

An introduction to R for modeling (lab 1)

Stephen Ellner*, modified by Ben Bolker†

September 7, 2013



Licensed under the Creative Commons attribution-noncommercial

license (<http://creativecommons.org/licenses/by-nc/3.0/>). Please

share & remix noncommercially, mentioning its origin.

Version: 2013-09-07 11:10:50

0 How to use this document

- These notes contain many sample calculations. It is important to do these yourself—**type them in at your keyboard and see what happens on your screen**—to get the feel of working in R.
- **Exercises** in the middle of a section should be done immediately when you get to them, and make sure you have them right before moving on. Some more challenging exercises (indicated by asterisks) appear at the end of some sections. These can be left until later, and may be assigned as homework.

These notes are based in part on course materials by former TAs Colleen Webb, Jonathan Rowell and Daniel Fink at Cornell, Professors Lou Gross (University of Tennessee) and Paul Fackler (NC State University), and on the book *Getting Started with Matlab* by Rudra Pratap (Oxford University Press). They also draw on the documentation supplied with R. They parallel, but go into more depth than, the chapter supplement for the book *Ecological Models and Data in R* by Ben Bolker (Princeton University Press).

You can find many other similar introductions scattered around the web, or in the “contributed documentation” section on the R web site (<http://cran.r-project.org/other-docs.html>). This particular version

*Ecology and Evolutionary Biology, Cornell University

†Department of Mathematics & Statistics and Biology, McMaster University

is limited (it has similar coverage to Sections 1 and 2 of the *Introduction to R* that comes with R).

1 What is R?

R is an object-oriented scripting language that combines

- a programming language called S, developed by John Chambers at Bell Labs, that can be used for numerical simulation of deterministic and stochastic dynamic models
- an extensive set of functions for classical and modern statistical data analysis and modeling
- graphics functions for visualizing data and model output
- a user interface with a few basic menus and extensive help facilities

R is an open source project, available for free download via the Web. Originally a research project in statistical computing (Ihaka and Gentleman, 1996), it is now managed by a development team that includes a number of well-regarded statisticians, and is widely used by statistical researchers (and a growing number of theoretical ecologists and ecological modelers) as a platform for making new methods available to users. A standard installation of R includes extensive documentation, including an introductory manual (≈ 100 pages) and a comprehensive reference manual (over 1000 pages). (There is a graphical front-end for parts of R, called “R commander” (`Rcmdr` for short), available at the R site, but we will not be using it in this class.)

1.1 Installing R on your computer: basics

If an up-to-date version of R is already installed on your computer (the current version is 3.0.1 as of this writing) you can skip this section.

The main source for R is the CRAN home page <http://cran.r-project.org>. You can get the source code, but most users will prefer a precompiled version. To get one of these from CRAN:

- go to <http://cran.r-project.org/mirrors.html> and find a mirror site that is geographically somewhat near you.
- Find the appropriate page for your operating system — when you get to the download section, go to `base` rather than `contrib`. Download the binary file (e.g. `base/R-x.y.z-win.exe` for Windows,

`R-x.y.z.pkg` for MacOS, where `x.y.z` is the version number). The binary files are fairly large (50 megabytes or so) — you will need to find a fast internet connection.

- Read and follow the instructions (which are pretty much “click on the icon”).
- In addition, you should install the “desktop” version of **RStudio** from <http://www.rstudio.com/ide/download/>

R should work well on any reasonably modern computer (although using versions of MacOS X before 10.3 will be a hassle). R moves quickly: if possible, you should make sure you have upgraded to the most recent version available, or at least that your version isn’t more than about a year old.

The standard distributions of R include several *packages*, user-contributed suites of add-on functions (unfortunately, the command to load a package into R is `library()`). These Notes use some packages that are not part of the standard distribution. In general, you can install additional packages from within R using the **Packages** menu, or the `install.packages` command. (See below.)

For Windows or MacOS, install R by launching the downloaded file and following the on-screen instructions. At the end you’ll have an R icon on your desktop that you can use to launch the program. If you use Linux or Unix you’ll probably be able to figure out what to do.

1.2 Starting R

(windows) Just click on the icon on your desktop, or in the **Start** menu (if you allowed the Setup program to make either or both of these). If you lose these shortcuts for some reason, you can search for the executable file `Rgui.exe` on your hard drive, which will probably be somewhere like `Program Files\R\R-x.y.z\bin\Rgui.exe`. If you are using **Rstudio**, just click on the icon.

1.3 Stopping R

When entering, always look for the exit.

Lebanese proverb

You can stop R from the **File** menu ((windows)), or you can stop it by typing `q()` at the command prompt (if you type `q` by itself, you will get

some confusing output which is actually R trying to tell you the definition of the `q` function; more on this later).

When you quit, R will ask you if you want to save the workspace (that is, all of the variables you have defined in this session); for now (and in general), say “no” in order to avoid clutter.

Should an R command seem to be stuck or take longer than you’re willing to wait, click on the stop sign on the menu bar or hit the **Escape** key (in Unix, type Control-C).

2 Interactive calculations

(windows) When you start R it opens the **console** window. The console has a few basic menus at the top; check them out on your own. The console is also where you enter commands for R to execute *interactively*, meaning that the command is executed and the result is displayed as soon as you hit the **Enter** key. For example, at the command prompt `>`, type in `2+2` and hit **Enter**; you will see

```
> 2+2  
[1] 4
```

(From now on we will leave out the command prompt (`>`) and comment out the results (`##`) to make it easier to cut and paste from this document.)

To do anything complicated, you have to store the results from calculations by *assigning* them to variables, using `=` or `<-`. For example:

```
a <- 2+2
```

R automatically creates the variable `a` and stores the result (4) in it, but it doesn’t print anything. This may seem strange, but you’ll often be creating and manipulating huge sets of data that would fill many screens, so the default is to skip printing the results. To ask R to print the value, just type the variable name by itself at the command prompt:

```
a  
## [1] 4
```

(the `[1]` at the beginning of the line is just R printing an index of element numbers; if you print a result that displays on multiple lines, R will put an

index at the beginning of each line. `print(a)` also works to print the value of a variable.) By default, a variable created this way is a *vector*, and it is *numeric* because we gave R a number rather than some other type of data (e.g. a character string like "pxqr"). In this case `a` is a numeric vector of length 1, which acts just like a number.

