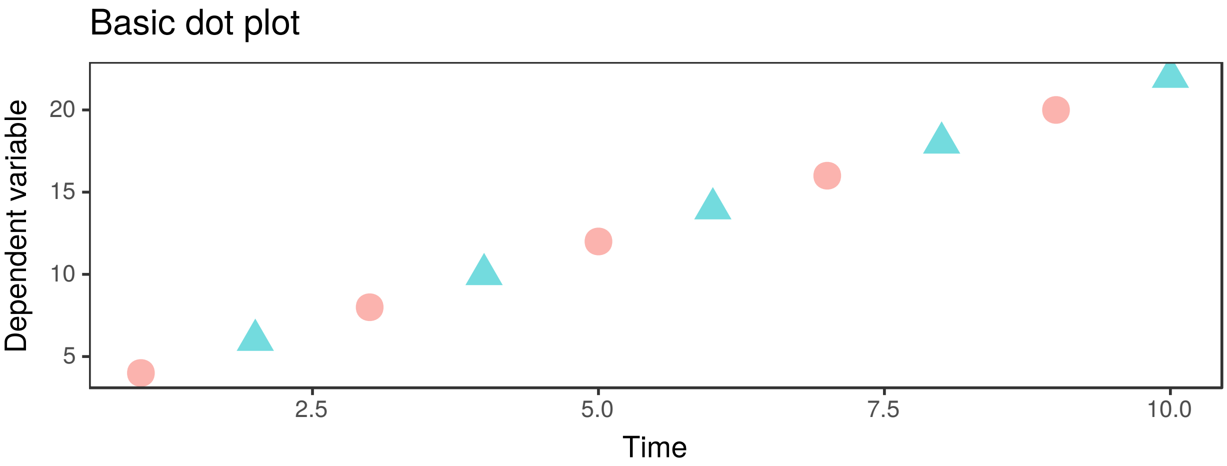


Figure 7.4: Scatterplot formated for journals

However we can use simple trick to make our plot in format that is actually supported by scientific journals (Fig. 7.4), i.e. white background, no gridlines etc.:

```
basePlot +
  labs(title = 'Basic dot plot', x = 'Time', y = 'Dependent variable') +
  theme_bw() +
  theme(panel.grid = element_blank())
```

7.4.2 Density plots

But what if we want to actually plot something useful? Lets say that we want to graphically compare distributions of two data sets. We can do it by histograms (boring) or by plotting density functions. First we should generate some data. Assume that you are tossing coin, but you have this strange gut feeling that something is not right. In 500 trials you had 218 tails and 282 heads. Fig. 7.5 shows simple density plot made with `ggplot`

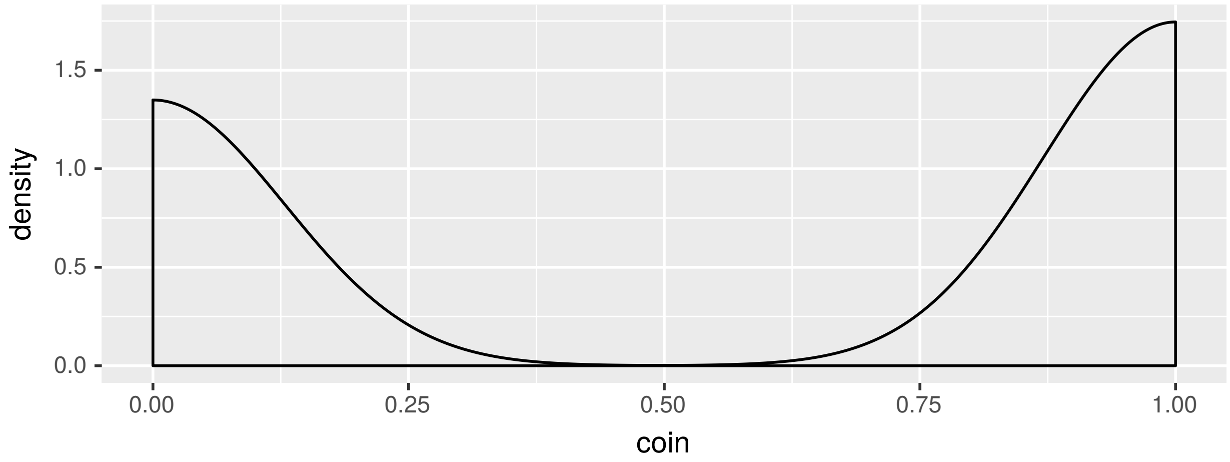


Figure 7.5: Simple density plot for coin tossing

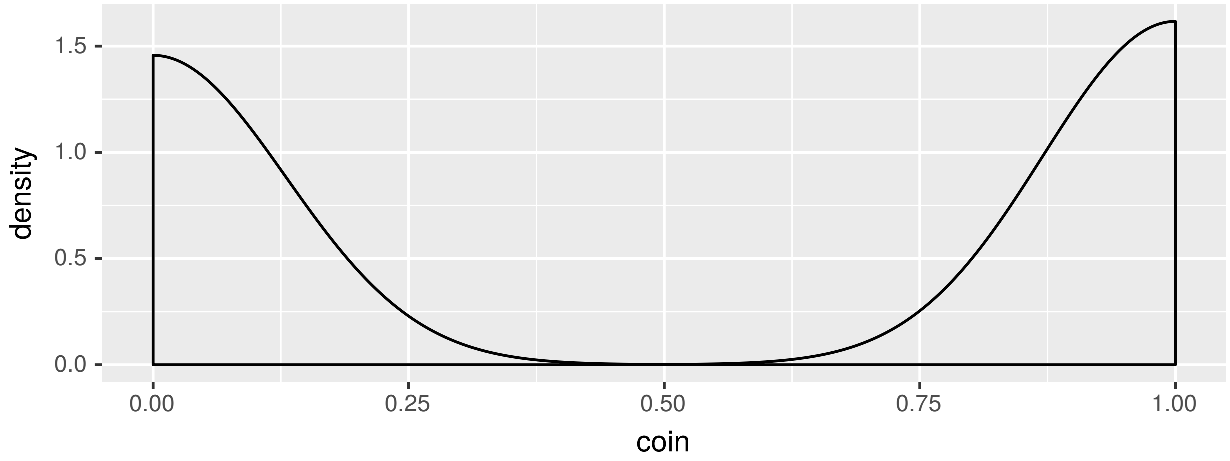


Figure 7.6: Density plot for binomial distribution

```
coinDensity <- data.frame(coin = c(rep(1, times = 282), rep(0, times = 218)))
ggplot(coinDensity, aes(x = coin)) + geom_density()
```

And we want to check if it is really different from binomial distribution (Fig. 7.6).

```
tCoinDensity <- data.frame(coin = rbinom(500, 1, 0.5))
ggplot(tCoinDensity, aes(x = coin)) + geom_density()
```

Now we have two plots which is not the best way to visually check two distributions. It would be much better if we plot them one by one, preferably with different colors and opacity (like in Fig. 7.7). Below it's short code snippet how to do it in **R**.

```
coinTest <- bind_rows(coinDensity, tCoinDensity) %>%
  bind_cols(distribution = rep(c('empirical', 'theoretical'), each = 500))
ggplot(coinTest, aes(x = coin, fill = distribution)) +
  geom_density(alpha = 0.7, colour = 'yellow') +
  scale_fill_manual(values = c('blue', 'violet'))
```

Indeed, the coin is slightly biased towards heads.

