

Biological Computing in R: Silwood Masters (CMEE, EEC, EA, NHM)

Imperial College London, 2014–15

Samraat Pawar (s.pawar@imperial.ac.uk),
(with inputs from many others!)

November 6, 2014

Contents

0	Introduction	1
0.1	What is this document all about?	1
0.2	Outline of the this module	1
0.3	Conventions used in this document	2
0.4	A note on being organized	2
0.5	Readings	2
1	Introduction to R	3
1.1	What is R?	3
1.2	Would you ever need anything other than R?	3
1.3	Installing R	3
1.4	Getting started	4
1.5	Useful R commands	5
1.6	Some Basics	5
1.6.1	Variable names and Tabbing	5
1.6.2	R likes E Notation	6
1.6.3	Operators	6
1.6.4	When things go wrong	6
1.6.5	Types of parentheses	7
1.7	Data types	7
1.7.1	Vectors	7
1.7.2	Matrices and arrays	7
1.7.3	Data frames	8
1.7.4	Lists	9
1.8	Variable Types, Type Conversion and Special Values	10
1.9	Creating and Manipulating Data structures	11
1.9.1	Sequences	11
1.9.2	Strings and Pasting	11
1.9.3	Indices and Indexing	12
1.9.4	Recycling	12
1.9.5	Basic vector-matrix operations	12
1.10	Your R Analysis Workflow	13
1.10.1	The Working Directory	13
1.11	The R analysis workflow	13
1.12	Importing and Exporting Data	14
1.13	Writing Functions	15
1.14	Practical 2.1	16
1.15	Control statements	16
1.16	Running R code	17

1.17	Useful R Functions	17
1.17.1	Mathematical	17
1.17.2	Strings	18
1.17.3	Statistical	18
1.17.4	Random number distributions	18
1.18	Packages	18
1.19	Readings	18
2	Plotting and graphics in R	21
2.1	Basic plotting	21
2.1.1	Scatter Plot	22
2.1.2	Histograms	23
2.1.3	Subplots	23
2.1.4	Overlaying plots	23
2.1.5	Exercise	23
2.1.6	Saving your graphics	24
2.1.7	Boxplots	24
2.1.8	Practical 3.1	24
2.2	Publication-quality figures in R	26
2.2.1	Basic graphs with <code>qplot</code>	26
2.2.2	Various <code>geom</code>	29
2.2.3	Practical 3.2	30
2.2.4	Advanced plotting: <code>ggplot</code>	32
2.2.5	Case study 1: plotting a matrix	32
2.2.6	Case study 2: plotting two dataframes	34
2.2.7	Case study 3: annotating the plot	35
2.2.8	Case study 4: mathematical display	36
2.3	Readings	37
3	Advanced topics in R	39
3.1	Vectorization	39
3.2	Breaking out of loops	40
3.3	Using <code>next</code>	41
3.4	“Catching” errors	41
3.5	Generating Random Numbers	41
3.6	Debugging	42
3.7	Practical 3.1	43
3.8	Launching/Running R in batch mode	43
3.9	Readings	44

Chapter 0

Introduction

Donald Knuth, 1995: *Science is what we understand well enough to explain to a computer. Art is everything else we do*

0.1 What is this document all about?

This document contains the content of the Imperial College London Department of Life Sciences Msc/MRes Modules on Scientific Computing in R. The content will be updated over the week, with further sections and chapters added as we go along. This document is accompanied by data and code on which you can practice your skills in your own time and during the Practical sessions. these materials are available (and will be updated regularly) at: <https://bitbucket.org/mhasoba/cmee2014masterepo/>.

It is important that you work through the exercises and problems in each chapter/section. This document does not tell you every single thing you need to know to perform the exercises in it. In programming, you learn far more from trying to solve problems than from reading about how others have solved them — that is, you have license to google it! You will be provided guidelines for what makes good or efficient solutions. Later, when you have submitted your exercises and practicals (only relevant to the CMEEs, but the rest should have a go!), feedback will be provided on your solutions.

0.2 Outline of the this module

The content and structure of this week is geared towards the following objectives:

- Give you a introduction to R syntax and programming conventions, assuming you have never set your eyes on R or any other programming language before — we will breeze through this as you have already been introduced to R in the Stats Week!
- Teach you principles of clean and efficient programming in R, including delightful things like vectorization and debugging.
- Teach you how to generate publication quality graphics in R — publication quality is thesis quality!
- Teach you how to develop reproducible data analyses “work flows” so (or anybody else) run and re-run your analyses, graphics outputs and all, in R.

You will use R a lot during the rest of your courses and probably your thesis and career — the aim is to lay down the foundations for you to become very comfortable with it!

0.3 Conventions used in this document

You will find all R commandline/console arguments, code snippets and output in colored boxes like this:

```
> ls()
```

Here `>` is the R prompt, and will type the commands/code that you see from this document into the R command line (or copy-paste, but not recommended!). I have aimed to make the content of this module computer platform (Mac, PC or Linux) independent because many of you are probably working (or later will be) with R on personal laptops or desktops. Indeed, platform-independence of data analyses is one of the main reasons why you are using R! Finally, note that:

★ In all subsequent chapters, lines marked with a star like this are things for you to do.

0.4 A note on being organized

In this module, you will write plenty of R code, deal with different data files, and produce text and graphic outputs. Please keep all your code, data inputs and results outputs organized in separate directories named `Code`, `Data`, `Results` (or equivalent) respectively. The CMEEs are already implementing this along with version control in git (if you are intrigued, please look up CMEE Week 1 lectures) at <https://bitbucket.org/mhasoba/cmee2014masterepo/>.

Note that R, as in UNIX like systems (Mac, Linux), uses `/` instead of the `\` used in Windows for directory path specification. Also, in general, we will be using relative paths throughout the exercises and practicals (more on this later).

0.5 Readings

(More will appear in different sections/chapters)

- Use the internet! Google “R tutorial”, and plenty will pop up. Choose one that seems the most intuitive to you.
- Bolker, B. M.: Ecological Models and Data in R (eBook and Hardcover available).
- Beckerman, A. P. & Petchey, O. L. (2012) Getting started with R: an introduction for biologists. Oxford, Oxford University Press.
Good, short, general introduction
- Crawley, R. (2013) The R book. 2nd edition. Chichester, Wiley.
Excellent but enormous reference book, code and data available from www.bio.ic.ac.uk/research/mjcraw/therbook/index.htm

Chapter 1

Introduction to R

1.1 What is R?

R is a freely available statistical software with strong programming capabilities. R has become incredibly popular in biology due to several factors: i) many packages are available to perform all sorts of statistical and mathematical analyses; ii) it has been developed and scrutinised by top level academic statisticians; iii) it can produce beautiful, publication-quality graphics; iv) it has a very good support for matrix-algebra.

1.2 Would you ever need anything other than R?

Although we can technically program in R, the programming environment is not the greatest: especially the way types are managed is problematic (e.g., often your matrix will become a vector if it has only one column/row!), and the way errors and warnings are handled and displayed (often unintelligibly!).

Nevertheless, being able to program R means you can develop and automate your statistical analyses and the generation of figures into a reproducible work flow (there's that term again!). However, if your work also includes extensive numerical simulations, manipulation of very large matrices, bioinformatics, or complex workflows including databases, you will be much better off if you *also* know another programming language that is more versatile, computationally efficient (like `python`, `PERL` or `C`).

But for most of you, R will do the job, so you may revel in that knowledge! In particular, `python` is recommended, as it is reasonably efficient and has very nice and clean syntax. With something like `python`, You can embed your R analysis work flow inside a more complex meta-workflow, for example, one that includes interfacing with the internet, manipulating/querying databases, and compiling \LaTeX documents.

1.3 Installing R

Linux/Ubuntu: run the following in terminal

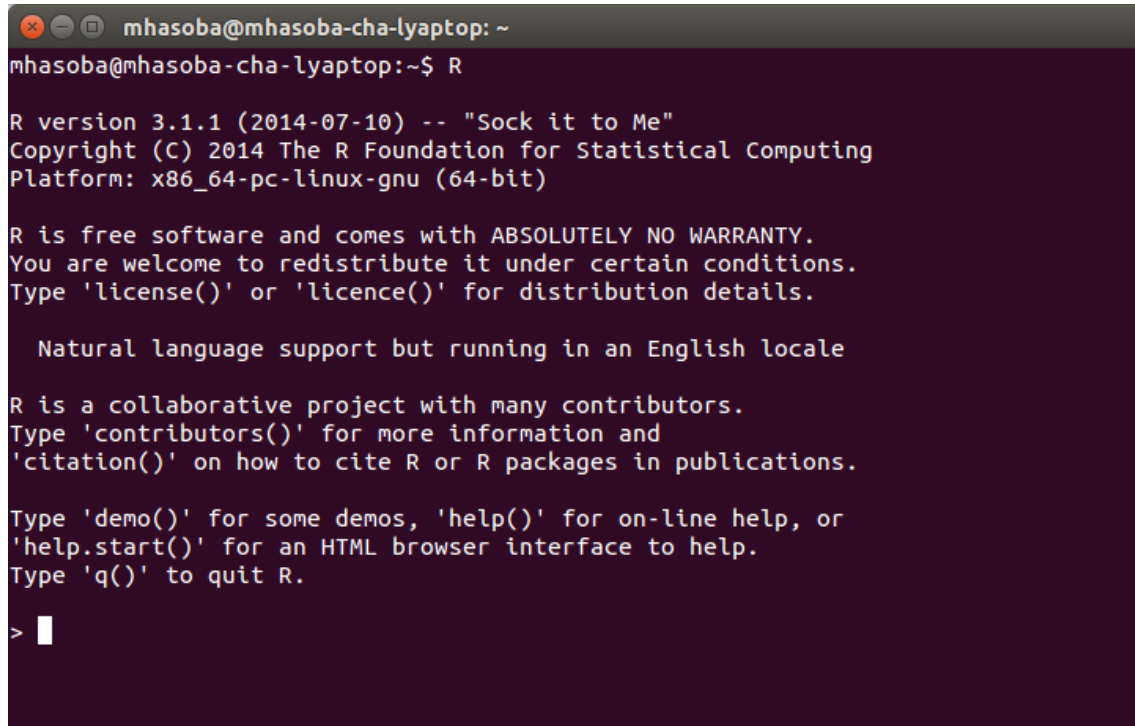
```
$ sudo apt-get install r-base r-base-dev
```

Mac OS X: download and install from <http://cran.r-project.org/bin/macosx/>

Windows: download and install from <http://cran.r-project.org/bin/windows/base/>

1.4 Getting started

Launch R (From Applications menu on Window or Mac, from terminal in Linux/Ubuntu) — it should look something like this (on Linux/Ubuntu or Mac terminal):