You could also type `a <- 2+2; a`, using a semicolon to put two or more commands on a single line. Conversely, you can break lines *anywhere that R can tell you haven't finished your command* and R will give you a "continuation" prompt (+) to let you know that it doesn't think you're finished yet: try typing

```
a=3*(4+ [Enter]
5)
```

to see what happens (you will sometimes see the continuation prompt when you don't expect it, e.g. if you forget to close parentheses). If you get stuck continuing a command you don't want—e.g. you opened the wrong parentheses—just hit the **Escape** key or the stop icon in the menu bar to get out.

Variable names in R must begin with a letter, followed by letters or numbers. You can break up long names with a period, as in `very.long.variable.number.3`, or an underscore (`_`), but you can't use blank spaces in variable names (or at least it's not worth the trouble). Variable names in R are case sensitive, so `Abc` and `abc` are different variables. Make variable names long enough to remember, short enough to type. `N.per.ha` or `pop.density` are better than `x` and `y` (too short) or `available.nitrogen.per.hectare` (too long). Avoid `c`, `l`, `q`, `t`, `C`, `D`, `F`, `I`, `O`, and `T`, which are either built-in R functions or hard to tell apart from other symbols (e.g. `l` (lower-case L) vs. `1` (numeral 1)).

R does calculations with variables as if they were numbers. It uses `+`, `-`, `*`, `/`, and `^` for addition, subtraction, multiplication, division and exponentiation, respectively. For example:

```
x <- 5
y <- 2
z1 <- x*y ## no output
z2 <- x/y ## no output
z3 <- x^y ## no output
z2

## [1] 2.5
```

```
z3
## [1] 25
```

Even though R did not display the values of `x` and `y`, it “remembers” that it assigned values to them. Type `x`; `y` to display the values.

You can retrieve and edit previous commands. The up-arrow (`↑`) key recalls previous commands to the prompt. For example, you can bring back the second-to-last command and edit it to

```
z3 <- 2*x^y
```

(experiment with the other arrow keys (`↓`, `→`, `←`), `Home` and `End` keys too). This will save you many hours in the long run.

You can combine several operations in one calculation:

```
A <- 3
C <- (A+2*sqrt(A))/(A+5*sqrt(A)); C
## [1] 0.5544
```

Parentheses specify the order of operations. The command

```
C <- A+2*sqrt(A)/A+5*sqrt(A)
```

is not the same as the one above; rather, it is equivalent to `C <- A + 2*(sqrt(A)/A) + 5*sqrt(A)`.

The default order of operations is: (1) parentheses; (2) exponentiation, or powers, (3) multiplication and division, (4) addition and subtraction (“**p**retty **p**lease **e**xecute **m**y **d**ear **A**unt **S**ally”).

```
b <- 12-4/2^3    gives 12 - 4/8 = 12 - 0.5 = 11.5
b <- (12-4)/2^3  gives 8/8 = 1
b <- -1^2        gives -(1^2) = -1
b <- (-1)^2      gives 1
```

In complicated expressions you might start off by *using parentheses to specify explicitly what you want*, such as `b <- 12 - (4/(2^3))` or at least `b <- 12 - 4/(2^3)`; a few extra sets of parentheses never hurt anything, although when you get confused it’s better to think through the order of operations rather than flailing around adding parentheses at random.

<code>abs</code>	absolute value
<code>cos</code> , <code>sin</code> , <code>tan</code>	cosine, sine, tangent of angle x in radians
<code>exp</code>	exponential function, e^x
<code>log</code>	natural (base- e) logarithm
<code>log10</code>	common (base-10) logarithm
<code>sqrt</code>	square root

Table 1: Some of the built-in mathematical functions in R. You can get a more complete list from the Help system: `?Arithmetic` for simple, `?log` for logarithmic, `?sin` for trigonometric, and `?Special` for special functions.

R also has many *built-in mathematical functions* that operate on variables (Table 1 shows a few).

Exercise 2.1: Using editing shortcuts wherever you can, have R compute the values of

1. $\frac{2^7}{2^7-1}$ and compare it with $(1 - \frac{1}{2^7})^{-1}$ (If any square brackets `[]` show up in your web browser's rendition of these equations, replace them with regular parentheses `()`.)
2.
 - $1 + 0.2$
 - $1 + 0.2 + 0.2^2/2$
 - $1 + 0.2 + 0.2^2/2 + 0.2^3/6$
 - $e^{0.2}$ (remember that R knows `exp` but not e ; how would you get R to tell you the value of e ? What is the point of this exercise?)
3. the standard normal probability density, $\frac{1}{\sqrt{2\pi}}e^{-x^2/2}$, for values of $x = 1$ and $x = 2$ (R knows π as `pi`.) (You can check your answers against the built-in function for the normal distribution; `dnorm(1)` and `dnorm(2)` should give you the values for the standard normal for $x = 1$ and $x = 2$.)

3 The help system

R has a help system, although it is generally better for providing detail or reminding you how to do things than for basic “how do I ...?” questions.

- You can get help on any R function by entering

```
?foo
```

(where `foo` is the name of the function you are interested in) in the console window (e.g., try `?sin`).

- The **Help** menu on the tool bar provides links to other documentation, including the manuals and FAQs, and a Search facility (‘Apropos’ on the menu) which is useful if you sort of maybe remember part of the the name of what it is you need help on.
- Typing `help.start()` opens a web browser with help information.
- `example(cmd)` will run any examples that are included in the help page for command `cmd`.
- `demo(topic)` runs demonstration code on topic `topic`: type `demo()` by itself to list all available demos

By default, R’s help system only provides information about functions that are in the base system and packages that you have loaded with `library` (see below).

- `??topic` or `help.search("topic")` (with quotes) will list information related to `topic` available in the base system or in any extra installed packages: then use `?topic` to see the information, perhaps using `library(pkg)` to load the appropriate package first. `help.search` uses “fuzzy matching” — for example, `help.search("log")` finds 528 entries (on my particular system) including lots of functions with “plot”, which includes the letters “lot”, which are *almost* like “log”. If you can’t stand it, you can turn this behavior off by specifying the incantation `help.search("log",agrep=FALSE)` (81 results which still include matches for “logistic”, “myelogenous”, and “phylogeny” ...)
- `help(package="pkg")` will list all the help pages for a loaded package.
- `example(fun)` will run the examples (if any) given in the help for a particular function `fun`: e.g. `example(log)`
- `RSiteSearch("topic")` does a full-text search of all the R documentation and the mailing list archives for information on `topic` (you need an active internet connection).
- the `sos` package is a web-aware help function that searches all of the packages on CRAN; its `findFn` function tries to find and organize functions in any package on CRAN that match a search string (again, you need a network connection for this).