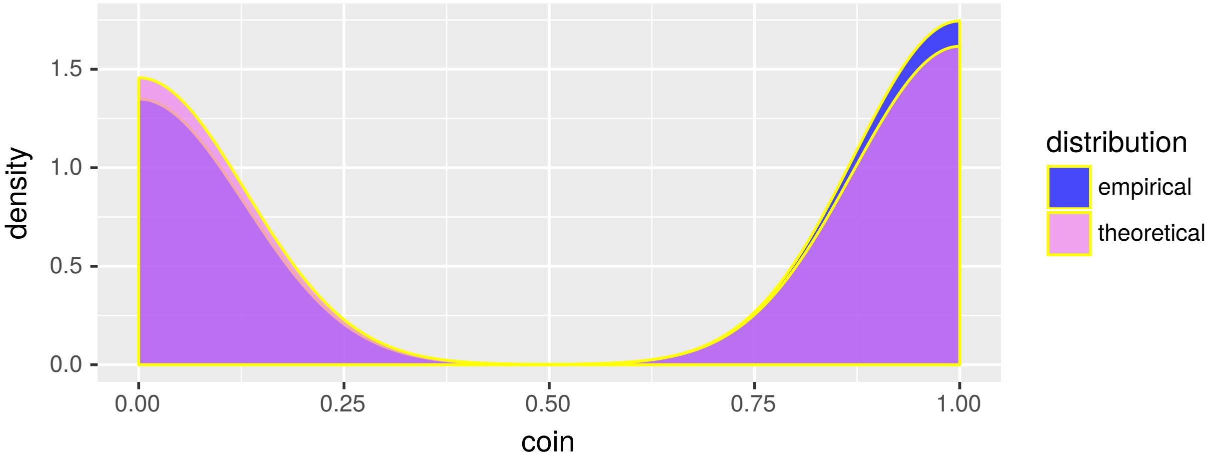


Figure 7.7: Combined density plots

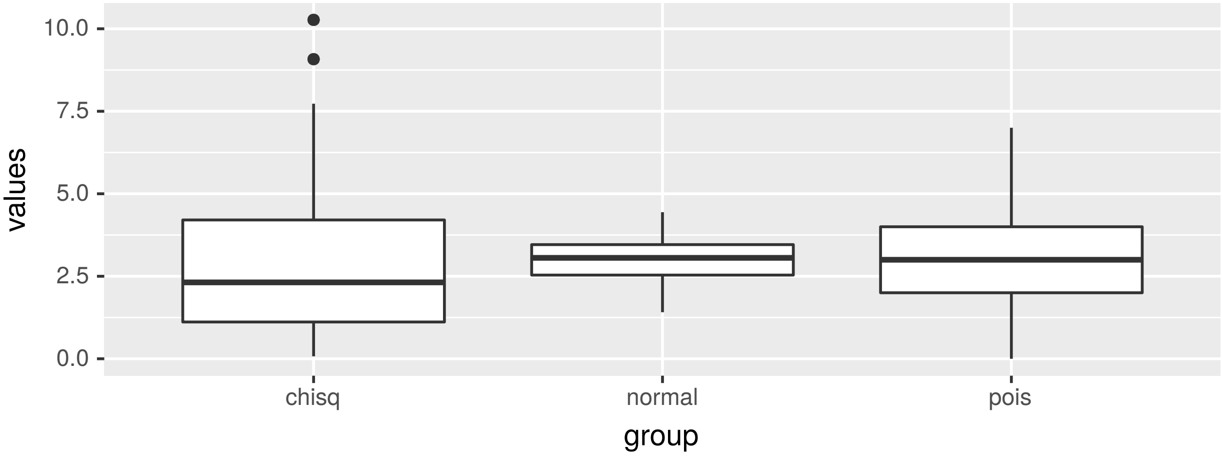


Figure 7.8: Box-plot of three groups

7.4.3 Box-plot and bar-plot

When dealing with comparison of two or more groups we usually check their distribution by visually inspecting so called *box plots* or *box-whiskers plots*. This plot shows interquartile range (IQR) as box, median as a line inside box, 1.5 IQR as *whiskers* and all of the data points that are more than 1st quartile - 1.5 IQR or 3rd quartile + 1.5 IRQ as separated dots called *outliers*. There are few different approaches to define *outlieres* and *whiskers* in box plots, nonetheless above is most common. It is very straightforward to make box-plots (Fig. 7.8) with `ggplot2`:

```
boxGroups <- data.frame(group = rep(c('normal', 'chisq', 'pois'), each = 100),
                        values = c(rnorm(100, 3, 0.65), rchisq(100, 3), rpois(100, 3)))
ggplot(boxGroups, aes(x = group, y = values)) +
  geom_boxplot()
```

Bar-plots are mainly used to show categorical variables as rectangles with height (length) that is proportional to highest value that each categorical variable represents. `ggplot` syntax sometimes is tricky. Unfortunately, that is the case when dealing with bar plots. Consider below example and try to figure out what happened on Fig. 7.9:

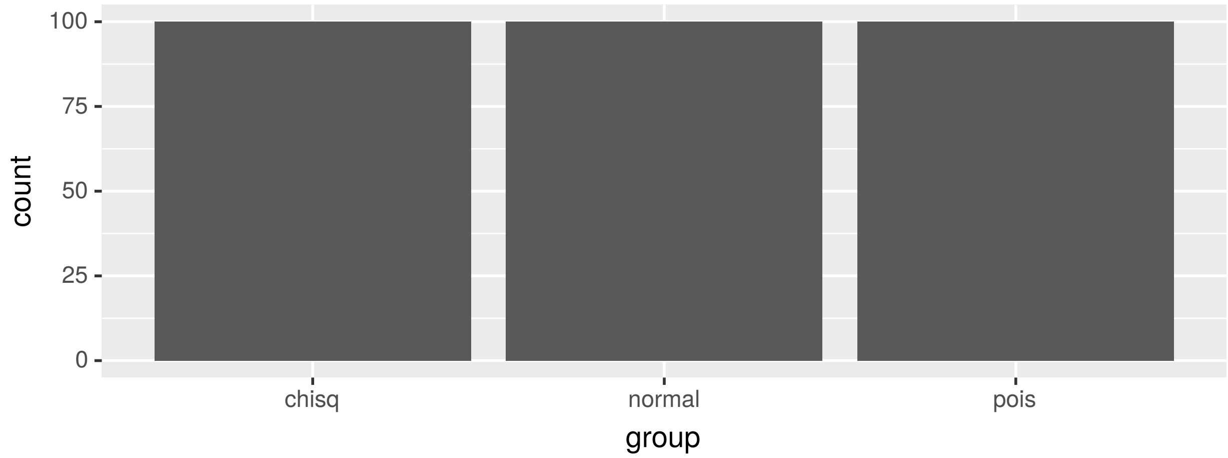


Figure 7.9: Mysterious bar-plot

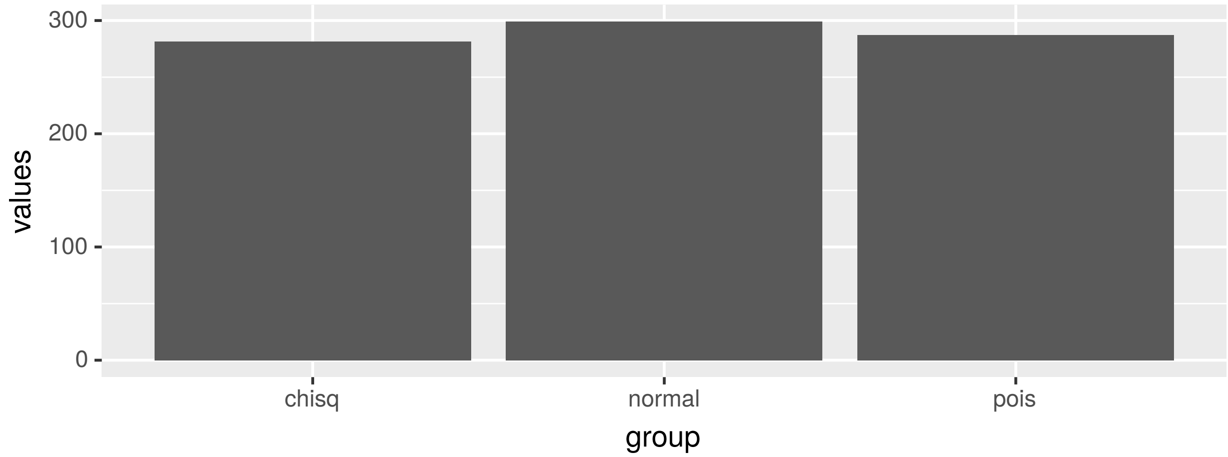


Figure 7.10: Sums by group bar-plot

```
barGroups <- boxGroups
ggplot(barGroups, aes(x = group)) +
  geom_bar(stat = 'count')
```

Instead of having bars that length represents highest value for each group, we created bar plot where length of bar is proportional to number of records in all group, i.e. 100. Depending on what value we are interested, we can take different approaches to solve this problem. First, instead of using `geom_bar` layer we can use `geom_col`. It will allow us to represent sum of values in particular group (Fig. 7.10).

```
brGroups <- boxGroups
ggplot(brGroups, aes(x = group, y = values)) +
  geom_col()
```

In the second approach, we first need to transform our data, so for each group and we calculate a value that will be represented as length of bar. This method gives us a lot of flexibility. We can calculate number of cases, mean, standard deviation, maximum value, while changing only one command and update plot to show different measures. Below you find code how to plot maximum values for each group on bar-plot (Fig. 7.11).