A terminal window titled 'mhasoba@mhasoba-cha-lyaptop: ~' showing the R startup sequence. The prompt is 'mhasoba@mhasoba-cha-lyaptop:~\$ R'. The output includes the R version (3.1.1), copyright information, platform details (x86_64-pc-linux-gnu 64-bit), a disclaimer about warranty, information about language support, contributors, and help options. The prompt returns to '>' at the end.

```
mhasoba@mhasoba-cha-lyaptop:~$ R
R version 3.1.1 (2014-07-10) -- "Sock it to Me"
Copyright (C) 2014 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

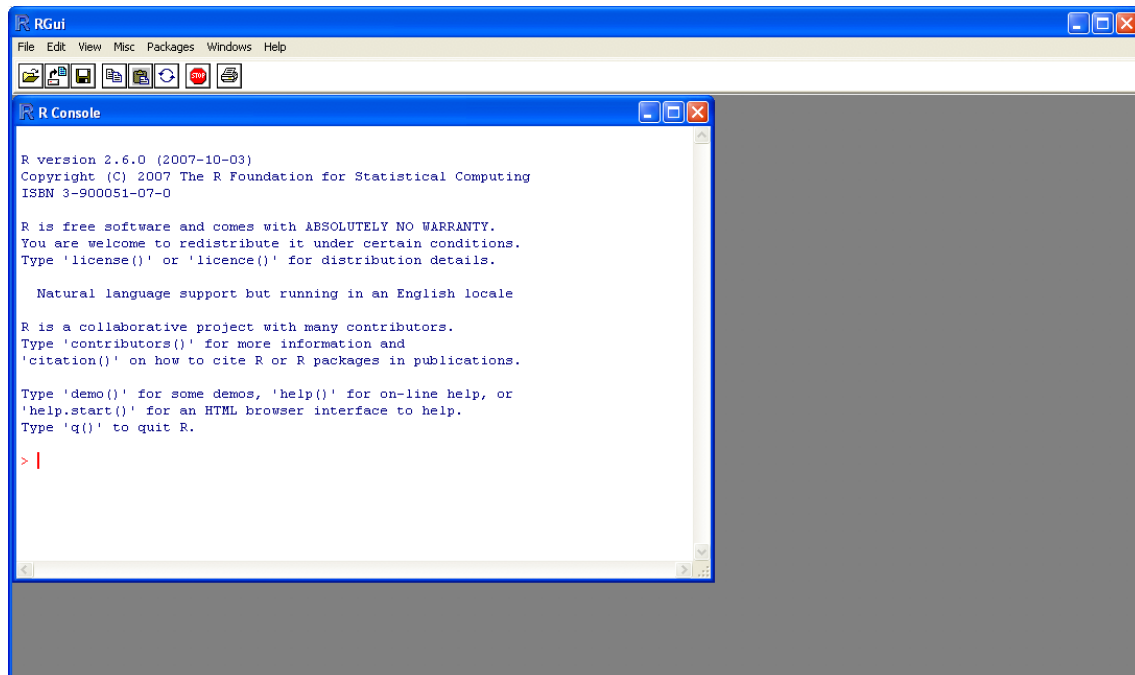
Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> 
```

Or like this (Windows “console”, similar in Mac):

A screenshot of the RGui application window. The title bar says 'RGui'. The menu bar includes 'File', 'Edit', 'View', 'Misc', 'Packages', 'Windows', and 'Help'. Below the menu bar is a toolbar with icons for file operations and help. The main window is titled 'R Console' and contains the same R startup text as the terminal window, ending with the prompt '> |'.

You can also use an IDE (Interactive Development Environment) that can offer delights like syntax highlighting (google it!), such as RStudio, geany, vim, etc.

1.5 Useful R commands

<code>ls()</code>	list all the variables in the work space
<code>rm('a', 'b')</code>	remove variable(s) a and b
<code>rm(list=ls())</code>	remove all variable(s)
<code>getwd()</code>	get current working directory
<code>setwd('Path')</code>	set working directory to Path
<code>q()</code>	quit R
<code>?Command</code>	show the documentation of Command
<code>??Keyword</code>	search the all packages/functions with Keyword, “fuzzy search”

1.6 Some Basics

Try out the following in the R console:

```
> a <- 4 # assignment
> a
[1] 4
> a*a # product
[1] 16
> a_squared <- a*a
> sqrt(a_squared) # square root
[1] 4
> v <- c(0, 1, 2, 3, 4) # c: "concatenate"
```

`c()` (concatenate) is one of the most commonly used functions — Don't forget it! (try `?c`)

Note that any text after a “#” is ignored by R — handy for commenting. In general, please comment your code and scripts, for *everybody's* sake!

```
> v # Display the vector variable you created
[1] 0 1 2 3 4
> is.vector(v) # check if it's a vector
[1] TRUE
> mean(v) # mean
[1] 2
> var(v) # variance
[1] 2.5
> median(v) # median
[1] 2
> sum(v) # sum all elements
[1] 10
> prod(v + 1) # multiply
[1] 120
> length(v) # length of vector
[1] 5
```

1.6.1 Variable names and Tabbing

```
> wing.width.cm <- 1.2 #Using dot notation
> wing.length.cm <- c(4.7, 5.2, 4.8)
```

Using tabbing with dot notation can be handy. Type:

```
> wing.
```

And then hit the `tab` key. This is handy, but good style and readability is more important than just convenient variable names. Variable names should be as obvious as possible, not over-long!

1.6.2 R likes E Notation

```
> 1E4
[1] 10000
> 1e4
[1] 10000
> 5e-2
[1] 0.05
```

R uses *E* notation to print very large or small numbers:

```
> 1E4 ^ 2
[1] 1e+08
> 1 / 3 / 1e8
[1] 3.333333e-09
```

1.6.3 Operators

The usual operators are available in R (slight differences from `python`):

+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Power
%%	Modulo
%/%	Integer division
==	Equals
!=	Differs
>	Greater
>=	Greater or equal
&	Logical and
	Logical or
!	Logical not

1.6.4 When things go wrong

Syntax errors are those where you've just made a typing mistake. Here are some common problems:

- missing close bracket leads to continuation line.

```
> x <- (1 + (2 * 3)
+
```

Hit `Ctrl C` (see below) or keep typing!

- Too many parentheses: `2 + (2*3)`
- wrong/mismatched brackets (see next subsection).
- Do not mix double quotes and single quotes.
- When things seem to take too long, try `Ctrl + C`

1.6.5 Types of parentheses

R has a somewhat confusing array of parentheses that you need to get used to:

- `f(3, 4)` – call the function `f`, with `arg1=3`, `arg2=4`.
- `a + (b*c)` – use to enforce order over which statements are executed.
- `{ expr1; expr2; ...exprn }` – group a set of expressions into one compound expression. Value returned is value of last expression; used in looping/conditionals.
- `x[4]` – get the 4th element of the vector `x`.
- `l[[3]]` – get the 3rd element of some list `l`, and return it. (compare with `l[3]` which returns a list with just the 3rd element inside) (more on lists in next section)

1.7 Data types

Like `python` (why `python`? Ask the CMEEs!), R comes with data-types. Mastering these will help you write better, more efficient programs and also handle diverse between datasets. Now get back into R (if you quit R using `q()`), and type:

1.7.1 Vectors

Vectors are a fundamental object for R. Scalars (single numbers) are treated as vector of length 1.

```
> a <- 5
> is.vector(a)
[1] TRUE
> v1 <- c(0.02, 0.5, 1)
> v2 <- c("a", "bc", "def", "ghij")
> v3 <- c(TRUE, TRUE, FALSE)
```

1.7.2 Matrices and arrays

R has many functions to manipulate matrices (two-dimensional vectors) and arrays (multi-dimensional vectors).

```
> m1 <- matrix(1:25, 5, 5)
> m1
      [,1] [,2] [,3] [,4] [,5]
[1,]    1    6   11   16   21
[2,]    2    7   12   17   22
[3,]    3    8   13   18   23
[4,]    4    9   14   19   24
```

```

[5,]    5    10    15    20    25
> m1 <- matrix(1:25, 5, 5, byrow=TRUE)
> m1
      [,1] [,2] [,3] [,4] [,5]
[1,]     1     2     3     4     5
[2,]     6     7     8     9    10
[3,]    11    12    13    14    15
[4,]    16    17    18    19    20
[5,]    21    22    23    24    25
> dim(m1)
[1] 5 5
> m1[1,2]
[1] 2
> m1[1,2:4]
[1] 2 3 4
> m1[1:2,2:4]
      [,1] [,2] [,3]
[1,]     2     3     4
[2,]     7     8     9
> arr1 <- array(1:50, c(5, 5, 2))
> arr1
, , 1

      [,1] [,2] [,3] [,4] [,5]
[1,]     1     6    11    16    21
[2,]     2     7    12    17    22
[3,]     3     8    13    18    23
[4,]     4     9    14    19    24
[5,]     5    10    15    20    25

, , 2

      [,1] [,2] [,3] [,4] [,5]
[1,]    26    31    36    41    46
[2,]    27    32    37    42    47
[3,]    28    33    38    43    48
[4,]    29    34    39    44    49
[5,]    30    35    40    45    50

```

1.7.3 Data frames

This is a very important data type that is peculiar to R. It is great for storing your data. Basically, it's a two-dimensional table in which each column can contain a different data type (e.g., numbers, strings, boolean). You can think of a dataframe as a spreadsheet. Data frames are great for plotting your data, performing regressions and such. Later (Chapter 2) you will see that fancy plotting using `ggplot` demands the use of dataframes. Now try the following:

```

> Col1 <- 1:10
> Col1
[1] 1 2 3 4 5 6 7 8 9 10
> Col2 <- LETTERS[1:10]
> Col2
[1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J"
> Col3 <- runif(10) # 10 random num. from uniform
> Col3
[1] 0.29109 0.91495 0.64962 0.95503 0.26589 0.02482 0.59718
[8] 0.99134 0.98786 0.86168
> MyDF <- data.frame(Col1, Col2, Col3)

```

```

> MyDF
  Col1 Col2      Col3
1    1    A 0.2910981
2    2    B 0.9149558
3    3    C 0.6496248
4    4    D 0.9550331
5    5    E 0.2658936
6    6    F 0.0248217
7    7    G 0.5971868
8    8    H 0.9913407
9    9    I 0.9878679
10   10    J 0.8616854
> names(MyDF) <- c("A.name", "another", "another.one")
> MyDF
  A.name another another.one
1      1      A 0.2910981
2      2      B 0.9149558
3      3      C 0.6496248
4      4      D 0.9550331
5      5      E 0.2658936
6      6      F 0.0248217
7      7      G 0.5971868
8      8      H 0.9913407
9      9      I 0.9878679
10     10      J 0.8616854
> MyDF$A.name
[1] 1 2 3 4 5 6 7 8 9 10
> MyDF[,1]
[1] 1 2 3 4 5 6 7 8 9 10
> MyDF[c("A.name", "another")]
  A.name another
1      1      A
2      2      B
3      3      C
4      4      D
5      5      E
6      6      F
7      7      G
8      8      H
9      9      I
10     10      J
> class(MyDF)
[1] "data.frame"
> str(MyDF) # a very useful command!
'data.frame': 10 obs. of 3 variables:
 $ A.name      : int  1 2 3 4 5 6 7 8 9 10
 $ another     : Factor w/ 10 levels "A", "B", "C", "D", ...
 $ another.one : num  0.291 0.915 0.65 0.955 0.266 ...