Try out one or more of these aspects of the help system.

Exercise 3.1 :

Compare the results of `apropos()`, `help()`, `help.search()`, and (after installing the `sos` package and loading it via `library(sos)`) `findFn()` on the search strings "glm" and "generalized linear model". Also try `findFn("{generalized linear model}");` what's the difference?

4 A first interactive session: AIDS data

To get a feel for working in R we'll take a quick look at the AIDS data set I looked at in class.

`read.csv` and `read.table` are the primary tools for getting data into R, for comma-separated and white-space-separated files respectively. `read.csv` assumes the file contains a header specifying the column names, `read.table` doesn't (but you can use `header=TRUE` or `header=FALSE` to override these defaults; use `skip` to skip lines at the beginning of the file).

```
aids <- read.csv("aids_cases.csv")
```

Three different ways of checking the input:

```
head(aids)
```

```
##   year quarter cases
## 1 1984         1     1
## 2 1984         2     6
## 3 1984         3    16
## 4 1984         4    23
## 5 1985         1    27
## 6 1985         2    39
```

```
summary(aids)
```

```
##           year           quarter           cases
##  Min.    :1984   Min.    :1.00   Min.    :  1.0
## 1st Qu.:1985   1st Qu.:1.75   1st Qu.: 29.2
## Median :1986   Median :2.50   Median : 57.0
## Mean   :1986   Mean   :2.50   Mean   : 65.5
## 3rd Qu.:1987   3rd Qu.:3.25   3rd Qu.: 98.8
## Max.    :1988   Max.    :4.00   Max.    :159.0
```

```
str(aids)

## 'data.frame': 20 obs. of 3 variables:
## $ year : int 1984 1984 1984 1984 1985 1985 1985 1985 1986 1986 ...
## $ quarter: int 1 2 3 4 1 2 3 4 1 2 ...
## $ cases : int 1 6 16 23 27 39 31 30 43 51 ...
```

Exercise 4.1: read the data into a temporary variable using `header=FALSE` to tell R (wrongly) that there is *not* a header line in the file. Examine the results of `head`, `summary`, `str`: what happened?

The simplest way to add the `date` column I used in class is:

```
aids$date <- aids$year+(aids$quarter-1)/4
```

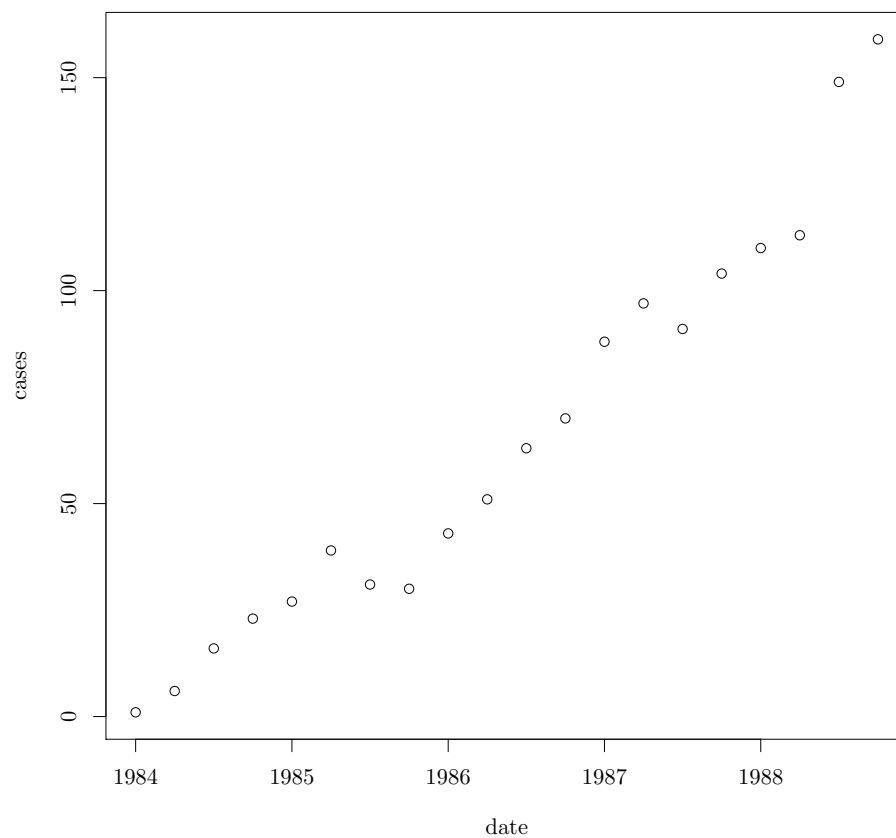
but I find it a bit cleaner to use

```
aids <- transform(aids,
                  date=year+(quarter-1)/4)
```

You can use whatever makes more sense to you.

Basic plot:

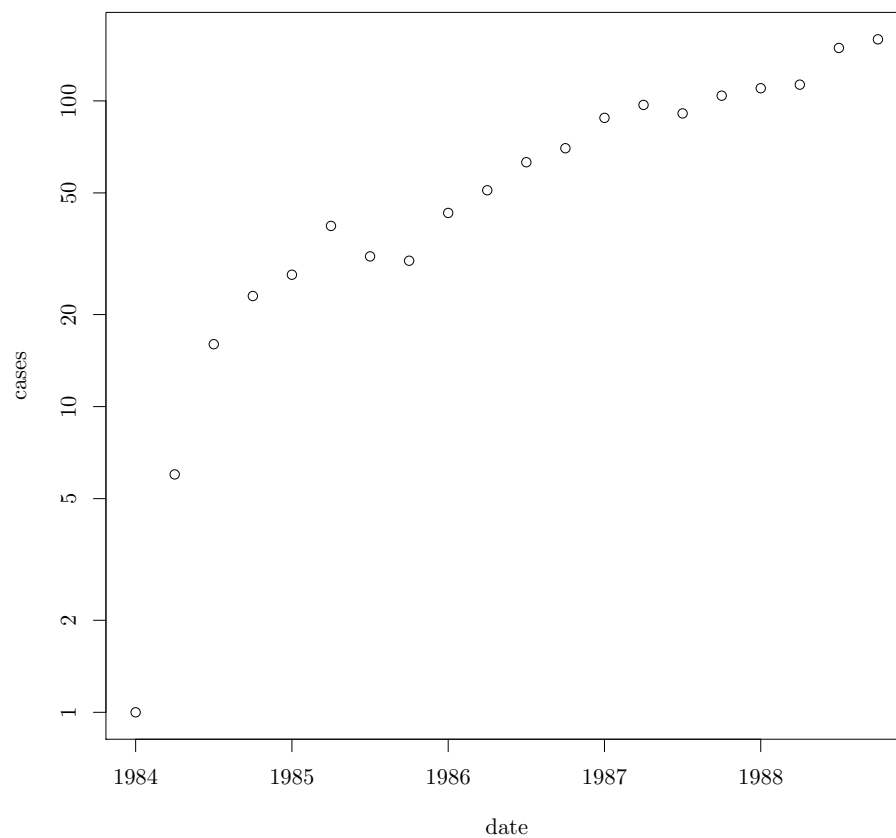
```
with(aids,plot(date,cases))
```



(again: we could say `plot(aids$date,aids$cases)` but I find the `with` syntax cleaner).

Try a logarithmic y axis:

```
with(aids,plot(date,cases,log="y"))
```



(stop and think about what this graph tells you about the applicability of fitting an exponential growth curve to the data!)

Fit a linear regression:

```
lm1 <- lm(cases~date,data=aids)
```

First, replot on the linear (not log) scale. Then add the regression line:

```
abline(lm1)
```

Printing out the results of the fit gives you very basic information about the fit:

```
lm1

##
## Call:
## lm(formula = cases ~ date, data = aids)
##
## Coefficients:
## (Intercept)      date
##   -60744.4      30.6
```

Use `summary` for a more detailed report:

```
summary(lm1)

##
## Call:
## lm(formula = cases ~ date, data = aids)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -16.417  -8.070  -0.164   7.360  20.743
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) -60744.41    3219.30  -18.9  2.6e-13 ***
## date         30.61       1.62    18.9  2.6e-13 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 10.4 on 18 degrees of freedom
## Multiple R-squared:  0.952, Adjusted R-squared:  0.949
## F-statistic: 357 on 1 and 18 DF, p-value: 2.58e-13
```

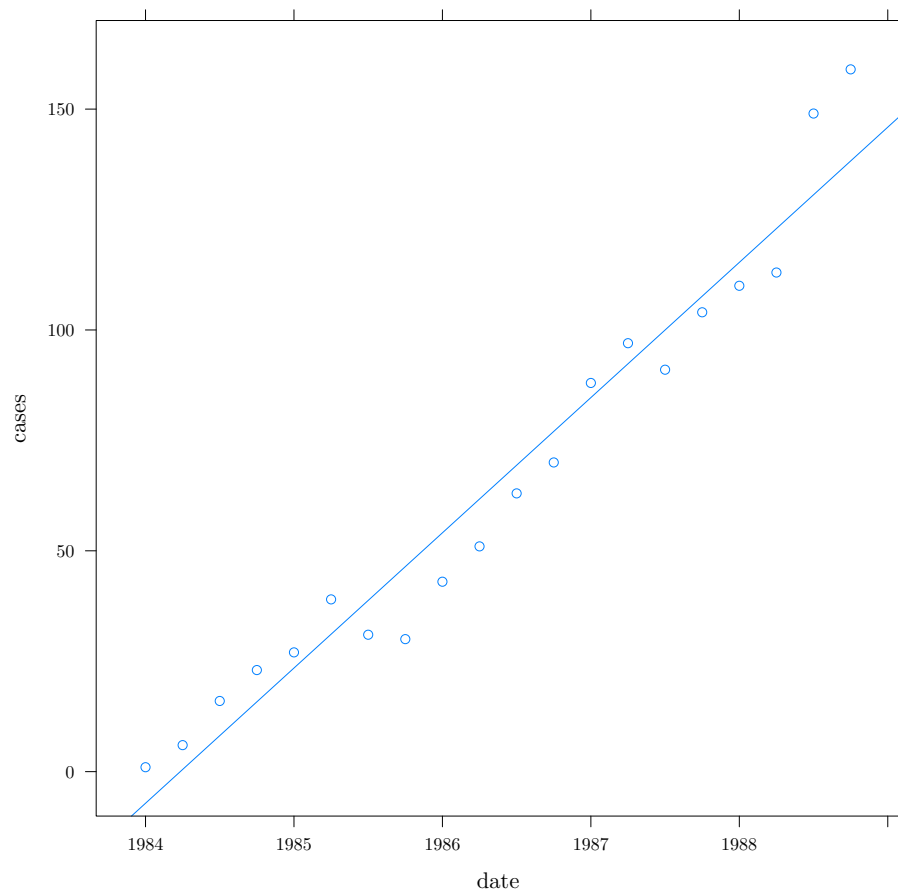
Table 2 gives a list of the standard methods (also called “accessors”) for dealing with fitted statistical models in R.

You can add the regression line slightly more easily with `xyplot` from the `lattice` package: using `type=c("p","r")` plots both points and a regression line.

Method	Function
<code>summary</code>	Detailed information on fit, with coefficient table, t or Z scores, Wald p values, etc.
<code>confint</code> (or intervals)	Confidence intervals on parameters
<code>plot</code>	Diagnostic plots
<code>coef</code>	Coefficient values
<code>anova(x)</code>	ANOVA table or sequential analysis of model x
<code>anova(x,y,...)</code>	Sequential model comparison
<code>simulate</code>	Simulate new data from a fitted model
<code>predict</code>	Predicted values (possibly for new values of the predictors, possibly standard errors and CIs of predictions)
<code>residuals</code>	Extract residuals
<code>vcov</code>	Extract estimated variance-covariance matrix of coefficients
<code>logLik</code>	log-likelihood
<code>AIC</code>	Akaike Information Criterion
<code>drop1</code>	Model-testing via dropping single terms from model
<code>profile</code>	Generate likelihood profiles for predictors

Table 2: Standard accessor methods for fitted models in R.