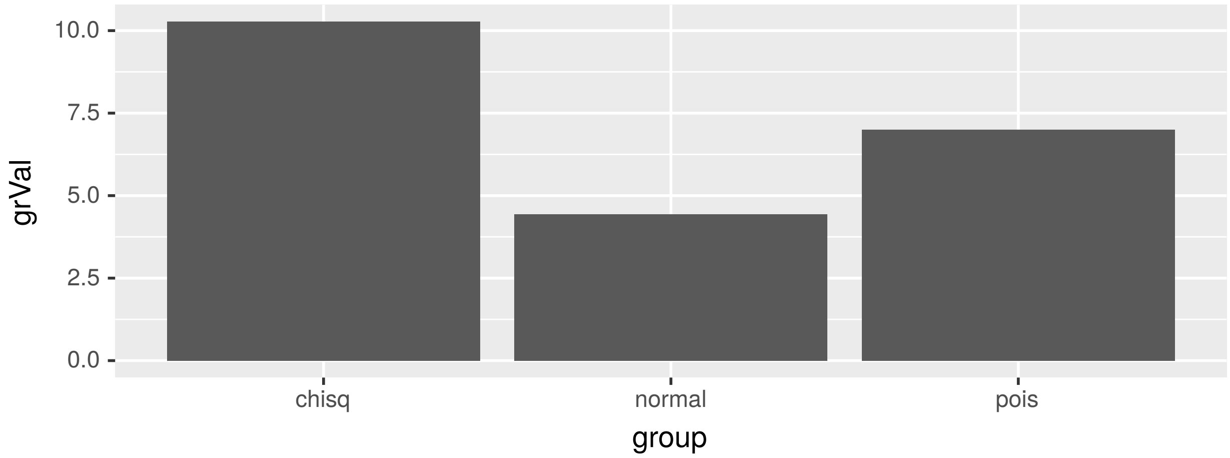


Figure 7.11: Maximum value by group bar-plot

```r
barGroups <- boxGroups %>% group_by(group) %>% summarise(grVal = max(values))
ggplot(barGroups, aes(x = group, y = grVal)) +
  geom_bar(stat = 'identity')
```

7.4.4 Regression

Last common visualization is regression plot. With layer by layer approach we can control all the elements of the plot. Below code shows how you can visualize both dots and lines for regression, change title and labels and set some non standard appearance of plot (Fig. 7.12).

```r
regVar <- c(sample(1:20, 100, replace = T))
regRes <- ((regVar * 2.6) + 0.8) + order(rnorm(100, 10, 4.25))
regDF <- data.frame(x = regVar, y = regRes)
ggplot(regDF, aes(x, y)) +
  geom_point(fill = 'yellow', colour = '#ff0099', shape = 21, stroke = 1.15) +
  geom_ribbon(stat='smooth', method = "lm", se = TRUE,
              fill = '#990011', colour = 'yellow', alpha = 0.4,
              linetype = 3, size = 0.2) +
  geom_line(stat='smooth', method = "lm",
            colour = 'green', alpha = 1,
            linetype = 4, size = 1.15) +
  theme(panel.background = element_rect(fill = NA),
        axis.line = element_line(size = 1.15, colour = 'grey')) +
  labs(x = 'Values of X',
       y = 'y = 2.6 * x + 0.8',
       title = 'Example of regression plot') +
  annotate('text', label = 'y==2.6*x+0.8+epsilon', parse = T,
           x = 3.5, y = 135, colour = 'blue')
```

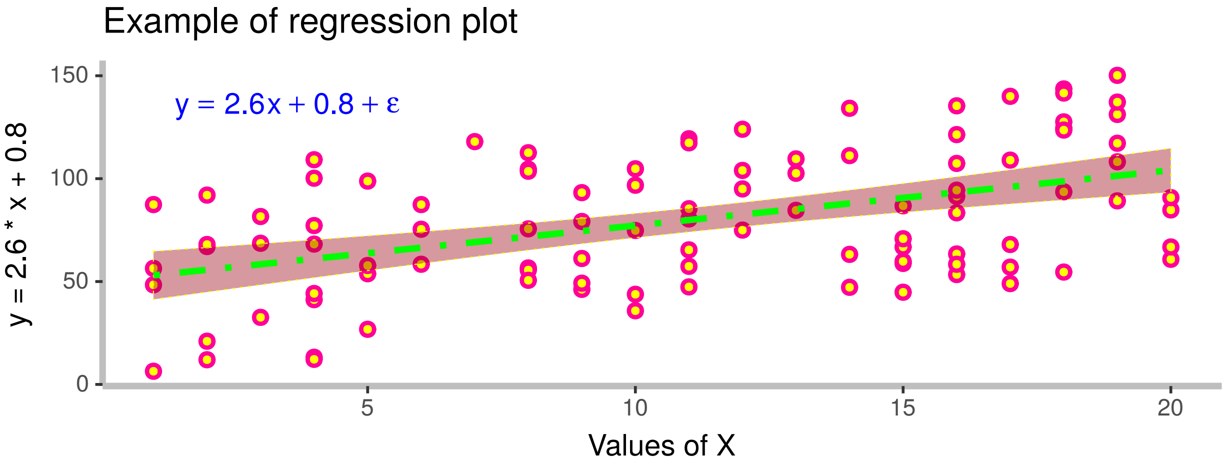


Figure 7.12: Formated Regression plot with 95% CI

Chapter 8

Simple workflow

8.1 Data

We manage to go through whole introduction to *R's Galaxy*. But that's not the end. For a final word I would like to go with you with once again through some of the stuff you will need more or less in your everyday job. We will use **R** to see, how we can predict some values in basic microbiology.

First we need a data set. I downloaded data from deposited in ComBase for *Bacillus cereus* in broth with conditions as follows: pH = 6.1, water activity (aw) = 0.974, sodium chloride in environment = 4.5%, temperatures = 15, 16, 20, 22, 30 degrees Celsius. This data comes from Food Standards Agency, UK, and is hardcoded (so you can copy-paste) in the end of this chapter.

8.2 Exploaratory visualisation and setup

We will start with visually inspecting this data set.

```
ggplot(BcDF, aes(x = Time, y = LogC)) +
  geom_point(aes(colour = resID)) +
  facet_wrap(~Temp)
```

Unfortunately, one plot on Fig. 8.1 (30 degrees Celsius) is bit compressed, so lets isolate it and look closer on the points (Fig. 8.2).

```
BcDFsub39 <- subset(BcDF, Temp == 30)
ggplot(BcDFsub39, aes(x = Time, y = LogC)) +
  geom_point()
```

It seems that there is at least short lag phase in each of the growth temperatures. We will use `nlsMicrobio` package, which contains some models, that we will use to estimate μ_{max}. First we will split, whole *data frame* into *list* of *data frames* – one for each temperature. Than we will use our good friend `lapply()` to change names of columns in all data sets so it fits needs of our function.

```
BcList <- split(BcDF, f= BcDF$resID)
BcList <- lapply(BcList, setNames, nm = c('t',  'LOG10N', 'Temp', 'resID'))
```

From plots above (i.e., Fig. 8.1 and Fig. 8.2) we make an assumption that all of those growth curves have shorter or longer lag phase.