```

1.7.4 Lists

A list is used to collect a group of objects of different sizes and types. A list is simply an ordered collection of objects (that can be other variables) (a bit like python lists!).

```

> l1 <- list(names=c("Fred", "Bob"), ages=c(42, 77, 13, 91))
> l1
$names
[1] "Fred" "Bob"

```

```
$ages
[1] 42 77 13 91

> l1[[1]]
[1] "Fred" "Bob"
> l1[[2]]
[1] 42 77 13 91
> l1[["ages"]]
[1] 42 77 13 91
> l1$ages
[1] 42 77 13 91
```

You can build lists of lists too. Lists are often returned as the result of a complex function (e.g. linear model fitting using `lm()`) to return all relevant information in one object.

1.8 Variable Types, Type Conversion and Special Values

There are different kinds of data variable types such as integer, float (including real numbers), and string (e.g., text words). Beware of the difference between NA (Not Available) and NaN (Not a Number).

```
> as.integer(3.1)
[1] 3
> as.real(4)
[1] 4
> as.roman(155)
[1] CLV
> as.character(155)
[1] "155"
> as.logical(5)
[1] TRUE
> as.logical(0)
[1] FALSE
> b <- NA
> is.na(b)
[1] TRUE
> b <- 0./0.
> b
[1] NaN
> is.nan(b)
[1] TRUE
> b <- 5/0
> b
[1] Inf
> is.nan(b)
[1] FALSE
> is.infinite(b)
[1] TRUE
> is.finite(b)
[1] FALSE
> is.finite(0/0)
[1] FALSE
```

1.9 Creating and Manipulating Data structures

1.9.1 Sequences

The `:` operator creates vectors of sequential integers:

```
> years <- 1990:2009
> years
[1] 1990 1991 1992 1993 1994 1995 1996 1997 1998 1999
[11] 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009

> years <- 2009:1990 # or in reverse order
> years
[1] 2009 2008 2007 2006 2005 2004 2003 2002 2001 2000
[11] 1999 1998 1997 1996 1995 1994 1993 1992 1991 1990
```

For sequences of fractional numbers, you have to use `seq()` :

```
> seq(1, 10, 0.5)
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 6.0 6.5 7.0 7.5 ↵
      8.0
[16] 8.5 9.0 9.5 10.0
```

You can also `seq(from=1,to=10, by=0.5)` OR `seq(from=1, by=0.5, to=10)` with the same effect (try it) — this explicit, “argument matching” approach is partly why R is so popular.

1.9.2 Strings and Pasting

R’s string handling ain’t elegant or pretty (unlike `python!`), but it works:

```
> species.name <- "Quercus robur" #double quotes
> species.name
[1] "Quercus robur"
> species.name <- 'Fraxinus excelsior' #single quotes
> species.name
[1] "Fraxinus excelsior"
> paste("Quercus", "robur")
[1] "Quercus robur"
> paste("Quercus", "robur", sep = "") #Get rid of space
"Quercusrobur"
> paste("Quercus", "robur", sep = ", ") #insert comma to separate
```

And as is the case with so many R functions, pasting works on vectors:

```
> paste('Year is:', 1990:2000)
[1] "Year is: 1990" "Year is: 1991" "Year is: 1992" "Year is: 1993"
[5] "Year is: 1994" "Year is: 1995" "Year is: 1996" "Year is: 1997"
[9] "Year is: 1998" "Year is: 1999" "Year is: 2000"
```

Note that this last example creates a vector of 11 strings.

1.9.3 Indices and Indexing

Every element of a vector in R has an order: the first value, second, third, etc. To illustrate this, type:

```
> MyVar <- c( 'a' , 'b' , 'c' , 'd' , 'e' ) # create a simple vector
```

Then, square brackets extract values based on their position in the order:

```
> MyVar[1] # Show element in first position
[1] "a"
> MyVar[4]
[1] "d" # Show element in fourth position
```

The values in square brackets are called “indices” — they give the index (position) of the required value. We can also select sets of values in different orders, or repeat values:

```
> MyVar[c(3,2,1)] # reverse order
[1] "c" "b" "a"
MyVar[c(1,1,5,5)] # repeat indices
[1] "a" "a" "e" "e"
```

So you can manipulate vectors by indexing:

```
> v <- c(0, 1, 2, 3, 4) # Re-create the vector variable v
> v[3] # access one element
[1] 2
> v[1:3] # access sequential elements
[1] 0 1 2
> v[-3] # remove elements
[1] 0 1 3 4
> v[c(1, 4)] # access non-sequential
[1] 0 3
```

1.9.4 Recycling

When vectors are of different lengths, R will recycle the shorter one to make a vector of the same length:

```
a <- c(1,5) + 2
x <- c(1,2); y <- c(5,3,9,2)
x + y
x + c(y,1) ## somewhat strange!
```

Recycling is convenient, but dangerous!

1.9.5 Basic vector-matrix operations

```
> v2 <- v
> v2 <- v2*2 # whole-vector operation
> v2
[1] 0 2 4 6 8
```



```

> v * v2 # product element-wise
[1] 0 2 8 18 32
> t(v) # transpose the vector
      [,1] [,2] [,3] [,4] [,5]
[1,] 0    1    2    3    4
> v %*% t(v) # matrix/vector product
      [,1] [,2] [,3] [,4] [,5]
[1,] 0    0    0    0    0
[2,] 0    1    2    3    4
[3,] 0    2    4    6    8
[4,] 0    3    6    9   12
[5,] 0    4    8   12   16
> v3 <- 1:7 # assign using sequence
> v3
[1] 1 2 3 4 5 6 7
> v4 <- c(v2, v3) # concatenate vectors
> v4
[1] 0 2 4 6 8 1 2 3 4 5 6 7
> q() # quit

```

1.10 Your R Analysis Workflow

1.10.1 The Working Directory

Before we go any further, let's get ourselves organized. In R, type:

```
> getwd()
```

This tells you what the current “working directory” is.

In using R for an analysis, you will likely use and create several files. This means that it is sensible to create a folder (directory) to keep all code files together. You can then set R to work from this directory, so that files are easy to find and run — this will be your working directory. So now do the following:

- ★ Create a directory called `Week5` in an appropriate location (For CMEEs, in `CMEECourseWork` for others, some place you can remember!
- ★ Create subdirectories within `CMEECourseWork/Week5` called `Code`, `Data`, and `Results`

You can create directories using `dir.create()` within R

Use relative paths Using relative paths in your R scripts and code will make your code computer independent and your life better! For example, in R, `../Data/mydata.txt` specifies a file named `mydata.txt` located in the “parent” of the current directory. Now, set the working directory to be `Week5/Code`:

```

> setwd("FullPathUpToHere/Week5/Code")
> dir()

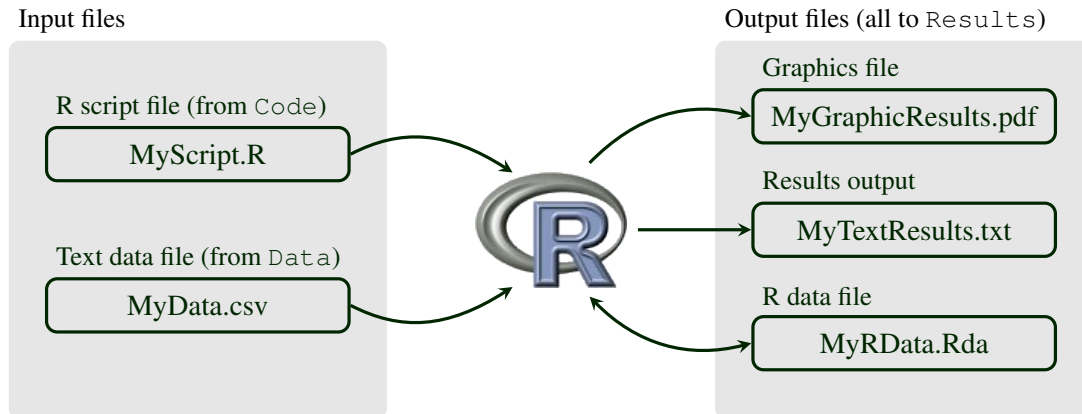
```

Note that `FullPathUpToHere` is not to be taken literally and entered! For example, if you created `Week5` in `H:`, you would use `setwd("\\H:/MyICStatsModules/Code")`

For CMEE, it would be `setwd("\\FullPathUpToHere/CMEECourseWork/Week5/Code")`

1.11 The R analysis workflow

Your typical R analysis workflow will be as follows:



Some details on each kind of file:

R script files These are plain text files containing all the R code needed for an analysis. These should always be created with a simple text editor like Notepad (Windows), TextEdit (MacOS) or Geany (Linux) and saved with the extension `*.R`. We will use the built-in editor in R in this class. Alternatively, if RStudio is a good option because it also works across platforms. Try it. A big advantage of something like RStudio is that you will get syntax highlighting, which is very handy and will make R programming far more convenient and error-free. You should *never* use Word to save or edit these files as R can only read code from plain text files.

Text data files These are files of data in plain text format containing one or more columns of data (numbers, strings, or both). Although there are several format options, we will typically be using csv files, where the entries are separated by commas. These are easy to create and export from Excel (if that's what you use...)¹.

Results output files These are a plain text files contain your results, such the the summary of output of a regression or ANOVA analysis. Typically, you will putput your results in a table format where the columns are separated by commas (csv) or tabs (tab-delimited)

Graphics files R can export graphics in a wide range of formats. This can be done automatically from R code and we will look at this later but you can also select a graphics window and click 'File > Save as...'

Rdata files You can save any data loaded or created in R, including model outputs and other things, into a single `Rdata` file. These are not plain text and can only be read by R, but can hold all the data from an analysis in a single handy location. I never use these, but you can, if you want.

1.12 Importing and Exporting Data

Now we are ready to see how to import and export data in R, typically the first step of your analysis. The best option is to have your data in a comma separated value text file or in a tab separated file. Then, you can use the function `read.csv` (or `read.table`) to import your data (note the relative paths!):

¹If you are using a computer from elsewhere in the EU, Excel may use a comma ($\pi = 3,1416$) instead of a decimal point ($\pi = 3.1416$). In this case, csv files may use a semi-colon to separate columns and you can use the alternative function `read.csv2()` to read them into R.

```

> MyData <- read.csv("../Data/trees.csv")
> head(MyData) # Have a quick look at the data frame
> str(MyData) # Have a quick look at the column types
> MyData <- read.csv("../Data/trees.csv", header = TRUE) # with headers
> MyData <- read.table("../Data/trees.csv", sep = ',',
header = TRUE) # A more general way
> head(MyData)
> MyData <- read.csv("../Data/trees.csv", skip = 5) # skip first 5 lines

```

Note that the resulting `MyData` in your workspace is a R dataframe. You can also save your data frames using `write.table` or `write.csv`:

```

> write.csv(MyData, "../Results/MyData.csv")
> dir("../Results/") # Check if it worked
> write.table(MyData[1,], file = "../Results/MyData.csv", append=TRUE) # ↔
  append
> write.csv(MyData, "../Results/MyData.csv", row.names=TRUE) # write row ↔
  names
> write.table(MyData, "../Results/MyData.csv", col.names=FALSE) # ignore col ↔
  names

```

1.13 Writing Functions

R lets you write your own functions. The syntax is quite simple, with each function accepting arguments and returning a value:

```

MyFunction <- function(Arg1, Arg2){

  ## statements involving Arg1, Arg2

  return (ReturnValue)

}

```

★ Type the following in a script file called `TreeHeight.R`, save it in your Code directory and run it using `source`:

```

# This function calculates heights of trees
# from the angle of elevation and the distance
# from the base using the trigonometric formula
# height = distance * tan(radians)
#
# Arguments:
# degrees      The angle of elevation
# distance     The distance from base
#
# Output:
# The height of the tree, same units as "distance"

TreeHeight <- function(degrees, distance)
{
  radians <- degrees * pi / 180
  height <- distance * tan(radians)
  print(paste("Tree height is:", height))
}

```

```

    return (height)
}

TreeHeight(37, 40)

```

1.14 Practical 2.1

Modify the script `TreeHeight.R` so that it does the following:

- ★ Loads `trees.csv` and calculates tree heights for all trees in the data. Note that the distances have been measured in meters. (Hint: use relative paths))
- ★ Creates a csv output file called `TreeHts.csv` in `Results` that contains the calculated tree heights along with the original data in the following format (only first two rows and headers shown):

```

"Species","Distance.m","Angle.degrees","Tree.Height.m"
"Populus tremula",31.6658337740228,41.2826361937914,25.462680727681
"Quercus robur",45.984992608428,44.5359166583512,46.094124200205

```

1.15 Control statements

In R, you can write `if`, `then`, `else` statements, and `for` and `while` loops like any programming language. However, loops are slow in R, so use them sparingly.