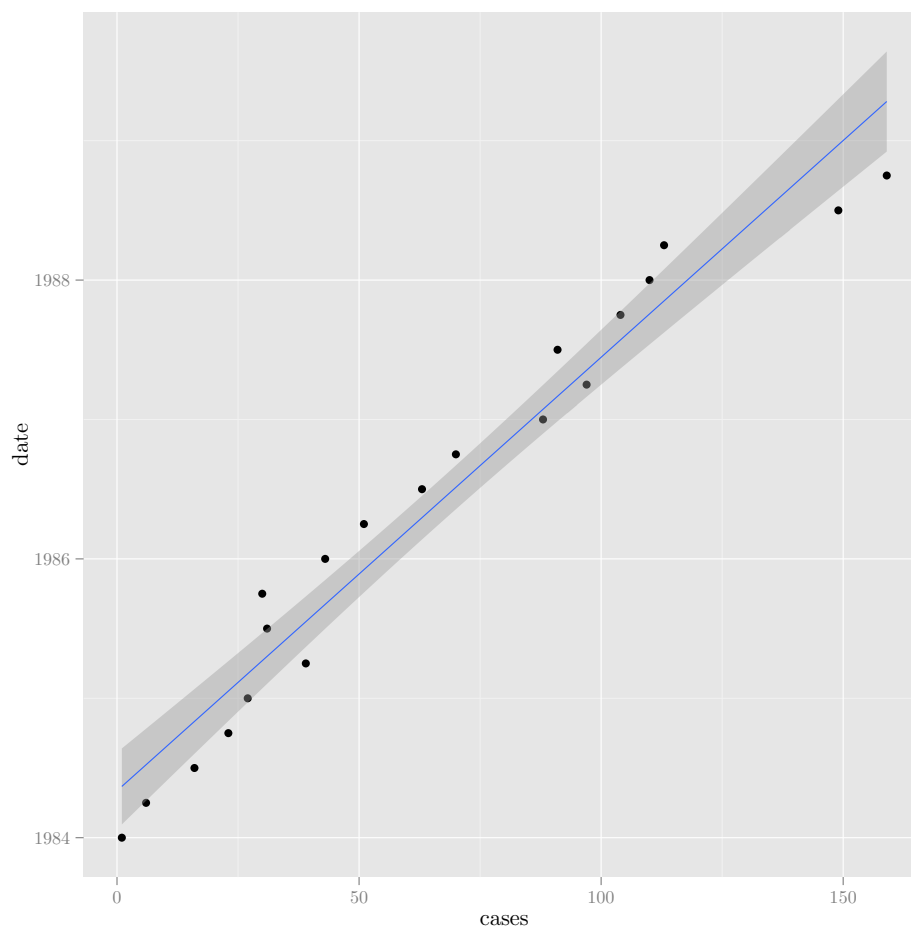
```
library(lattice)
xyplot(cases~date,data=aids,type=c("p","r"))
```



Try out the effects of adding "l" (lines) or "smooth" (a locally weighted linear regression (loess) fit).

Or, with `ggplot`:

```
library(ggplot2)
qplot(cases,date,data=aids)+geom_smooth(method="lm")
```



which adds confidence intervals for the regression as well as the predicted line.

5 R interfaces

5.1 Editors and GUIs

While previous versions of this document recommended a variety of different alternative R editors, as of now (2013) **RStudio** is clearly the best alternative: it works on almost all computers, is well designed, has lots of great features, and is rapidly improving. Unless you already use a powerful editor/development environment such as emacs, vim, Eclipse, or Notepad++ (which all have the capability to handle R scripts intelligently), you should

just use RStudio. If you're curious you can find other editors and graphical front-ends for R at <http://www.r-project.org/GUI/>.

5.2 Script files and data files

Modeling and complicated data analysis are **much** easier if you use *scripts*, which are a series of commands stored in a text file. RStudio makes it easy for you to edit scripts and run the commands within an R session. You **should never, ever** use Microsoft Word to edit R code.

Most programs for working with models or analyzing data follow a simple pattern of program parts:

1. “Setup” statements.
2. (Possibly) input some data from a file or the keyboard.
3. Carry out the calculations that you want.
4. Print the results, graph them, or save them to a file.

For example, a script file might

1. Load some packages, or run another script file that creates some functions (more on functions later).
2. Read in data from a text file.
3. Fit several statistical models to the data and compare them.
4. Graph the results, and save the graph to disk for including in your term project.

Even for relatively simple tasks, script files are useful for building up a calculation step-by-step, making sure that each part works before adding on to it.

Tips for working with data and script files (sounding slightly scary but just trying to help you avoid common pitfalls):

- To tell R where data and script files are located, you can do any one of the following:
 - change your working directory to wherever the file(s) are located using **Session/Set Working Directory** in RStudio (or **Ctrl-Shift-K**).

- change your working directory to wherever the file(s) are located from the R session using the `setwd` (**set** **working** **directory**) function, e.g. `setwd("c:/temp")`
- spell out the *path*, or file location, explicitly. (Use a single forward slash to separate folders (e.g. "c:/My Documents/R/script.R"): this works on all platforms.)
- use `filename=file.choose()` (this works on all platforms, but is only useful on Windows and MacOS).

Changing your working directory is more efficient in the long run, if you save all the script and data files for a particular project in the same directory and switch to that directory when you start work.

You can set your working directory permanently in RStudio via the Tools/Global Option menu item.

- it's vital that you save your data and script files as *plain text* (or sometimes comma-separated) files. There are three things that can go wrong here: (1) if you use a web browser to download files, be careful that it doesn't automatically append some weird suffix to the files; (2) if your web browser has a "file association" (e.g. it thinks that all files ending in `.dat` are Excel files), make sure to save the file as plain text, and without any extra extensions; (3) **never, (never, never) use Microsoft Word to edit your data and script files**; MS Word will try very hard to get you to save them as Word (rather than text) files, which will screw them up!
- If you send script files by e-mail, even if you paste them into the message as plain text, lines will occasionally get broken in different places — leading to confusion. Beware.

Exercise 5.1 *: Download the `aids_script.R` script file. Modify it to fit (and plot the fit of) a negative binomial model instead of a quasi-Poisson model; `glm.nb` is the function in R for fitting a negative binomial (extended) GLM (in the MASS package: you will need to use `library(MASS)` before you can use it). You don't need to specify `family` within `glm.nb`, because it is restricted to negative binomial fits.

6 The R package system

R has many extra packages that provide extra functions. You may be able to install new packages from a menu within R. You can always type

```
install.packages("plotrix")
```

(for example — this installs the `plotrix` package). You can install more than one package at a time:

```
install.packages(c("ellipse", "plotrix"))
```

(`c` stands for “combine”, and is the command for combining multiple things into a single object.) If the machine on which you use R is not connected to the Internet, you can download the packages to some other medium (such as a flash drive or CD) and install them later, using **Install from local zip file** in the menu or

```
install.packages("plotrix", repos=NULL)
```

If you do not have permission to install packages in R’s central directory, R will may ask whether you want to install the packages in a user-specific directory. Go ahead and say yes.