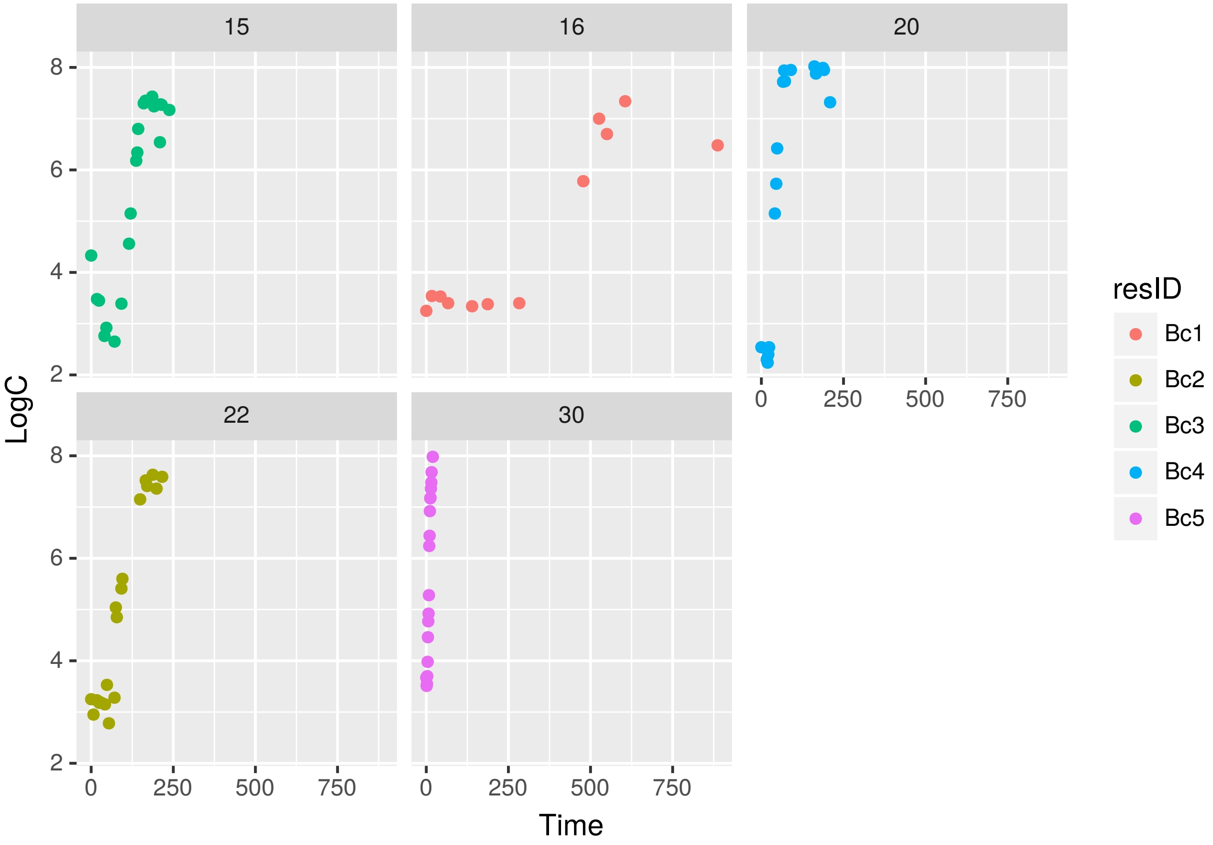


Figure 8.1: Growths in temperature range

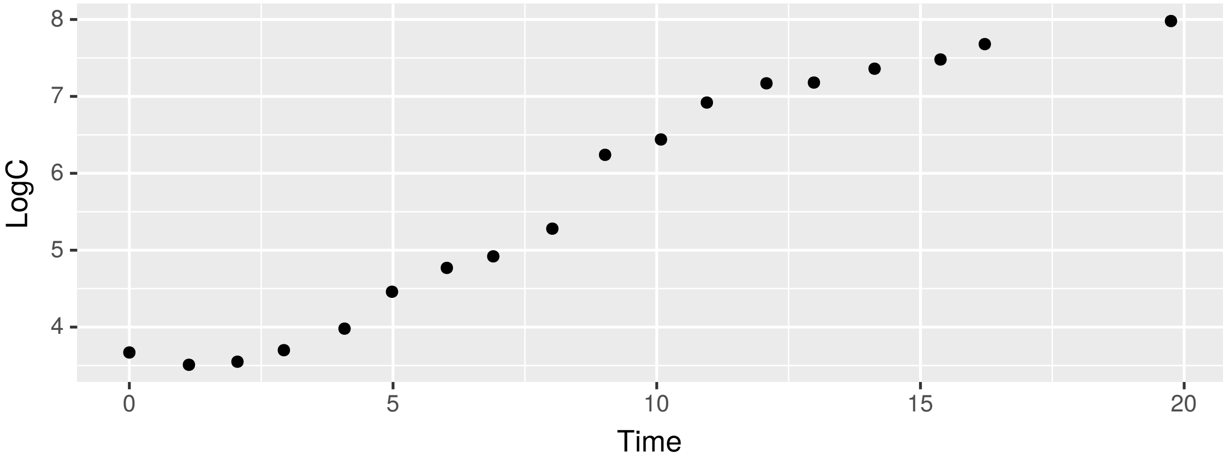


Figure 8.2: Growth in 30 deg. C

8.3 Growth models

It would be nice to use trilinear (Buchanan) and Baranyi model to estimate μ_{max}. However writing it by hand is error prone, we can use predefined models from `nlsMicrobio` package (and that is why we need precise names of columns). Lets see how models look:

```
baranyi
```

```
## LOG10N ~ LOG10Nmax + log10((-1 + exp(mumax * lag) + exp(mumax *
##     t))/(exp(mumax * t) - 1 + exp(mumax * lag) * 10^(LOG10Nmax -
##     LOG10N0)))
## <environment: namespace:nlsMicrobio>
```

```
buchanan
```

```
## LOG10N ~ LOG10N0 + (t >= lag) * (t <= (lag + (LOG10Nmax - LOG10N0) *
##     log(10)/mumax)) * mumax * (t - lag)/log(10) + (t >= lag) *
##     (t > (lag + (LOG10Nmax - LOG10N0) * log(10)/mumax)) * (LOG10Nmax -
##     LOG10N0)
## <environment: namespace:nlsMicrobio>
```

To estimate *trilinear* model we could also use `segmented` package, however than we would use different approach to solve the problem – we would estimate *breakpoints* or *knots* and than fit linear regression for each segment. The problem with this approach is that because of data in *lag* and *stationary* phase, with this method we would obtain slope values $\neq 0$ for those phases.

Unfortunately, when using *non-linear* regression algorithms we sometimes run into trouble and cannot really estimate values we are interested in. Sometimes we need to make small changes in initial values that we parse as function arguments. In other times, **R** is unable to make estimations due to data structure. The main problem is that there is no *one size fits all* solution, and often finding best way to solve a problem is an iterative process of manually finding best starting parameters or using multiple functions to find the working one. Below we attempt to find μ_{max} with *Baranyi* and *Buchanan* approach.

```
bar1.nls <- nls(baranyi, BcList$Bc1,
                list(lag = 200, mumax = 0.03, LOG10N0 = 3, LOG10Nmax = 6.5),
                control = nls.control(maxiter = 500))
```

```
## Error in nls(baranyi, BcList$Bc1, list(lag = 200, mumax = 0.03, LOG10N0 = 3, : step factor 0.000488281 redu
```

```
bar2.nls <- nls(baranyi, BcList$Bc2,
                list(lag = 2, mumax = 0.5, LOG10N0 = 2, LOG10Nmax = 8))
```

```
## Warning in numericDeriv(form[[3L]], names(ind), env): NaNs produced
```

```
## Error in numericDeriv(form[[3L]], names(ind), env): Missing value or an infinity produced when evaluating
```

```
bar3.nls <- nls(baranyi, BcList$Bc3,
                list(lag = 5, mumax = 0.01, LOG10N0 = 3, LOG10Nmax = 6))
```

```
## Error in numericDeriv(form[[3L]], names(ind), env): Missing value or an infinity produced when evaluating
```

```
bar4.nls <- nls(baranyi, BcList$Bc4,
                list(lag = 1, mumax = 0.5, LOG10N0 = 2, LOG10Nmax = 8))
bar5.nls <- nls(baranyi, BcList$Bc5,
                list(lag = 1, mumax = 0.5, LOG10N0 = 3, LOG10Nmax = 8))
```