- ★ Type the following in a script file called `control.R` (save it in your `Code` directory)

```

## If statement
a <- TRUE
if (a == TRUE){
  print ("a is TRUE")
} else {
  print ("a is FALSE")
}

## On a single line
z <- runif(1) ##random number
if (z <= 0.5) {
  print ("Less than a quarter")}

## For loop using a sequence
for (i in 1:100){
  j <- i * i
  print(paste(i, " squared is", j ))
}

## For loop over vector of strings
for(species in c('Heliodoxa rubinoides',
                 'Boissonneaua jardini',
                 'Sula neboxii'))
{
  print(paste('The species is', species))
}

## for loop using a vector

```

```

v1 <- c("a", "bc", "def")
for (i in v1){
  print(i)
}

## While loop
i <- 0
while (i<100){
  i <- i+1
  print(i^2)
}

```

1.16 Running R code

You can run the code you wrote in blocks to test and understand it:

- ★ Place the cursor on the first line of code and run it by pressing the keyboard shortcut (PC: ctrl+R, Mac: command+enter, Linux: ctrl+enter if you are using geany).

Typing in commands interactively or running in blocks is good for starters, but you will want to switch to putting your sequence of commands into a script file, and then ask R to run those commands. This is necessary also because you will want to just run your full analysis and outputs all the results. The way to run `*.R` script/code from the command line is to `source` it. This causes R to accept code input from a named file and run it:

```
> source("control.R") # Assuming you are in Code directory!
```

Note that you will need to add the directory path to the file name (`control.R` in the above example), if the file is not in your working directory. For example, `../Code/control.R` if you are in, say, `Data`.

1.17 Useful R Functions

There are a number of very useful functions available by default (in the “base packages”).

1.17.1 Mathematical

<code>log(x)</code>	Natural logarithm
<code>log10(x)</code>	Logarithm in base 10
<code>exp(x)</code>	e^x
<code>abs(x)</code>	Absolute value
<code>floor(x)</code>	Largest integer $< x$
<code>ceiling(x)</code>	Smallest integer $> x$
<code>pi</code>	π
<code>sqrt(x)</code>	\sqrt{x}
<code>sin(x)</code>	Sinus function

1.17.2 Strings

<code>strsplit(x, ' ; ')</code>	Split the string according to ' ; '
<code>nchar(x)</code>	Number of characters
<code>toupper(x)</code>	Set to upper case
<code>tolower(x)</code>	Set to lower case
<code>paste(x1, x2, sep=' ; ')</code>	Join the strings inserting ' ; '

1.17.3 Statistical

<code>mean(x)</code>	Compute mean (of a vector or matrix)
<code>sd(x)</code>	Standard deviation
<code>var(x)</code>	Variance
<code>median(x)</code>	Median
<code>quantile(x, 0.05)</code>	Compute the 0.05 quantile
<code>range(x)</code>	Range of the data
<code>min(x)</code>	Minimum
<code>max(x)</code>	Maximum
<code>sum(x)</code>	Sum all elements

1.17.4 Random number distributions

<code>rnorm(10, m=0, sd=1)</code>	Draw 10 normal random numbers with mean 0 and s.d. 1
<code>dnorm(x, m=0, sd=1)</code>	Density function
<code>qnorm(x, m=0, sd=1)</code>	Cumulative density function
<code>runif(20, min=0, max=2)</code>	Twenty random numbers from uniform [0,2]
<code>rpois(20, lambda=10)</code>	Twenty random numbers from Poisson(λ)

1.18 Packages

The main strength of R is that users can easily build packages and share them through `cran.r-project.org`. There are packages to do most statistical and mathematical analysis you might conceive, so check them out before reinventing the wheel! Visit `cran.r-project.org` and go to Packages to see a list and a brief description. To install a package, within R type `install.packages()` and choose the package to install.

1.19 Readings

- The Use R! series (the yellow books) by Springer are really good. In particular, consider: “A Beginner’s Guide to R”, “R by Example”, “Numerical Ecology With R”, “ggplot2” (we’ll see this in Chapter 2), “A Primer of Ecology with R”, “Nonlinear Regression with R”, “Analysis of Phylogenetics and Evolution with R”.
- For more focus on dynamical models: Soetaert & Herman. 2009 “A practical guide to ecological modelling: using R as a simulation platform”.
- There are excellent websites besides cran. In particular, check out `www.statmethods.net` and `http://en.wikibooks.org/wiki/R_Programming`.

- For those who are coming with Matlab experience: <http://www.math.umaine.edu/~hiebeler/comp/matlabR.html>

Chapter 2

Plotting and graphics in R

2.1 Basic plotting

R can produce beautiful graphics, without the time consuming and fiddly methods that you might have used in Excel or equivalent (it's not you, it's Excel!). Later, you can explore how the package `ggplot2`, can allow the rapid creation of truly elegant publication-grade graphics. However, in many cases you just want to quickly plot the data for exploratory analysis of your data. Here are the basic plotting commands:

<code>plot(x, y)</code>	Scatterplot
<code>plot(y~x)</code>	Scatterplot with <code>y</code> as a response variable
<code>hist(mydata)</code>	Histogram
<code>barplot(mydata)</code>	Bar plot
<code>points(y1~x1)</code>	Add another series of points
<code>boxplot(y~x)</code>	Boxplot

Let's try some basic plotting. As a case study, we will use a simplified version of a dataset on predator-prey body mass ratios taken from the Ecological Archives of the ESA (Barnes *et al.* 2008, Ecology 89:881).



These data should be in your `Data` directory. Let's import the data:

```
> MyDF <- read.csv("../Data/EcolArchives-E089-51-D1.csv")
> dim(MyDF)
[1] 34931    15
> MyDF$
MyDF$Record.number      MyDF$Predator.mass
MyDF$In.refID           MyDF$Prey
MyDF$IndividualID       MyDF$Prey.common.name
MyDF$Predator            MyDF$Prey.taxon
MyDF$Predator.common.name MyDF$Prey.mass
MyDF$Predator.taxon      MyDF$Prey.mass.unit
MyDF$Predator.lifestage  MyDF$Location
MyDF$Type.of.feeding.interaction
```

2.1.1 Scatter Plot

Let's start by plotting Predator mass vs. Prey mass:

```
> plot(MyDF$Predator.mass, MyDF$Prey.mass)
```

That doesn't look very nice! Let's try taking logarithms (why?).

```
> plot(log(MyDF$Predator.mass), log(MyDF$Prey.mass))
```

We can change almost any aspect of the resulting graph; let's change the symbols by specifying the plot characters using `pch`:

```
> plot(log(MyDF$Predator.mass), log(MyDF$Prey.mass), pch=20) # Change marker
> plot(log(MyDF$Predator.mass), log(MyDF$Prey.mass), pch=20,
       xlab = "Predator Mass (kg)", ylab = "Prey Mass (kg)") # Add labels
```

2.1.2 Histograms

Now plot a histogram of Predator body masses:

```
> hist(MyDF$Predator.mass)
> hist(log(MyDF$Predator.mass),
       xlab = "Predator Mass (kg)", ylab = "Count") # labels
> hist(log(MyDF$Predator.mass), xlab="Predator Mass (kg)", ylab="Count",
       col = "lightblue", border = "pink") # Change bar and borders colors
```

2.1.3 Subplots

We can also plot both predator and prey body masses in different sub-plots using `par`

```
> par(mfcol=c(2,1), lwd = 1.5) #initialize multi-paneled plot
> par(mfg = c(1,1))
> hist(log(MyDF$Predator.mass),
       xlab = "Predator Mass (kg)", ylab = "Count",
       col = "lightblue", border = "pink",
       main = 'Predator') # Add title
> par(mfg = c(2,1));
> hist(log(MyDF$Prey.mass),
       xlab="Prey Mass (kg)", ylab="Count",
       col = "lightgreen", border = "pink",
       main = 'prey')
```

2.1.4 Overlaying plots

Better still, we would like to see if the predator mass and prey mass distributions are similar by overlaying them.

```
> hist(log(MyDF$Predator.mass), # Predator histogram
       xlab="Body Mass (kg)", ylab="Count",
       col = rgb(1, 0, 0, 0.5), # Note 'rgb'
       main = "Overlay")
> hist(log(MyDF$Prey.mass), col = rgb(0, 0, 1, 0.5), add = T) # Plot prey
> legend('topleft', c('Predators', 'Prey'), # Add legend
       fill=c(rgb(1, 0, 0, 0.5), rgb(0, 0, 1, 0.5))) # Define legend colors
```

2.1.5 Exercise

We can do a lot more beautification! As an exercise, try adjusting the bin widths to make them same for the predator and prey, and making the x and y labels larger and in boldface.

2.1.6 Saving your graphics

And you can also save the figure in a vector graphics format (important to learn to do this!). PDF is a good option:

```
> pdf("../Results/Pred_Prey_Overlay.pdf", # Open blank pdf page
      11.7, 8.3) # These numbers are page dimensions in inches
> hist(log(MyDF$Predator.mass), # Plot predator histogram (note 'rgb')
      xlab="Body Mass (kg)", ylab="Count",
      col = rgb(1, 0, 0, 0.5),
      main = "Overlay")
> hist(log(MyDF$Prey.mass), # Plot prey weights
      col = rgb(0, 0, 1, 0.5),
      add = T) # Add to same plot = TRUE
> legend('topleft',c('Predators','Prey'), # Add legend
      fill=c(rgb(1, 0, 0, 0.5), rgb(0, 0, 1, 0.5)))
> dev.off()
```

You can also try other graphic output formats. For example, `png()` (a raster format) instead of `pdf()`.