If you are working on a public computer, or have installed R somewhere on your computer that requires administrative privileges to access, you may get a warning message something like **Warning message: In file.create(f.tg) : cannot create file ‘.../packages.html’, reason ‘Permission denied’**. Don’t worry about this; it means the package has been installed successfully, but the main help system index files couldn’t be updated because of file permissions problems.

7 Statistics in R

Some of the important functions and packages (collections of functions) for statistical modeling and data analysis are summarized in Table 3. Venables and Ripley (2002) give a good practical (although somewhat advanced) overview, and you can find a list of available packages and their contents at CRAN, the main R website (<http://www.cran.r-project.org> — select a mirror site near you and click on **Package sources**).

8 Vectors

Vectors and matrices (1- and 2-dimensional rectangular arrays of numbers) are pre-defined data types in R. Operations with vectors and matrices may

<code>aov, anova</code>	Analysis of variance or deviance
<code>lm</code>	Linear models (regression, ANOVA, ANCOVA)
<code>glm</code>	Generalized linear models (e.g. logistic, Poisson regression)
<code>gam</code>	Generalized additive models (in package <code>mgcv</code>)
<code>nls</code>	Fit nonlinear models by least-squares
<code>lme, nlme, lmer, glmer</code>	Linear, generalized linear, and nonlinear mixed-effects models (repeated measures, block effects, spatial models): in packages <code>nlme</code> and <code>lme4</code>
<code>boot</code>	Package: bootstrapping functions
<code>splines</code>	Package: nonparametric regression (more in packages <code>fields</code> , <code>KernSmooth</code> , <code>logspline</code> , <code>sm</code> and others)
<code>princomp, manova, lda, cancor</code>	Multivariate analysis (some in package <code>MASS</code> ; also see packages <code>vegan</code> , <code>ade4</code>)
<code>survival</code>	Package: survival analysis
<code>tree, rpart</code>	Packages: tree-based regression

Table 3: A few of the functions and packages in R for statistical modeling and data analysis. There are *many* more, but you will have to learn about them somewhere else.

seem a bit abstract now, but we need them to do useful things later. Vectors' only properties are their type (or *class*) and length, although they can also have an associated list of names.

We've already seen two ways to create vectors in R:

1. A command in the console window or a script file listing the values, such as

```
initialsize=c(1,3,5,7,9,11)
```

2. Using `read.table`:

```
initialsize=read.table("c:/temp/initialdata.txt")
```

(assuming of course that the file exists in the right place).

You can then use a vector in calculations as if it were a number (more or less)

```
(finalsize=initialsize+1)

## [1]  2  4  6  8 10 12

(newsize=sqrt(initialsize))

## [1] 1.000 1.732 2.236 2.646 3.000 3.317
```

(The parentheses are an R trick that tell it to print out the results of the calculation even though we assigned them to a variable.)

Notice that R applied each operation to every entry in the vector. Similarly, commands like `initialsize-5`, `2*initialsize`, `initialsize/10` apply subtraction, multiplication, and division to each element of the vector. The same is true for

```
initialsize^2

## [1]  1  9 25 49 81 121
```

In R the default is to apply functions and operations to vectors in an *element by element* (or “vectorized”) manner.

8.1 Functions for creating vectors

You can use the `seq` function to create a set of regularly spaced values. `seq`’s syntax is

```
x <- seq(from,to,by)
```

or

```
x <- seq(from,to)
```

or

```
x <- seq(from,to,length.out) ## can abbreviate length.out to 'length'
```

The first form generates a vector (`from`, `from+by`, `from+2*by`, ...) with the last entry not extending further than `to`. In the second form the value of `by` is assumed to be 1 or -1, depending on whether `from`

or `to` is larger. The third form creates a vector with the desired endpoints and length. As we saw above, the syntax `from:to` is a shortcut for `seq(from,to)`:

```
1:8
## [1] 1 2 3 4 5 6 7 8
```

Exercise 8.1 * Use `seq` to create the vector `v=(1 5 9 13)`, and to create a vector going from 1 to 5 in increments of 0.2.

You can use `rep` to create a constant vector such as `(1 1 1 1)`; the basic syntax is `rep(values,lengths)`. For example,

```
rep(3,5)
## [1] 3 3 3 3 3
```

creates a vector in which the value 3 is repeated 5 times. `rep` will repeat a whole vector multiple times

```
rep(1:3,3)
## [1] 1 2 3 1 2 3 1 2 3
```

or will repeat each of the elements in a vector a given number of times:

```
rep(1:3,each=3)
## [1] 1 1 1 2 2 2 3 3 3
```

Even more flexibly, you can repeat each element in the vector a different number of times:

```
rep( c(3,4), c(2,5) )
## [1] 3 3 4 4 4 4 4
```

The value 3 was repeated 2 times, followed by the value 4 repeated 5 times. `rep` can be a little bit mind-blowing as you get started, but it will turn out to be useful.

Table 4 lists some of the main functions for creating and working with vectors.

<code>seq(from,to,by=1)</code>	Vector of evenly spaced values, default increment = 1)
<code>seq(from, to, length.out)</code>	Vector of evenly spaced values, specified length)
<code>c(u,v,...)</code>	Combine a set of numbers and/or vectors into a single vector
<code>rep(a,b)</code>	Create vector by repeating elements of a by amounts in b
<code>as.vector(x)</code>	Convert an object of some other type to a vector
<code>hist(v)</code>	Histogram plot of value in v
<code>mean(v),var(v),sd(v)</code>	Estimate of population mean, variance, standard deviation based on data values in v
<code>cor(v,w)</code>	Correlation between two vectors

Table 4: Some important R functions for creating and working with vectors. Many of these have other optional arguments; use the help system (e.g. `?cor`) for more information. The statistical functions such as `var` regard the values as samples from a population and compute an estimate of the population statistic; for example `sd(1:3)=1`.

8.2 Vector indexing

You will often want to extract a specific entry or other part of a vector. This procedure is called *vector indexing*, and uses square brackets (`[]`):

```
z=c(1,3,5,7,9,11)
z[3]

## [1] 5
```

(how would you use `seq` to construct **z**?) `z[3]` extracts the third item, or *element*, in the vector **z**. You can also access a block of elements using the functions for vector construction, e.g.

```
z[2:5]

## [1] 3 5 7 9
```

extracts the second through fifth elements.