OK. That is a bit worrying. We were able to solve equations only for two data sets. One solution we can try is to use `minipack.lm` which uses different approach which often works better than standard `nls` function. We will also allow more iterations of algorithm, which is second common solution.

```
bar1.nlsLM <- nlsLM(baranyi, BcList$Bc1,
                    list(lag = 200, mumax = 0.03, LOG10N0 = 3, LOG10Nmax = 6.5),
                    control = nls.lm.control(maxiter = 500))
summary(bar1.nlsLM)
```

```
##
## Formula: LOG10N ~ LOG10Nmax + log10((-1 + exp(mumax * lag) + exp(mumax *
##     t))/(exp(mumax * t) - 1 + exp(mumax * lag) * 10^(LOG10Nmax -
##     LOG10N0)))
##
## Parameters:
##           Estimate Std. Error t value Pr(>|t|)
## lag       4.622e+02  1.121e+06    0.00        1
## mumax     3.515e-01  2.497e+04    0.00        1
## LOG10N0   3.406e+00  9.269e-02   36.74 3.30e-10 ***
## LOG10Nmax 6.880e+00  1.416e-01   48.59 3.56e-11 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.2452 on 8 degrees of freedom
##
## Number of iterations to convergence: 41
## Achieved convergence tolerance: 1.49e-08
```

```
BaranyiBc1 <- summary(bar1.nlsLM)$parameters
```

Lets substitute variables in equation (later I'll show you haw to do it automatically):

`LOG10N ~ LOG10Nmax + log10((-1 + exp(mumax * lag) + exp(mumax * t))/(exp(mumax * t) - 1 + exp(mumax * lag) * 10^(LOG10Nmax - LOG10N0`

with calculated ones and see how predicted curve fits our data (Fig. 8.3).

```
xBarBc1 <- 1:800
predBarBc1 <- 6.880 +
  log10((-1 + exp(0.3537 * 462.2242) +
           exp(0.3515 * xBarBc1))/
          (exp(0.3515 * xBarBc1) - 1 +
             exp(0.3515 * 462.2242) * 10^(6.880 - 3.406)))
```

```
plot(BcList$Bc1$t, BcList$Bc1$LOG10N)
lines(predBarBc1~xBarBc1)
```

This does not look like a good fit. Yet another solution to this problem is to use package `segmented`. It's not the most elegant solution to the problem, but hey, our data is not the best one as well.

```
segmented1.lm <- lm(LOG10N~1, data = BcList$Bc1) %>%
  segmented(seg.Z = ~t,
            psi = c(200, 400),
            control = seg.control(it.max = 500, n.boot = 300))
Segmented1 <- list(confint.segmented(segmented1.lm),slope(segmented1.lm))
```

> ** NOTE: You should rather use formula: lm(LOG10N~t, data = BcList$Bc1), however for unknown reasons when using `bookdown` build functions fails to calculate linear model. So, till I find a solution to this problem we'll stick to simplified one.

Now we make some quick plot (Fig. 8.4) to check how does the prediction fits data.

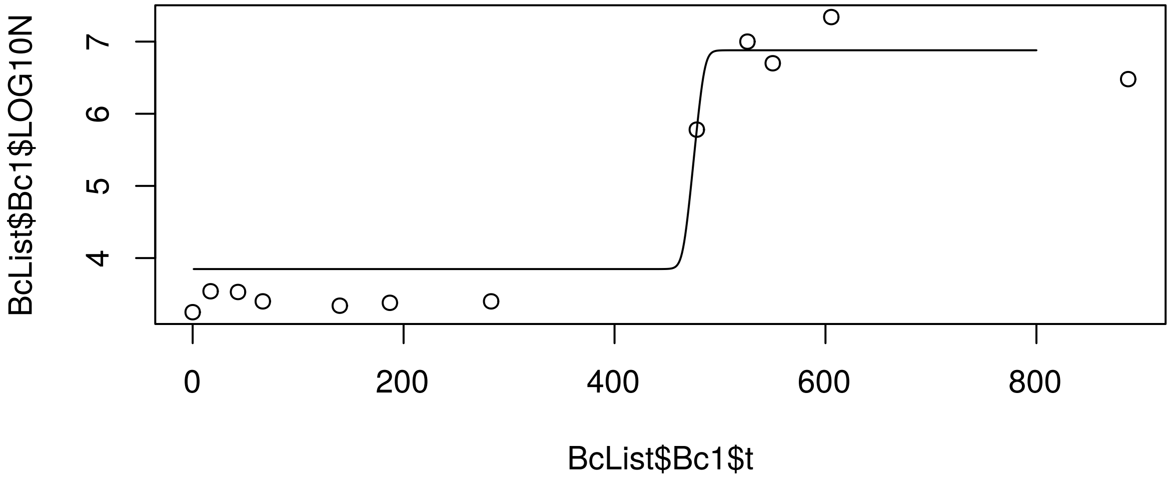


Figure 8.3: Predicted growth from Baranyi equation for Bc1 dataset

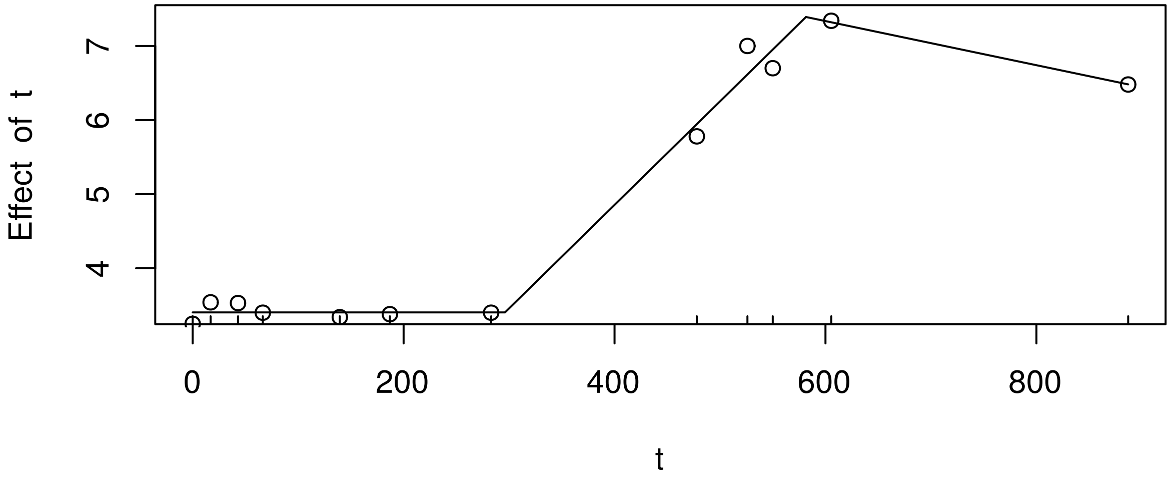


Figure 8.4: Predicted growth from segmented regression for Bc1 dataset

```
plot(segmented1.lm)
points(BcList$Bc1$LOG10N~BcList$Bc1$t)
```

As I stated before, the *lag* and *stationary* phase segments are not nice, however slope of the *growth* phase seems to fit slightly better than the one calculated with `nlsML()` and `baranyi`. Also we need to remember that the more data points exist in set, and the better they are spread among independent variable range, the easier it gets to make statistical procedures, and the results will be more robust.