2.1.7 Boxplots

Now, let's try plotting boxplots instead of histograms. These are useful for getting a visual summary of your data :

```
> boxplot(log(MyDF$Predator.mass) ~ MyDF$Location, # Why the tilde?
      xlab = "Location", ylab = "Predator Mass",
      main = "Predator mass by location")
```

That's a lot of locations! You will need an appropriately wide plot to see all the boxplots adequately. Let's try boxplots by feeding interaction type:

```
> boxplot(log(MyDF$Predator.mass) ~ MyDF$Type.of.feeding.interaction,
      xlab = "Location", ylab = "Predator Mass",
      main = "Predator mass by feeding interaction type")
```

2.1.8 Practical 3.1

You can also make lattice graphs to avoid the somewhat laborious `par()` approach above. For this, you will need:

```
> library(lattice)
```

A lattice plot of the above data for predator mass could look like Fig. 2.1.8 (as a density plot). This was generated using (and printing to a pdf with particular dimensions):

```
> densityplot(~log(Predator.mass) | Type.of.feeding.interaction, data=MyDF)
```

Look up <http://www.statmethods.net/advgraphs/trellis.html> and the lattice package help.

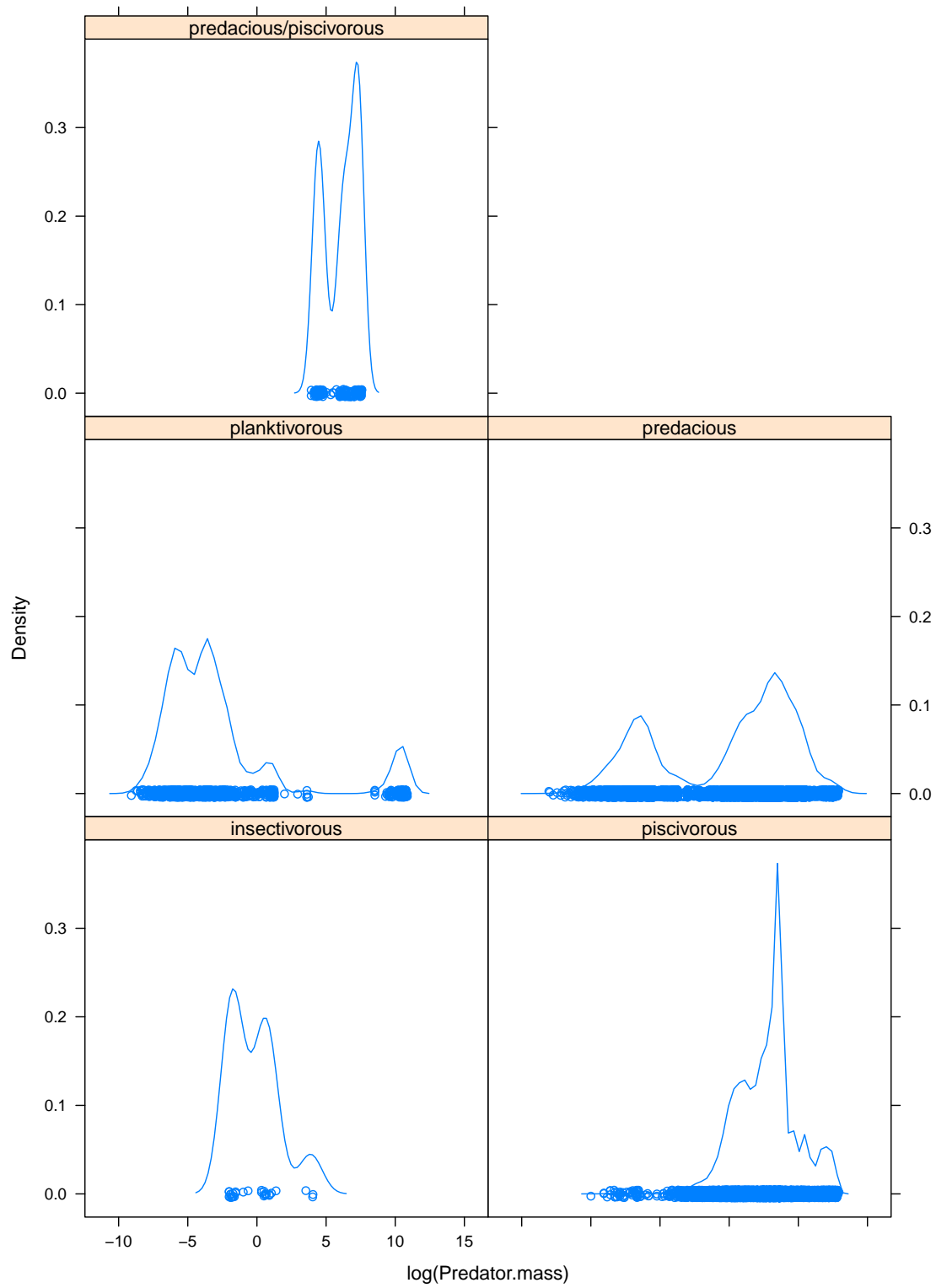


Figure 2.1: A lattice representation of the predator size data

In this practical, you will write script that draws and saves three lattice graphs by feeding interaction type: one of predator mass, one of prey mass and one of the size ratio of prey mass over predator mass. Note that you would want to use logarithms of masses (or mass-ratios) for all three plots. In addition, the script will calculate the mean and median predator mass, prey mass and predator-prey size-ratios to a csv file. The workflow would be:

- ★ Write a script file called `PP_Lattice.R` and save it in the `Code` directory — sourcing or running this script should result in three files called `Pred_Lattice.pdf`, `Prey_Lattice.pdf`, and `SizeRatio_Lattice.pdf` being saved in the `Results` directory (the names are self-explanatory, I hope).
- ★ In addition, the script should calculate the mean log predator mass, prey mass, and predator-prey size ratio, and save it as a single csv output table called `PP_Results.csv` to the `Results` directory. The table should have appropriate headers (e.g., Feeding type, mean, median). (Hint: you will have to initialize a new dataframe in the script to first store the calculations)
- ★ The script should be self-sufficient and not need any external inputs — it should import the above predator-prey dataset from the appropriate directory, and save the graphic plots to the appropriate directory (Hint: use relative paths).

2.2 Publication-quality figures in R

R can produce beautiful graphics, but it takes a lot of work to obtain the desired result. This is because the starting point is pretty much a “bare” plot, and adding features commonly required for publication-grade figures (legends, statistics, regressions, etc.) can be quite involved.

Moreover, it is very difficult to switch from one representation of the data to another (i.e., from boxplots to scatterplots), or to plot several dataset together. To overcome these issues, the R package `ggplot2` is very powerful. It can be used to produce truly high-quality graphics for papers, theses and reports. *One thing to note though is that at present, ggplot2 cannot be used to create 3D graphs or mosaic plots.*

`ggplot2` differs from other approaches as it attempts to provide a “grammar” for graphics in which each layer is the equivalent of a verb, subject etc. and a plot is the equivalent of a sentence. All graphs start with a layer showing the data, other layers and commands are added to modify the plot.

For a complete reference, please see the book “`ggplot2`: Elegant Graphics for Data Analysis”, by H. Wickham. Also, the website `ggplot2.org` a great resource. To install `ggplot2`, open a session of R and type (launch R using `sudo R` in Linux first):

```
> install.packages("ggplot2")
> install.packages("reshape") #A handy additional package
```

2.2.1 Basic graphs with `qplot`

`qplot` stands for quick plot, and is the basic plotting function provided by `ggplot2`. It can be used to quickly produce graphics for exploratory data analysis, and as a base for more complex graphics.

In `ggplot2`, it is necessary to use data frames to store the data. Again we can start plotting the `Predator.mass` vs `Prey.mass`.

```
> require(ggplot2) ## Load the package
```

```

Loading required package: ggplot2
> qplot(Prey.mass, Predator.mass, data = MyDF)

```

Again, let's take logarithms and plot:

```

> qplot(log(Prey.mass), log(Predator.mass), data = MyDF)

```

Now, color the points according to the type of feeding interaction:

```

> qplot(log(Prey.mass), log(Predator.mass), data = MyDF,
  colour = Type.of.feeding.interaction)

```

The same as above, but changing the shape:

```

> qplot(log(Prey.mass), log(Predator.mass), data = MyDF,
  shape = Type.of.feeding.interaction)

```

To manually set a color or a shape, you have to use `I()` (meaning “Identity”):

```

> qplot(log(Prey.mass), log(Predator.mass),
  data = MyDF, colour = I("red"))
> qplot(log(Prey.mass), log(Predator.mass),
  data = MyDF, shape = I(3))

```

Because there are so many points, we can make them semi-transparent using `alpha` so that the overlaps can be seen:

```

> qplot(log(Prey.mass), log(Predator.mass), data = MyDF,
  colour = Type.of.feeding.interaction, alpha = I(.5))

```

Now add a smoother to the points.

```

> qplot(log(Prey.mass), log(Predator.mass), data = MyDF,
  geom = c("point", "smooth"))

```

If we want to have a linear regression, we can specify the method `lm`:

```

> qplot(log(Prey.mass), log(Predator.mass), data = MyDF,
  geom = c("point", "smooth"), method = "lm")

```

We can add a smoother for each type of interaction:

```

> qplot(log(Prey.mass), log(Predator.mass), data = MyDF,
  geom = c("point", "smooth"), method = "lm",
  colour = Type.of.feeding.interaction)

```

To extend the lines to the full range, use `fullrange = TRUE`:

```

> qplot(log(Prey.mass), log(Predator.mass), data = MyDF,
  geom = c("point", "smooth"), method = "lm",
  colour = Type.of.feeding.interaction,
  fullrange = TRUE)

```

Now we want to see how the ratio between prey and predator mass changes according to the type of interaction:

```
> qplot(Type.of.feeding.interaction,
        log(Prey.mass/Predator.mass), data = MyDF)
```

Because there are so many points, we can “jitter” them to get a better idea of the spread:

```
> qplot(Type.of.feeding.interaction,
        log(Prey.mass/Predator.mass), data = MyDF,
        geom = "jitter")
```

Or we can draw a boxplot of the data (note the `geom` argument):

```
> qplot(Type.of.feeding.interaction,
        log(Prey.mass/Predator.mass), data = MyDF,
        geom = "boxplot")
```

Now let’s draw an histogram of predator-prey mass ratios:

```
> qplot(log(Prey.mass/Predator.mass), data = MyDF,
        geom = "histogram")
```

Color the histogram according to the interaction type:

```
> qplot(log(Prey.mass/Predator.mass), data = MyDF,
        geom = "histogram",
        fill = Type.of.feeding.interaction)
```

You may want to define binwidth (in units of x axis):

```
> qplot(log(Prey.mass/Predator.mass), data = MyDF,
        geom = "histogram",
        fill = Type.of.feeding.interaction,
        binwidth = 1)
```

To make it easier to read, we can plot the smoothed density of the data:

```
> qplot(log(Prey.mass/Predator.mass), data = MyDF,
        geom = "density", fill = Type.of.feeding.interaction)
```

And you can make the densities transparent so that the overlaps are visible:

```
> qplot(log(Prey.mass/Predator.mass), data = MyDF,
        geom = "density", fill = Type.of.feeding.interaction, alpha =
        I(0.5))
```

Or using `colour` instead of `fill` draws only the edge of the curve:

```
> qplot(log(Prey.mass/Predator.mass), data = MyDF,
        geom = "density", colour = Type.of.feeding.interaction)
```


Similarly, `geom = \bar` produces a barplot, `geom = \line` a series of points joined by a line, etc.