What happens if you enter `v=z[seq(1,5,2)]` ? Try it and see, and make sure you understand what happened.

You can extract irregularly spaced elements of a vector. For example

```
z[c(1,2,5)]
## [1] 1 3 9
```

You can also use indexing to **set specific values within a vector**. For example,

```
z[1]=12
```

changes the value of the first entry in **z** while leaving all the rest alone, and

```
z[c(1,3,5)]=c(22,33,44)
```

changes the first, third, and fifth values (note that we had to use **c** to create the vector — can you interpret the error message you get if you try `z[1,3,5]` ?)

Exercise 8.2 * Write a *one-line* command to extract a vector consisting of the second, first, and third elements of **z** *in that order*.

You may be wondering if vectors in R are row vectors or column vectors (if you don't know what those are, don't worry). The answer is “both and neither”. Vectors are printed out as row vectors, but if you use a vector in an operation that succeeds or fails depending on the vector's orientation, R will assume that you want the operation to succeed and will proceed as if the vector has the necessary orientation. For example, R will let you add a vector of length 5 to a 5×1 matrix or to a 1×5 matrix, in either case yielding a matrix of the same dimensions.

Exercise 8.3 Write a script file that computes values of $y = \frac{(x-1)}{(x+1)}$ for $x = 1, 2, \dots, 10$, and plots y versus x with the points plotted and connected by a line (hint: in `?plot`, search for `type`).

Exercise 8.4 * The sum of the geometric series $1 + r + r^2 + r^3 + \dots + r^n$ approaches the limit $1/(1-r)$ for $r < 1$ as $n \rightarrow \infty$. Set the values $r = 0.5$ and $n = 10$, and then write a **one-line** command that creates the vector $G = (r^0, r^1, r^2, \dots, r^n)$. Compare the sum (using `sum`) of this vector to the limiting value $1/(1-r)$. Repeat for $n = 50$. (*Note* that comparing very similar numeric values can be tricky: rounding can happen, and some numbers cannot be represented exactly in binary (computer) notation. By default R displays 7 significant digits (`options("digits")`). For example:

```
x = 1.999999
x
## [1] 2

x-2
## [1] -1e-06

x=1.999999999999999
x
## [1] 2

x-2
## [1] -9.992e-14
```

All the digits are still there, in the second case, but they are not shown. Also note that `x-2` is not exactly -1×10^{-13} ; this is unavoidable.)

8.3 Logical operators

Logical operators return a value of `TRUE` or `FALSE`. For example, try:

```
a=1
b=3
c=a<b
d=(a>b)
c
## [1] TRUE

d
## [1] FALSE
```

The parentheses around `(a>b)` are optional but make the code easier to read. One special case where you *do* need parentheses (or spaces) is when you make comparisons with negative values; `a<-1` will surprise you by setting `a=1`, because `<-` (representing a left-pointing arrow) is equivalent to `=` in R. Use `a< -1`, or more safely `a<(-1)`, to make this comparison.

<code>x<y</code>	less than
<code>x>y</code>	greater than
<code>x<=y</code>	less than or equal to
<code>x>=y</code>	greater than or equal to
<code>x==y</code>	equal to
<code>x!=y</code>	<i>not</i> equal to

Table 5: Some comparison operators in R. Use `?Comparison` to learn more.

When we compare two vectors or matrices of the same size, or compare a number with a vector or matrix, comparisons are done element-by-element. For example,

```
x=1:5
(b=(x<=3))

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

So if `x` and `y` are vectors, then `(x==y)` will return a vector of values giving the element-by-element comparisons. If you want to know whether `x` and `y` are identical vectors, use `identical(x,y)` which returns a single value: `TRUE` if each entry in `x` equals the corresponding entry in `y`, otherwise `FALSE`. You can use `?Logical` to read more about logical operators. **Note the difference between `=` and `==`: can you figure out what happened in the following cautionary tale?**

```
a = 1:3
b = 2:4
a==b

## [1] FALSE FALSE FALSE

a=b
a==b

## [1] TRUE TRUE TRUE
```

Exclamation points `!` are used in R to mean “not”; `!=` (not `!==`) means “not equal to”.

R also does arithmetic on logical values, treating `TRUE` as 1 and `FALSE` as 0. So `sum(b)` returns the value 3, telling us that three entries of `x`

satisfied the condition (`x<=3`). This is useful for (e.g.) seeing how many of the elements of a vector are larger than a cutoff value.

Build more complicated conditions by using *logical operators* to combine comparisons:

! Negation

&, && AND

|, || OR

OR is *non-exclusive*, meaning that `x|y` is true if either `x` or `y` or *both* are true (a ham-and-cheese sandwich would satisfy the condition “ham OR cheese”). For example, try

```
a=c(1,2,3,4)
b=c(1,1,5,5)
(a<b)& (a>3)

## [1] FALSE FALSE FALSE  TRUE

(a<b) | (a>3)

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

and make sure you understand what happened (if it’s confusing, try breaking up the expression and looking at the results of `a<b` and `a>3` separately first). The two forms of AND and OR differ in how they handle vectors. The shorter one does element-by-element comparisons; the longer one only looks at the first element in each vector.

8.4 Vector indexing II

We can also use *logical* vectors (lists of TRUE and FALSE values) to pick elements out of vectors. This is important, e.g., for subsetting data (getting rid of those pesky outliers!)

Suppose we only want to look at the largest values in the `aids` data set defined earlier:

```
(large.cases <- aids$cases>0.9*max(aids$cases))
aids$date[large.cases]
aids$cases[large.cases]
```

We use `aids$cases>0.9*max(aids$cases)` to generate a logical vector the same length as the data set (FALSE FALSE FALSE ...) which is then used to select the appropriate values.

If you want the *positions* at which the AIDS cases are greater than 90% of the maximum, you could say `(1:length(aids$cases))[aids$cases>0.9*max(aids$cases)]`, but you can also use a built-in function: `which(aids$cases>0.9*max(aids$cases))`. If you wanted the position at which the maximum value occurs, you could say `which(aids$cases==max(aids$cases))`. (This normally results in a vector of length 1; when could it give a longer vector?) There is even a built-in command for this specific function, `which.max` (although `which.max` always returns just the *first* position at which the maximum occurs).