We can also try to fit non linear regression for `buchanan` with `nlsML()` and `nls2` (the latter with brute force). Unfortunately, `nls` methods are quite sensitive on data and initial values, thus sometimes it takes a while to find something that makes sens or does not throw error now as `singular gradient matrix at initial parameter estimates`.

```
## Levenberg-Marquardt algorithm
buch1.nlsLM <- nlsLM(buchanan, BcList$Bc1,
                     list(lag = 200, mumax = 0.03,
                          LOG10N0 = 3, LOG10Nmax = 6.5),
                     control = nls.lm.control(maxiter = 500))
```

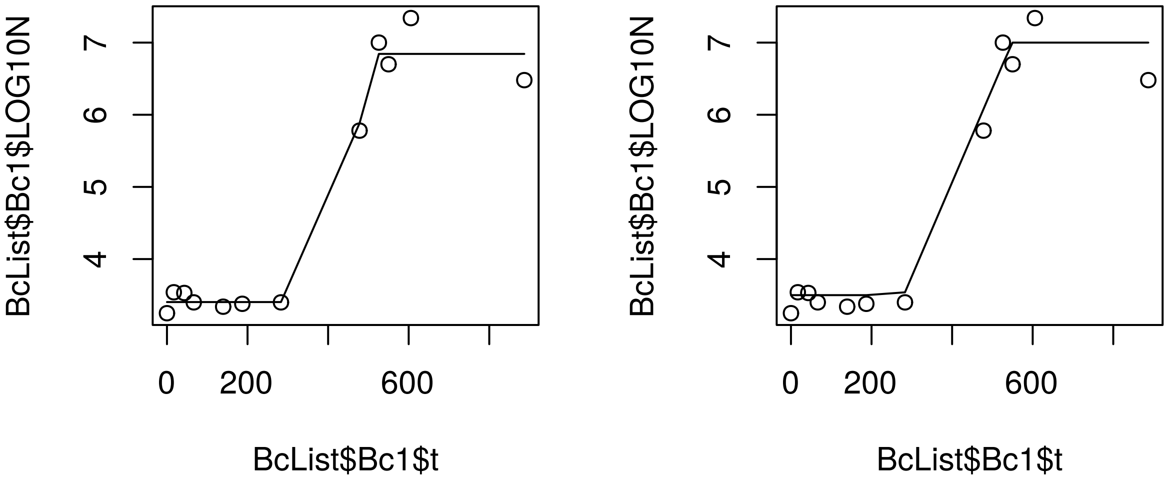


Figure 8.5: Comparission of fiting Buchanan equation with Levenberg-Marquardt and brute-force Gauss-Newton

```
## Gauss-Newton algorithm with brute-force
buch1.nlsBF <- nls2(buchanan, data = BcList$Bc1,
                    start = list(lag = 280, mumax = 0.03,
                                 LOG10N0 = 3.5, LOG10Nmax = 7),
                    control = nls.control(maxiter = 1000,
                                          tol = 1e-01, minFactor = 1/100),
                    algorithm = 'brute-force')
BuchananBFBc1<- summary(buch1.nlsLM)$parameters
BuchananBc1<- summary(buch1.nlsBF)$parameters
```

And quick plot (Fig. 8.5) for visual inspection:

```
par(mfrow = c(1,2))
plot(BcList$Bc1$t, BcList$Bc1$LOG10N)
lines(predict(buch1.nlsLM)~BcList$Bc1$t)

plot(BcList$Bc1$t, BcList$Bc1$LOG10N)
lines(predict(buch1.nlsBF)~BcList$Bc1$t)
```

```
par(mfrow = c(1,1))
```

Lets compare our parameters estimates:

```
Segmented1
```

```
## [[1]]
## [[1]]$t
##           Est. CI(95%).l CI(95%).u
## psi1.t 296.138   143.988   448.289
## psi2.t 581.507   531.232   631.782
##
##
## [[2]]
## [[2]]$t
##              Est.   St.Err. t value  CI(95%).l   CI(95%).u
## slope1  0.0000000        NA      NA         NA          NA
## slope2  0.0139720 0.0040006  3.4926  0.0045125  0.02343200
## slope3 -0.0029774 0.0010420 -2.8573 -0.0054413 -0.00051338
```

```
BaranyiBc1
```

```
##              Estimate   Std. Error      t value     Pr(>|t|)
## lag       462.2241887 1.120639e+06 4.124648e-04 9.996810e-01
## mumax       0.3515197 2.497031e+04 1.407751e-05 9.999891e-01
## LOG10N0     3.4057143 9.269492e-02 3.674111e+01 3.301879e-10
## LOG10Nmax   6.8800000 1.415938e-01 4.858969e+01 3.561034e-11
```

```
BuchananBc1
```

```
##            Estimate   Std. Error   t value     Pr(>|t|)
## lag          280.00 26.043862981 10.751093 4.930689e-06
## mumax          0.03  0.003968372  7.559775 6.547676e-05
## LOG10N0        3.50  0.128057657 27.331439 3.461481e-09
## LOG10Nmax      7.00  0.181100875 38.652491 2.205201e-10
```

```
BuchananBFBc1
```

```
##                Estimate  Std. Error   t value     Pr(>|t|)
## lag        354.25006466 55.09540195  6.429757 2.026158e-04
## mumax        0.04610793  0.01690905  2.726819 2.597327e-02
## LOG10N0      3.40424344  0.09420648 36.135981 3.768335e-10
## LOG10Nmax    6.84343632  0.14390277 47.555974 4.227236e-11
```

As you can see, all the methods differ in their estimates (for Baranyi it μ_{max} is expressed as natural logarithm, so we can use equation $log_{10}(x) = \frac{ln(x)}{ln(10)}$ to obtain value 0.1526631). Some of them work better some work worse. Sometimes they produce false convergence, sometimes they highly rely even on small changes in initial values and sometimes they just do not work with our data set. Other thing one should consider is very wise proverb:

> *Garbage in, garbage out.*

Meaning, that even with best methods, our data set might just not be sufficient to produce reliable output. Unfortunately in many cases we need to deal somehow with this kind of data. And we should always check if our output makes sense.

But lets try to find a solution to other Bc2 and Bc3 data sets. We'll simplify things and start estimation of parameters with Buchanan and Baranyi models using `nlsLM`

```
buch2.nlsLM <- nlsLM(buchanan, BcList$Bc2,
                     list(lag = 100, mumax = 0.03, LOG10N0 = 3, LOG10Nmax = 6.5),
                     control = nls.lm.control(maxiter = 500))
buch3.nlsLM <- nlsLM(buchanan, BcList$Bc3,
                     list(lag = 100, mumax = 0.03, LOG10N0 = 3, LOG10Nmax = 6.5),
                     control = nls.lm.control(maxiter = 500))
bar2.nlsLM <- nlsLM(baranyi, BcList$Bc2,
                 list(lag = 10, mumax = 0.2, LOG10N0 = 3, LOG10Nmax = 6.5),
                 control = nls.lm.control(maxiter = 500))
bar3.nlsLM <- nlsLM(baranyi, BcList$Bc3,
                 list(lag = 10, mumax = 0.03, LOG10N0 = 3, LOG10Nmax = 6.5),
                 control = nls.lm.control(maxiter = 500))
```