An alternative way of displaying data belonging to different classes is using “faceting”. A simple example:

```
> qplot(log(Prey.mass/Predator.mass),
  facets = Type.of.feeding.interaction ~ .,
  data = MyDF, geom = "density")
```

A more elegant way of drawing logarithmic quantities is to set the axes to be logarithmic:

```
> qplot(Prey.mass, Predator.mass, data = MyDF, log="xy")
```

Let’s add a title and labels:

```
> qplot(Prey.mass, Predator.mass, data = MyDF, log="xy",
  main = "Relation between predator and prey mass",
  xlab = "log(Prey mass) (g)",
  ylab = "log(Predator mass) (g)")
```

Adding `+ theme_bw()` makes it suitable for black and white printing.

```
> qplot(Prey.mass, Predator.mass, data = MyDF, log="xy",
  main = "Relation between predator and prey mass",
  xlab = "Prey mass (g)",
  ylab = "Predator mass (g)") + theme_bw()
```

Finally, let’s save a pdf file of the figure (same approach as we used before):

```
> pdf("../Results/MyFirst-ggplot2-Figure.pdf")
> print(qplot(Prey.mass, Predator.mass, data = MyDF, log="xy",
  main = "Relation between predator and prey mass",
  xlab = "log(Prey mass) (g)",
  ylab = "log(Predator mass) (g)") + theme_bw())
> dev.off()
```

Using `print` ensures that the whole command is kept together and that you can use the command in a script.

Other important options to keep in mind:

<code>xlim</code>	limits for x axis: <code>xlim = c(0,12)</code>
<code>ylim</code>	limits for y axis
<code>log</code>	log transform variable <code>log = \x</code> , <code>log = \y</code> , <code>log = \xy</code>
<code>main</code>	title of the plot <code>main = \My Graph</code>
<code>xlab</code>	x-axis label
<code>ylab</code>	y-axis label
<code>asp</code>	aspect ratio <code>asp = 2</code> , <code>asp = 0.5</code>
<code>margins</code>	whether or not margins will be displayed

2.2.2 Various geom

`geom` Specifies the geometric objects that define the graph type. The `geom` option is expressed as a character vector with one or more entries. `geom` values include “point”, “smooth”, “boxplot”,

“line”, “histogram”, “density”, “bar”, and “jitter”. Try the following:

```
# load the package
require(ggplot2)

# load the data
MyDF <- as.data.frame(
  read.csv("../Data/EcolArchives-E089-51-D1.csv"))

# barplot
qplot(Predator.lifestage,
      data = MyDF, geom = "bar")

# boxplot
qplot(Predator.lifestage, log(Prey.mass),
      data = MyDF, geom = "boxplot")

# density
qplot(log(Predator.mass),
      data = MyDF, geom = "density")

# histogram
qplot(log(Predator.mass),
      data = MyDF, geom = "histogram")

# scatterplot
qplot(log(Predator.mass), log(Prey.mass),
      data = MyDF, geom = "point")

# smooth
qplot(log(Predator.mass), log(Prey.mass),
      data = MyDF, geom = "smooth")

qplot(log(Predator.mass), log(Prey.mass),
      data = MyDF, geom = "smooth", method = "lm")
```

2.2.3 Practical 3.2

In this practical, you will write script that draws and saves a pdf file of Fig. 2.2.3, and writes the accompanying regression results to a formatted table in csv. Note that the plots show that the analysis must be subsetted by the `Predator.lifestage` field of the dataset. The guidelines are:

- ★ Write a script file called `PP_Regress.R` and save it in the Code directory — sourcing or running this script should result in one pdf file containing (Fig. 2.2.3) being saved in the Results directory (Hint: Use the `print()` command to write to the pdf).
- ★ In addition, the script should calculate the regression results visible in Fig. 2.2.3 and save it to a csv delimited table called (`PP_Regress.Results.csv`), in the Results directory. (Hint: you will have to initialize a new dataframe in the script to first store the calculations and then `write.csv()` or `write.table()` it.)
- ★ The regression results should include the following with appropriate headers (e.g., slope, intercept, etc, in each Feeding type \times life stage category): regression slope, regression intercept, R^2 , F-statistic value, and p-value of the overall regression (Hint: Review Practical 8 of the Stats week).
- ★ The script should be self-sufficient and not need any external inputs — it should import the above predator-prey dataset from the appropriate directory, and save the graphic plots to the

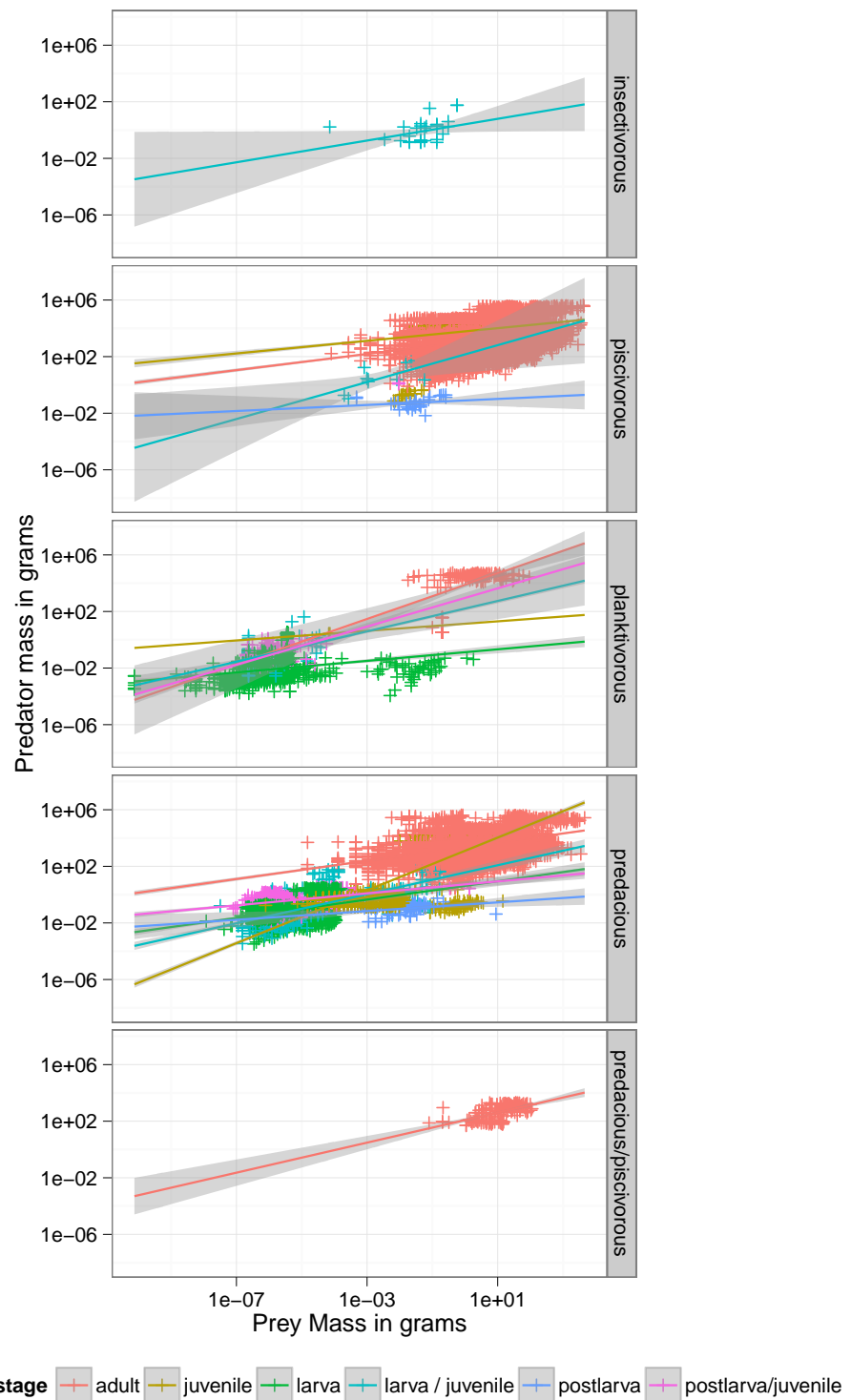


Figure 2.2: Write a script that generates this figure.

appropriate directory (Hint: use relative paths). I should be able to `source` it without errors.

Extra Credit: Do the same as above, but the analysis this time should be separate by the dataset's `Location` field. This is a substantial extra effort, so I do think it deserves substantial extra credit!

2.2.4 Advanced plotting: `ggplot`

The command `ggplot` allows you to use only a single dataset and a single set of “aesthetics” (x, y, etc.). To make full use of `ggplot2`, we need to use the command `ggplot`. We need:

- The data to be plotted, in a data frame;
- Aesthetics mappings, specifying which variables we want to plot, and how;
- The `geom`, defining how to draw the data;
- (Optionally) some `stat` that transform the data or perform statistics using the data.

To start a graph, we can specify the data and the aesthetics:

```
> p <- ggplot(MyDF, aes(x = log(Predator.mass),
                        y = log(Prey.mass),
                        colour = Type.of.feeding.interaction ))
```

Now try to plot the graph:

```
> p
Error: No layers in plot
```

In fact, we have to specify a geometry in order to see the graph:

```
> p + geom_point()
```

We can use the “plus” sign to concatenate different commands:

```
> p <- ggplot(MyDF, aes(x = log(Predator.mass),
                        y = log(Prey.mass),
                        colour = Type.of.feeding.interaction ))
> q <- p + geom_point(size=I(2), shape=I(10)) + theme_bw()
> q
```

Let's remove the legend:

```
> q + opts(legend.position = "none")
```

2.2.5 Case study 1: plotting a matrix

In this section we will plot a matrix of random values taken from a normal distribution $\mathcal{N}[0,1]$. Our goal is to produce the plot in Figure 2.3. Because we want to plot a matrix, and `ggplot2` accepts only dataframes, we use the package `reshape` that can “melt” a matrix into a dataframe:

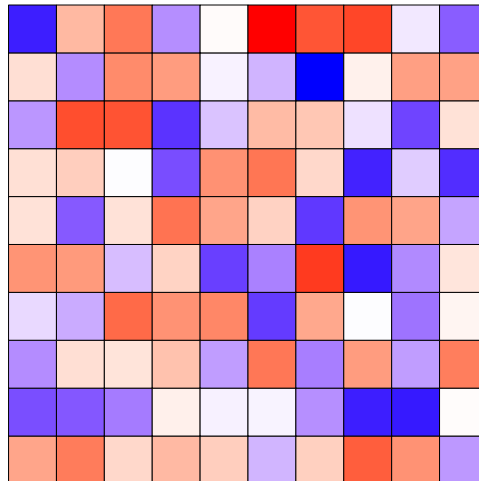


Figure 2.3: Random matrix with values sampled from uniform distribution.

```
require(ggplot2)
require(reshape)

GenerateMatrix <- function(N){
  M <- matrix(runif(N * N), N, N)
  return(M)
}

> M <- GenerateMatrix(10)

> M[1:3, 1:3]
      [,1]      [,2]      [,3]
[1,] 0.2700254 0.8686728 0.7365857
[2,] 0.1744879 0.8488169 0.4165879
[3,] 0.3980783 0.7727821 0.4271121

> Melt <- melt(M)

> Melt[1:4,]
      X1 X2      value
1  1  1 0.2700254
2  2  1 0.1744879
3  3  1 0.3980783
4  4  1 0.3196671

> ggplot(Melt, aes(X1, X2, fill = value)) + geom_tile()

# adding a black line dividing cells
> p <- ggplot(Melt, aes(X1, X2, fill = value))
> p <- p + geom_tile(colour = "black")

# removing the legend
> q <- p + opts(legend.position = "none")

# removing all the rest
```

```

> q <- p + opts(legend.position = "none",
  panel.background = theme_blank(),
  axis.ticks = theme_blank(),
  panel.grid.major=theme_blank(),
  panel.grid.minor=theme_blank(),
  axis.text.x = theme_blank(),
  axis.title.x=theme_blank(),
  axis.text.y = theme_blank(),
  axis.title.y=theme_blank())

# exploring the colors
> q + scale_fill_continuous(low = "yellow",
  high = "darkgreen")
> q + scale_fill_gradient2()
> q + scale_fill_gradientn(colours = grey.colors(10))
> q + scale_fill_gradientn(colours = rainbow(10))
> q + scale_fill_gradientn(colours =
  c("red", "white", "blue"))

```

2.2.6 Case study 2: plotting two dataframes

According to Girko's circular law, the eigenvalues of a matrix M of size $N \times N$ are approximately contained in a circle in the complex plane with radius \sqrt{N} . We are going to draw a simulation displaying this result (Figure 2.4).

```

require(ggplot2)

# function that returns an ellipse
build_ellipse <- function(hradius, vradius){
  npoints = 250
  a <- seq(0, 2 * pi, length = npoints + 1)
  x <- hradius * cos(a)
  y <- vradius * sin(a)
  return(data.frame(x = x, y = y))
}

# Size of the matrix
N <- 250
# Build the matrix
M <- matrix(rnorm(N * N), N, N)
# Find the eigenvalues
eigvals <- eigen(M)$values
# Build a dataframe
eigDF <- data.frame("Real" = Re(eigvals),
  "Imaginary" = Im(eigvals))

# The radius of the circle is sqrt(N)
my_radius <- sqrt(N)
# Ellipse dataframe
ellDF <- build_ellipse(my_radius, my_radius)
# rename the columns
names(ellDF) <- c("Real", "Imaginary")

# Now the plotting:
# plot the eigenvalues
p <- ggplot(eigDF, aes(x = Real, y = Imaginary))
p <- p +
  geom_point(shape = I(3)) +

```

```

opts(legend.position = "none")

# now add the vertical and horizontal line
p <- p + geom_hline(aes(intercept = 0))
p <- p + geom_vline(aes(intercept = 0))

# finally, add the ellipse
p <- p + geom_polygon(data = ellDF,
                      aes(x = Real,
                          y = Imaginary,
                          alpha = 1/20,
                          fill = "red"))

pdf("Girko.pdf")
print(p)
dev.off()

```

2.2.7 Case study 3: annotating the plot

In the plot in Figure 2.5, we use the geometry “text” to annotate the plot.

```

require(ggplot2)

filename <- "Results.txt"
a <- read.table(filename, header = TRUE)
# here's how the data looks like
print(a[1:3,])
print(a[90:95,])

# append a col of zeros
a$ymin <- rep(0, dim(a)[1])

# print the first linerange
p <- ggplot(a)
p <- p + geom_linerange(data = a, aes(
  x = x,
  ymin = ymin,
  ymax = y1,
  size = (0.5)
),
  colour = "#E69F00",
  alpha = 1/2, show_guide = FALSE)

# print the second linerange
p <- p + geom_linerange(data = a, aes(
  x = x,
  ymin = ymin,
  ymax = y2,
  size = (0.5)
),
  colour = "#56B4E9",
  alpha = 1/2, show_guide = FALSE)

# print the third linerange
p <- p + geom_linerange(data = a, aes(
  x = x,
  ymin = ymin,
  ymax = y3,
  size = (0.5)

```

```

    ),
    colour = "#D55E00",
    alpha = 1/2, show_guide = FALSE)

# annotate the plot with labels
p <- p + geom_text(data = a,
                  aes(x = x, y = -500, label = Label))

# now set the axis labels,
# remove the legend, prepare for bw printing
p <- p + scale_x_continuous("My x axis",
                          breaks = seq(3, 5, by = 0.05)
                          ) +
  scale_y_continuous("My y axis") + theme_bw() +
  opts(legend.position = "none")

# Finally, print in a pdf
pdf("MyBars.pdf", width = 12, height = 6)
print(p)
dev.off()

```

2.2.8 Case study 4: mathematical display

In Figure 2.6, you can see the mathematical annotation of the axis and on the plot.

```

require(ggplot2)

# create an "ideal" linear regression data!
x <- seq(0, 100, by = 0.1)
y <- -4. + 0.25 * x +
  rnorm(length(x), mean = 0., sd = 2.5)

# now a dataframe
my_data <- data.frame(x = x, y = y)

# perform a linear regression
my_lm <- summary(lm(y ~ x, data = my_data))

# plot the data
p <- ggplot(my_data, aes(x = x, y = y,
                        colour = abs(my_lm$residual)))
  ) +
  geom_point() +
  scale_colour_gradient(low = "black", high = "red") +
  opts(legend.position = "none") +
  scale_x_continuous(
    expression(alpha^2 * pi / beta * sqrt(Theta)))

# add the regression line
p <- p + geom_abline(
  intercept = my_lm$coefficients[1][1],
  slope = my_lm$coefficients[2][1],
  colour = "red")

# throw some math on the plot
p <- p + geom_text(aes(x = 60, y = 0,
                      label = "sqrt(alpha) * 2* pi"),
                  parse = TRUE, size = 6,
                  colour = "blue")

```

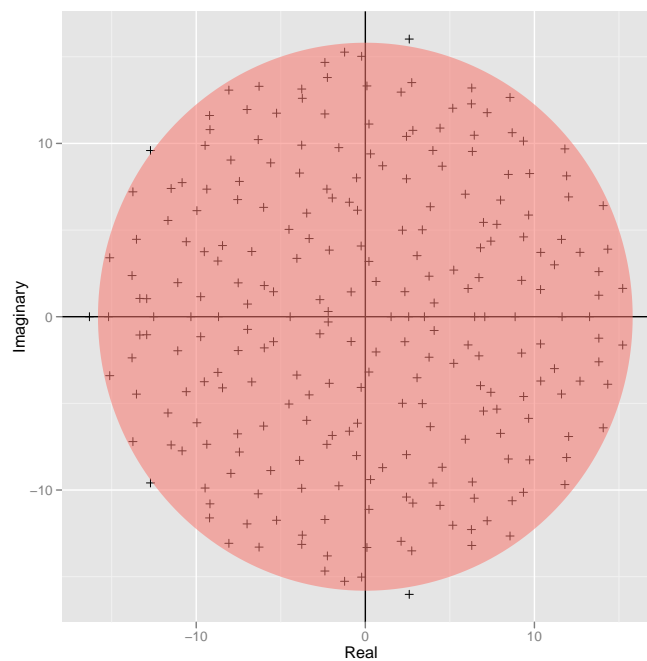



Figure 2.4: Girko's circular law.

```
# print in a pdf
pdf("MyLinReg.pdf")
print(p)
dev.off()
```

2.3 Readings

- The classic Tufte www.edwardtufte.com/tufte/books_vdqi (btw, check out what Tufte thinks of PowerPoint!)
Available in the Central Library, I have also added extracts and a related book in pdf (on Blackboard)
- Rolandi et al. "A Brief Guide to Designing Effective Figures for the Scientific Paper", doi:10.1002/adma.201102518 (on Blackboard)
- Lauren et al. "Graphs, Tables, and Figures in Scientific Publications: The Good, the Bad, and How Not to Be the Latter", doi:10.1016/j.jhsa.2011.12.041 (on Blackboard)
- Effective scientific illustrations: www.labtimes.org/labtimes/issues/lt2008/lt05/lt_2008_05_52_53.pdf (on Blackboard)
- <https://web.archive.org/web/20120310121708/http://addictedtor.free.fr/graphiques/thumbs.php>
- Make xkcd style graphs in R: <http://xkcd.r-forge.r-project.org/>

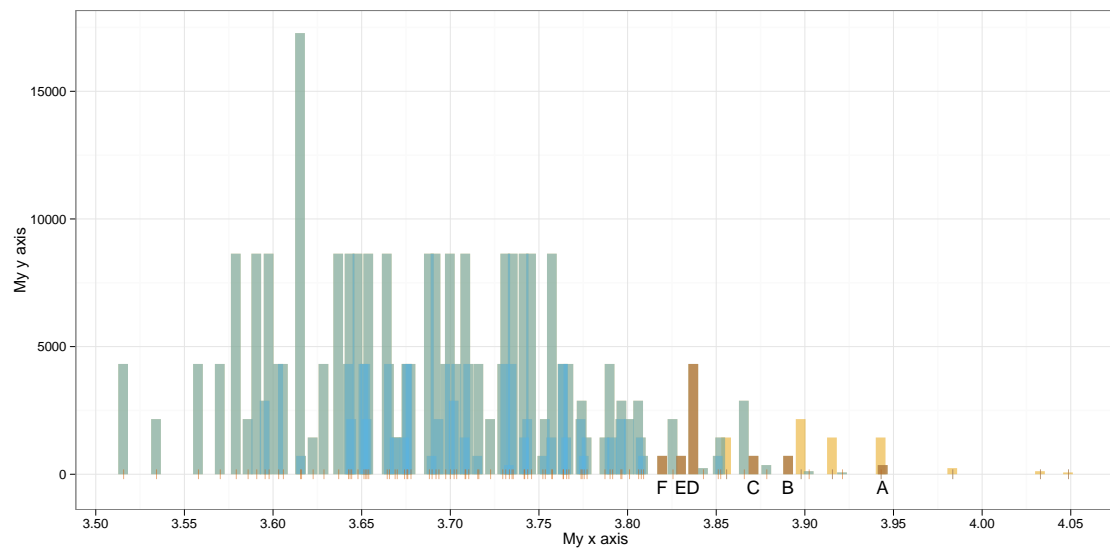


Figure 2.5: Overlay of three lineranges and a text geometry.