Hurray! No errors... for now at least. Now, we can use our variable that contains estimated parameters to make prediction. We will need new *data frame* containing values for which we want to make prediction. Because, we want also to plot predicted lines (to check how they fit to our data sets), we can predict values along sequence of numbers from 1 to 270. So, lets inspect visually how our models fit to data (Fig. 8.6).

```
newTime <- data.frame(t = 1:270)
predBuch2 <- predict(buch2.nlsLM, newdata = newTime)
predBuch3 <- predict(buch3.nlsLM, newdata = newTime)
predBar2 <- predict(bar2.nlsLM, newdata = newTime)
predBar3 <- predict(bar3.nlsLM, newdata = newTime)

#4 plots in one picture 2 x 2
par(mfrow = c(2,2))
plot(predBuch2 ~ newTime$t, type = 'l', col = 'red',
     main = 'Buchanan', sub = 'dataset Bc2', cex.main = 0.75, cex.sub = 0.6,
     xlab = 'time', ylab = 'Log10N', cex.lab = 0.6)
  points(BcList$Bc2$t, BcList$Bc2$LOG10N,
         col = 'blue', pch = 15)

plot(predBuch3 ~ newTime$t, type = 'l', col = 'red',
     main = 'Buchanan', sub = 'dataset Bc3', cex.main = 0.75, cex.sub = 0.6,
     xlab = 'time', ylab = 'Log10N', cex.lab = 0.6)
  points(BcList$Bc3$t, BcList$Bc3$LOG10N,
         col = 'blue', pch = 15)

plot(predBar2 ~ newTime$t, type = 'l', col = 'red',
     main = 'Baranyi', sub = 'dataset Bc2', cex.main = 0.75, cex.sub = 0.6,
     xlab = 'time', ylab = 'Log10N', cex.lab = 0.6)
  points(BcList$Bc2$t, BcList$Bc2$LOG10N,
         col = 'blue', pch = 15)

plot(predBar3 ~ newTime$t, type = 'l', col = 'red',
     main = 'Baranyi', sub = 'dataset Bc3', cex.main = 0.75, cex.sub = 0.6,
     xlab = 'time', ylab = 'Log10N', cex.lab = 0.6)
  points(BcList$Bc3$t, BcList$Bc3$LOG10N,
         col = 'blue', pch = 15)
```

Fit seems to be quite reasonable. Finally lets check the parameters of models:

```
summary(buch2.nlsLM)$parameters
```

```
##                Estimate Std. Error   t value     Pr(>|t|)
## lag         46.05276132 6.39164691  7.205148 2.096717e-06
## mumax        0.09676246 0.01095127  8.835730 1.493145e-07
## LOG10N0      3.16285715 0.14910019 21.212965 3.852824e-13
## LOG10Nmax    7.50200000 0.17641773 42.524072 6.902309e-18
```

```
summary(buch3.nlsLM)$parameters
```

```
##               Estimate Std. Error   t value     Pr(>|t|)
## lag         91.8457093 5.86309056 15.665067 3.979643e-11
## mumax        0.1501464 0.02126317  7.061337 2.691088e-06
## LOG10N0      3.2650000 0.16502452 19.784939 1.130406e-12
## LOG10Nmax    7.1975000 0.14291542 50.361954 4.696585e-19
```

```
summary(bar2.nlsLM)$parameters
```

```
##               Estimate Std. Error   t value     Pr(>|t|)
## lag         60.0785947 5.80981604 10.340877 1.719577e-08
## mumax        0.1727468 0.03662681  4.716403 2.328926e-04
## LOG10N0      3.1450278 0.12117868 25.953641 1.666670e-14
```

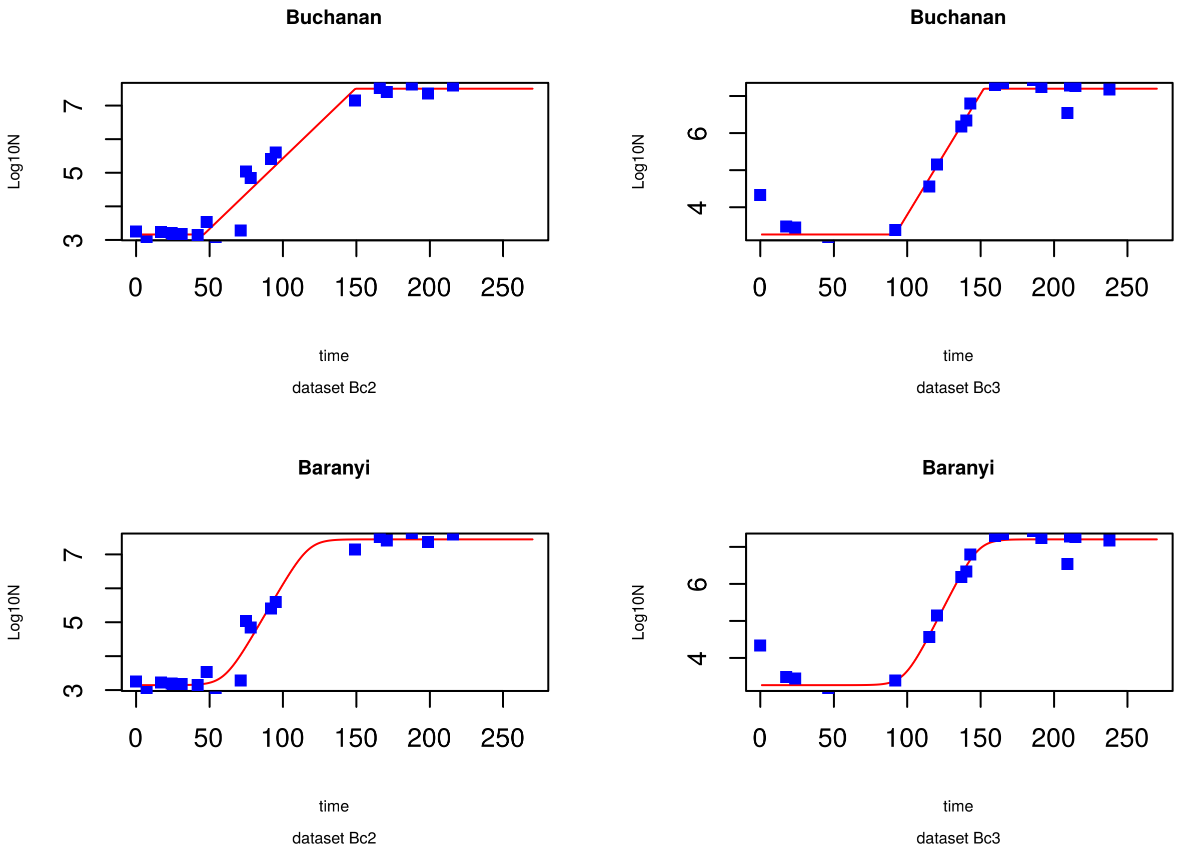


Figure 8.6: Predicted values vs. data points

```
## LOG10Nmax  7.4431721 0.13857817 53.710999 1.686156e-19
```

```
summary(bar3.nlsLM)$parameters
```

```
##              Estimate Std. Error   t value     Pr(>|t|)
## lag       98.0820517  7.6186089 12.874011 7.381774e-10
## mumax      0.1808689  0.0392226  4.611345 2.889377e-04
## LOG10N0    3.2651346  0.1592311 20.505639 6.509231e-13
## LOG10Nmax  7.2034057  0.1467807 49.075958 7.086484e-19
```

8.4 Going further

Now you have all the necessary techniques, to go beyond the scope of this book. With more data sets you will be easily able to make prediction how μ_{max} changes along temperature gradient. From standard error it will be fairly easy to calculate standard deviation, and use it for estimation of your parameters distribution. Than you can use random sampling for stochastic simulations. With some effort and clever queries, you will find how to make beautiful plots using powerful `ggplot2` package and its extensions. Of course, in the beginning it will take some time, and probably frustration, but all in all you are now using one of the most powerful and differentiated IT tools. It's only upon you how you will use the knowledge.

8.5 Copy-paste data set

The *Bacillus cereus* data

```
BcDF <- data.frame(Time = c(0.00, 17.00, 43.00, 66.50, 139.50, 187.00, 283.00,
                            478.00, 526.00, 550.00, 605.50, 887.00,
                            0.00, 7.00, 17.00, 24.00, 25.00, 31.00, 42.00,
                            48.00, 54.00, 71.00, 75.00, 78.00, 92.00, 95.00,
                            149.00, 166.00, 170.50, 187.50, 199.00, 216.00,
                            0.00, 17.62, 23.82, 40.43, 46.18, 71.17, 92.03,
                            115.17, 120.10, 137.05, 140.40, 143.08, 159.85,
                            165.25, 185.68, 191.27, 209.27, 210.95, 214.55,
                            237.90,
                            0.00, 16.22, 19.02, 20.93, 23.40, 40.98, 45.23,
                            47.98, 66.07, 69.35, 71.68, 89.60, 161.32, 166.12,
                            187.20, 190.45, 209.13,
                            0.00, 1.13, 2.05, 2.93, 4.08, 4.98, 6.02, 6.90,
                            8.02, 9.02, 10.08, 10.95, 12.08, 12.98, 14.13,
                            15.38, 16.22, 19.75),
                   LogC = c(3.25, 3.54, 3.53, 3.40, 3.34, 3.38, 3.40, 5.78,
                            7.00, 6.70, 7.34, 6.48, 3.25, 2.95, 3.23, 3.20,
                            3.18, 3.18, 3.15, 3.53, 2.78, 3.28, 5.04, 4.85,
                            5.41, 5.60, 7.15, 7.52, 7.41, 7.63, 7.36, 7.59,
                            4.33, 3.48, 3.45, 2.76, 2.92, 2.65, 3.39, 4.56,
                            5.15, 6.18, 6.34, 6.80, 7.30, 7.35, 7.43, 7.24,
                            6.54, 7.28, 7.27, 7.17, 2.54, 2.30, 2.24, 2.40,
                            2.54, 5.15, 5.73, 6.42, 7.72, 7.94, 7.73, 7.95,
                            8.02, 7.88, 7.99, 7.95, 7.32, 3.67, 3.51, 3.55,
                            3.70, 3.98, 4.46, 4.77, 4.92, 5.28, 6.24, 6.44,
                            6.92, 7.17, 7.18, 7.36, 7.48, 7.68, 7.98),
                   Temp = c(rep(16, times = 12), rep(22, times = 20),
                            rep(15, times = 20), rep(20, times = 17),
                            rep(30, times = 18)),
                   resID = c(rep('Bc1', times = 12), rep('Bc2', times = 20),
                             rep('Bc3', times = 20), rep('Bc4', times = 17),
                             rep('Bc5', times = 18)))
```

Chapter 9

Final indications

9.1 Use Projects

To organize your work, not only in **RStudio**, but also on hard drive you should use projects. It is extremely easy in this IDE. Just click button – *Create new project* – choose *New Directory* and type of project you need. **RStudio** will take care of everything for you. Later you can easily add files of sub-folders with specifying content into your project and use relational links. Also, using projects makes easy to maintain clean global environment, which beginners often tend to underestimate and than they run into troubles.

9.2 Use RMarkdown

To make your work more reproducible, and also to make nice looking output (in html or PDFs) it is a good idea to work in RMarkdown files instead of RScripts. The syntax of **RMarkdown** is nearly identical to original **Markdown**, but allows you to execute and store **RCode**. You can find RMarkdown introduction here. There is also nice cheat sheet which contains all commands you will need. As there is strong pressure to make research and science more open and reproducible, it is strongly advised that you work with proper tools to do it like **RMarkdown** or **Project Jupyter**. If you want to learn more, there is very nice guide by British Ecological Society you should read.

Also, you need to know that to make output in PDFs format, you will first need to install **LaTeX** distribution. There are tons of tutorials how to do it on different operating system. The thing is that even you don't use **LaTeX** directly, **R** uses it *under the hood* to (oversimplifying) to convert **RMarkdown** to PDFs. For **Linux** users, one piece of advice is to install *full* distribution instead of *base*, it take more time and disk space, but will save you frustration of installing additional packages later.

9.3 Use .rds files

Usually we work with plain text files like csv or tsv in **R**. However, there is also a highly valuable format of files called rds, that makes work even more pleasant. It not only preserves all classes of variables, but also takes less space than plain text file, due to compression algorithms. Also when using `saveRDS()` you don't need to define hundreds of parameters, just name of object you want to save and its file name. The downside is that it is format to use directly with **R** and won't work with other programs.

9.4 Nice resources you SHOULD read

There is few hundred of books relating more or less to work in **R**. There is also a huge number of blogs, web pages and other on-line resources that helps with work in **R**. Here you will find some of them that I find very useful in everyday work with R and statistics.

R and its packages

- Basic Transformations and Explorations in the Tidyverse
- R Graphic CookBook free on-line version
- Advanced R free on-line version
- Collection of R cheat sheets
- R Weekly – Collection of best news from data science world

Statistics, programming and modeling

- RECONlearn – free resources on epidemiology
- Norm Matloff's blog discussing issues with statistics and data science
- Rosetta Code – a repository for code to solve large number of problems with tens of different programming languages

9.5 Solving problems

As you saw, **R** can be used for many different purposes, like statistics or modeling, whiteout any programming knowledge. However, the more you will work with it, the sooner you will find how programming skills, can help you will daily work. Of course some problems in the beginning will be to complicated to solve quickly, but spending time on finding solution will save you lots of time in future. Also, if you follow few simple rules, you will work faster and be less frustrated. This rules make simple workflow:

- take some time to analyze the problem
- divide your problem into small pieces
- start with solving piece that you know (or are very close to) solve with least effort
- don't be afraid to ask or look into web for solutions
- take all the small pieces and put them together

Remember – it is better to do something than do nothing. Even if you are stuck with something, you will still benefit from solutions for smaller pieces of your problem, that you managed to find. In the beginning try to think about robustness and speed of your code, but do not seek for best optimized code at all cost. Sometimes it's just not worth spending half an hour to make your code faster by 1 second. The more you will read, the more ideas about how to write best code you will get.

Bibliography

Baty, F. and Delignette-Muller, M.-L. (2014). *nlsMicrobio: Nonlinear regression in predictive microbiology.* R package version 0.0-1.

Chang, W. (2013). *R Graphics Cookbook.* O'Reilly Media, Inc.

Delignette-Muller, M.-L., Dutang, C., and Siberchicot, A. (2017). *fitdistrplus: Help to Fit of a Parametric Distribution to Non-Censored or Censored Data.* R package version 1.0-9.

Elzhov, T. V., Mullen, K. M., Spiess, A.-N., and Bolker, B. (2016). *minpack.lm: R Interface to the Levenberg-Marquardt Nonlinear Least-Squares Algorithm Found in MINPACK, Plus Support for Bounds.* R package version 1.2-1.

Grothendieck, G. (2013). *nls2: Non-linear regression with brute force.* R package version 0.2.

Muggeo, V. M. R. (2017). *segmented: Regression Models with Break-Points / Change-Points Estimation.* R package version 0.5-3.0.

Pouillot, R. (2017). *mc2d: Tools for Two-Dimensional Monte-Carlo Simulations.* R package version 0.1-18.

R Core Team (2017). *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria.

RStudio Team (2016). *RStudio: Integrated Development Environment for R.* RStudio, Inc., Boston, MA.

Soetaert, K., Petzoldt, T., and Setzer, R. W. (2018). *deSolve: Solvers for Initial Value Problems of Differential Equations ('ODE', 'DAE', 'DDE').* R package version 1.21.

Wickham, H. (2010). A layered grammar of graphics. *Journal of Computational and Graphical Statistics*, 19(1):3–28.

Wickham, H. (2014). Tidy data. *Journal of Statistical Software*, 59(10):1–23.

Wickham, H. (2017). *tidyverse: Easily Install and Load the 'Tidyverse'.* R package version 1.2.1.

Wickham, H. and Chang, W. (2016). *ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics.* R package version 2.2.1.

Wickham, H., Francois, R., Henry, L., and Müller, K. (2017). *dplyr: A Grammar of Data Manipulation.* R package version 0.7.4.

Wickham, H. and Henry, L. (2018). *tidyr: Easily Tidy Data with 'spread()' and 'gather()' Functions.* R package version 0.8.0.

Wilkinson, L. (2005). *The Grammar of Graphics (Statistics and Computing).* Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Xie, Y. (2015). *Dynamic Documents with R and knitr.* Chapman and Hall/CRC, Boca Raton, Florida, 2nd edition. ISBN 978-1498716963.

Xie, Y. (2016). *bookdown: Authoring Books and Technical Documents with R Markdown*. Chapman and Hall/CRC, Boca Raton, Florida. ISBN 978-1138700109.

Xie, Y. (2018). *bookdown: Authoring Books and Technical Documents with R Markdown*. R package version 0.7.