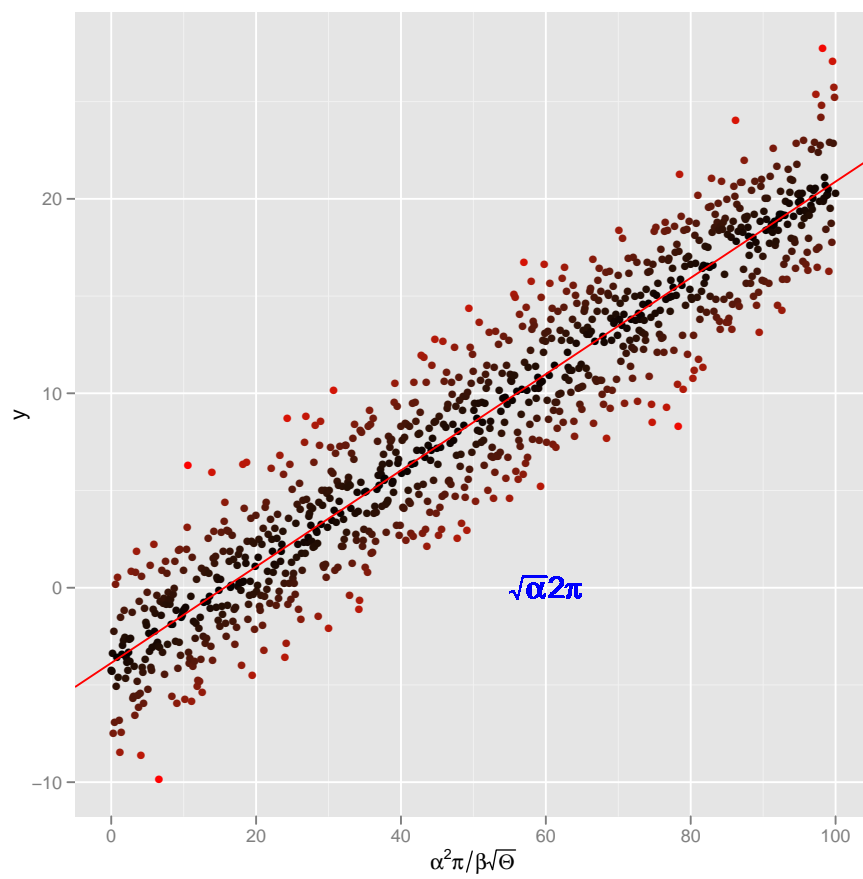


Figure 2.6: Linear regression with colors expressing residuals and mathematical annotations.

Chapter 3

Advanced topics in R

3.1 Vectorization

R is very slow at running cycles (`for` and `while` loops). This is because R is a “nimble” language: at execution time R does not know what you’re going to perform until it “reads” the code to perform. Compiled languages such as C, know exactly what the flow of the program is, as the code is compiled before execution. As a metaphor, C is a musician playing a score she has seen before – optimizing each passage, while R is playing it “a prima vista” (i.e., at first sight).

Hence, in R you should try to avoid loops like the plague. In practical terms, sometimes it is much easier to throw in a `for` loop, and then optimize the code to avoid the loop if the running time is not satisfactory. R has several functions that can operate on entire vectors and matrices.

- ★ For example, type (save in Code) in `Vectorize1.R` and run it (it sums all elements of a matrix):

```
M <- matrix(runif(1000000),1000,1000)

SumAllElements <- function(M) {
  Dimensions <- dim(M)
  Tot <- 0
  for (i in 1:Dimensions[1]){
    for (j in 1:Dimensions[2]){
      Tot <- Tot + M[i,j]
    }
  }
  return (Tot)
}

## This on my computer takes about 1 sec
print(system.time(SumAllElements(M)))
## While this takes about 0.01 sec
print(system.time(sum(M)))
```

Both approaches are correct, and will give you the right answer. However, one is 100 times faster than the other!

Fortunately, R offers several ways of avoiding loops. Here are the main ones:

```
## apply:
# applying the same function to rows/columns of a matrix

## Build a random matrix
```

```

M <- matrix(rnorm(100), 10, 10)
## Take the mean of each row
RowMeans <- apply(M, 1, mean)
print (RowMeans)
## Now the variance
RowVars <- apply(M, 1, var)
print (RowVars)
## By column
ColMeans <- apply(M, 2, mean)
print (ColMeans)

## You can use it to define your own functions
## (What does this function do?)
SomeOperation <- function(v) {
  if (sum(v) > 0) {
    return (v * 100)
  }
  return (v)
}
print (apply(M, 1, SomeOperation))

## by:
## apply the function to a dataframe, using some factor
## to define the subsets

## import some data
attach(iris)
print (iris)

## use colMeans (as it is better for dataframes)
by(iris[,1:2], iris$Species, colMeans)
by(iris[,1:2], iris$Petal.Width, colMeans)

## There are many other methods: lapply, sapply, eapply, etc.
## Each is best for a given data type (lapply -> lists)

## replicate:
## this is quite useful to avoid a loop for function that typically
## involve random number generation
print(replicate(10, runif(5)))

```

3.2 Breaking out of loops

Often it is useful (or necessary) to break out of a loop when some condition is met. Use `break` (like in pretty much any other programming language, like `python`) in situations when you cannot set a target number of iterations, as you would with a `while` loop (Chapter 1). Try this (type into `break.R` and save in Code):

```

i <- 0 #Initialize i
while(i < Inf) {
  if (i == 10) {
    break # Break out of the while loop!
  }
  else {
    cat("i equals ", i, "\n")
    i <- i + 1 # Update i
  }
}

```

3.3 Using next

You can also skip to next iteration of a loop. Both `next` and `break` can be used within other loops (`while`, `for`). Try this (type into `next.R` and save in Code (what does this script do?)):

```
for (i in 1:10) {
  if ((i %% 2) == 0)
    next # pass to next iteration of loop
  print(i)
}
```

Reminder: Indent your code! Indentation helps you see the flow of the logic, rather than flattened version, which is hard for you and everybody else to read. I recommend using the `tab` key to indent.

3.4 “Catching” errors

Often, you don’t know if a simulation or a R function will work on a particular data or variable, or a value of a variable (can happen in many stats functions). Rather than having R throw you out of the code, you would rather catch the error and keep going. This can be done using `try`. Type the following into `try.R` and save in Code (what does this script do?):

```
## run a simulation that involves sampling from a population

x <- stats::rnorm(50)
doit <- function(x){
  x <- sample(x, replace = TRUE)
  if(length(unique(x)) > 30) {
    mean(x)
  }
  else {
    stop("too few unique points")
  }
}

## Try using "try" with vectorization:
result <- lapply(1:100, function(i) try(doit(x), FALSE))

## Or using a for loop:
res <- vector("list", 100) #Preallocate/Initialize
for(i in 1:100) {
  result[[i]] <- try(doit(x), FALSE)
}
```

Note the functions `sample` and `stop` in the above script. Also check out `tryCatch`.

3.5 Generating Random Numbers

Computers don’t really generate mathematically random numbers, but instead a sequence of numbers that are close to random: “pseudo-random numbers”. They are generated based on some iterative formula:

$$x_{new} = f(x_{old}) \mod N$$

where modulo operation provides the “remainder” division.

To generate the first random number, you need a **seed**. Setting the seed allows you to reliably generate the same sequence of numbers, which can be useful when debugging programs (next section).

R has many routines for generating random samples from various probability distributions — we have already used `runif()`, `rnorm()`. Try this:

```
> set.seed(1234567)
> rnorm(1)
0.1567038
```

What happened?! If this were truly a random number, how would everybody get the same answer? Now try `rnorm(10)` and compare the results with your neighbour. Thus “random” numbers generated in R and in any other software are in fact “deterministic”, but from a very complex formula that yields numbers with properties like random numbers.

Effectively, `rnorm` has an enormous list that it cycles through. The random seed starts the process, i.e., indicates where in the list to start. This is usually taken from the clock when you start R.

But why bother with this? Well, for debugging (next section). Bugs in code can be hard to find — harder still if you are generating random numbers, so repeat runs of your code may or may not all trigger the same behaviour. You can set the seed once at the beginning of the code — ensuring repeatability, retaining (pseudo) randomness. Once debugged, if you want, you can remove the set seed line.

3.6 Debugging

Indeed, as most of you must have already experienced by now, there can be frustrating, puzzling bugs in programs that lead to mysterious errors. Often, the error and warning messages you get are un-understandable, especially in R! Some useful debugging functions in R:

- Warnings vs Errors; converting warnings to errors: `stopifnot()` — a bit like `try`
- What to do when you get an error: `traceback()`
- Simple `print` commands in the right places can be useful for testing (but not strongly recommended)
- Use of `browser()` at key points in code — my favourite option (also look up `recover()`)
- `debug(fn)`, `undebg(fn)` : More technical approach to debugging — explore them

Let’s look at an example using `browser()`. `browser()` is handy because it will allow you to “single-step” through your code. Place it within your function at the point you want to examine (e.g.) local variables.

Here’s an example usage of `browser()` (type in `browse.R` and save in Code):

```
Exponential <- function(N0 = 1, r = 1, generations = 10)
{
  # Runs a simulation of exponential growth
  # Returns a vector of length generations

  N <- rep(NA, generations)      # Creates a vector of NA

  N[1] <- N0
```

```

for (t in 2:generations)
{
  N[t] <- N[t-1] * exp(r)
  browser()
}
return (N)
}
plot(Exponential(), type="l", main="Exponential growth")

```

Now, within the browser, you can enter expressions as normal, or you can use a few particularly useful debug commands:

- n: single-step
- c: exit browser and continue
- Q: exit browser and abort, return to top-level.

3.7 Practical 3.1

Non-CMEE students can ignore this one!

The Ricker model is a classic discrete population model which was introduced in 1954 by Ricker to model recruitment of stock in fisheries. It gives the expected number (or density) N_{t+1} of individuals in generation $t + 1$ as a function of the number of individuals in the previous generation t :

$$N_{t+1} = N_t e^{r(1 - \frac{N_t}{k})} \quad (3.1)$$

Here r is intrinsic growth rate and k as the carrying capacity of the environment. Try this script that runs it:

```

Ricker <- function(N0=1, r=1, K=10, generations=50)
{
  # Runs a simulation of the ricker model
  # Returns a vector of length generations

  N <- rep(NA, generations) # Creates a vector of NA

  N[1] <- N0
  for (t in 2:generations)
  {
    N[t] <- N[t-1] * exp(r*(1.0-(N[t-1]/K)))
  }
  return (N)
}
plot(Ricker(generations=10), type="l")

```

Now open and run the script `Vectorize2.R` (available on the bitbucket Git repository). This is the stochastic Ricker model (compare with the above script to see where the stochasticity (random error) enters. Now modify the script to complete the exercise given. CMEEs, As always, bring your functional code and data under version control!

3.8 Launching/Running R in batch mode

Often, you may want to run the final analysis without opening R in interactive mode. In in Mac or linux, you can do so by typing:

```
R CMD BATCH MyCode.R MyResults.Rout
```

This will create an `MyResults.Rout` file containing all the output. On Microsoft Windows, its more complicated — (change the path to `R.exe` and output file as needed:

```
\C:\Program Files\R\R-3.1.1\bin\R.exe" CMD BATCH --vanilla --slave  
\C:\PathToMyResults\Results\MyCode.R"
```

3.9 Readings

- See **An introduction to the Interactive Debugging Tools in R**, Roger D Peng for detailed usage. <http://www.biostat.jhsph.edu/~rpeng/docs/R-debug-tools.pdf>