Exercise 8.5: What would happen if instead of setting `large.cases` you replaced `aids$cases` by saying `aids$cases <- aids$cases[aids$cases>0.9*max(aids$cases)]`? What if you then said `and then aids$date <- aids$date[aids$cases>0.9*max(aids$cases)]`? This leads to some very funny results, so don't try it on the `aids` data set (if you do, be prepared to read the data in from the file again).

Try it with some temporary variables — set `junk <- aids` and then play with `junk` so you don't mess up your working variables — and work out what happened ...

We can also combine logical operators (making sure to use the element-by-element `&` and `|` versions of AND and OR):

```
aids$cases[aids$cases>0.9*max(aids$cases) | aids$date >= 1987]
```

When you work with data frames (that's what `aids` is — see below) it may be cleaner to work with the `subset` command, which looks inside the data frame for the variables, so that you don't have to include `aids$` to refer to them, for example:

```
subset(aids,cases>0.9*max(cases) | date >= 1987)
```

This is much easier to read. Remember that just using `subset` doesn't change the original data frame — if you want to work with the subsetted data, you need `aids <- subset(aids,...)` (or, perhaps better, define a new variable `aids_sub <- subset(aids,...)`).

Exercise 8.6 `runif(n)` is a function (more on it soon) that generates a vector of `n` random, uniformly distributed numbers between 0 and 1. Create a vector of 20 numbers, then select the subset of those numbers that is less than the mean. (If you want your answers to match mine exactly, use

`set.seed(273)` to set the random-number generator to a particular starting point before you use `runif`; [273 is an arbitrary number I chose].)

Exercise 8.7* Find the *positions* of the elements that are less than the mean of the vector you just created (e.g. if your vector were (0.1 0.9 0.7 0.3) the answer would be (1 4)).

As I mentioned in passing above, vectors can have names associated with their elements: if they do, you can also extract elements by name (use `names` to find out the names).

```
x = c(first=7,second=5,third=2)
names(x)

## [1] "first" "second" "third"

x["first"]

## first
##      7

x[c("third","first")]

## third first
##      2      7
```

Finally, it is sometimes handy to be able to drop a particular set of elements, rather than taking a particular set: you can do this with negative indices. For example, `x[-1]` extracts all but the first element of a vector.

Exercise 8.8*: Specify two ways to take only the elements in the odd positions (first, third, ...) of a vector of arbitrary length.

9 Matrices

9.1 Creating matrices

Like vectors, you can create matrices by using `read.table` to read in values from a data file. (Actually, this creates a data frame, which is *almost* the same as a matrix — see section 10.3.) You can also create matrices of numbers by creating a vector of the matrix entries, and then reshaping them to the desired number of rows and columns using the function `matrix`. For example

```
(X=matrix(1:6,nrow=2,ncol=3))
```

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

takes the values 1 to 6 and reshapes them into a 2 by 3 matrix.

By default R loads the values into the matrix *column-wise* (this is probably counter-intuitive since we're used to reading tables row-wise). Use the optional parameter `byrow` to change this behavior. For example :

```
(A=matrix(1:9,nrow=3,ncol=3,byrow=TRUE))
```

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

R will re-cycle through entries in the data vector, if necessary to fill a matrix of the specified size. So for example

```
matrix(1,nrow=10,ncol=10)
```

creates a 10×10 matrix of ones.

Exercise 9.1 Use a command of the form `X=matrix(v,nrow=2,ncol=4)` where `v` is a data vector, to create the following matrix `X`:

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

If you can, try to use R commands to construct the vector rather than typing out all of the individual values.

R will also collapse a matrix to behave like a vector whenever it makes sense: for example `sum(X)` above is 12.

Exercise 9.2 Use `rnorm` (which is like `runif`, but generates Gaussian (normally distributed) numbers with a specified mean and standard deviation instead) and `matrix` to create a 5×7 matrix of Gaussian random numbers with mean 1 and standard deviation 2. (Use `set.seed(273)` again for consistency.)

<code>matrix(v,nrow=m,ncol=n)</code>	$m \times n$ matrix using the values in <code>v</code>
<code>t(A)</code>	transpose (exchange rows and columns) of matrix <code>A</code>
<code>dim(X)</code>	dimensions of matrix <code>X</code> . <code>dim(X)[1]</code> =# rows, <code>dim(X)[2]</code> =# columns
<code>data.entry(A)</code>	call up a spreadsheet-like interface to edit the values in <code>A</code>
<code>diag(v,n)</code>	diagonal $n \times n$ matrix with <code>v</code> on diagonal, 0 elsewhere (<code>v</code> is 1 by default, so <code>diag(n)</code> gives an $n \times n$ identity matrix)
<code>cbind(a,b,c,...)</code>	combine compatible objects by attaching them along columns
<code>rbind(a,b,c,...)</code>	combine compatible objects by attaching them along rows
<code>as.matrix(x)</code>	convert an object of some other type to a matrix, if possible
<code>outer(v,w)</code>	“outer product” of vectors <code>v</code> , <code>w</code> : the matrix whose $(i,j)^{\text{th}}$ element is <code>v[i]*w[j]</code> ; <code>v %*% t(v)</code> is equivalent
<code>t(v) %*% v</code>	inner product (or equivalently <code>sum(v^2)</code>)

Table 6: Some important functions for creating and working with matrices. Many of these have additional optional arguments; use the help system for full details.

Another useful function for creating matrices is `diag`. `diag(v,n)` creates an $n \times n$ matrix with data vector `v` on its diagonal. So for example `diag(1,5)` creates the 5×5 *identity matrix*, which has 1’s on the diagonal and 0 everywhere else. Try `diag(1:5,5)` and `diag(1:2,5)`; why does the latter produce a warning?

Finally, you can use the `data.entry` function. This function can only edit existing matrices, but for example (try this now!!)

```
A=matrix(0,nrow=3,ncol=4)
data.entry(A)
```

will create `A` as a 3×4 matrix, and then call up a spreadsheet-like interface in which you can change the values to whatever you need.

9.2 cbind and rbind

If their sizes match, you can combine vectors to form matrices, and stick matrices together with vectors or other matrices. `cbind` (“column bind”) and `rbind` (“row bind”) are the functions to use.

`cbind` binds together columns of two objects. One thing it can do is put vectors together to form a matrix:

```
(C=cbind(1:3,4:6,5:7))
```

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

R interprets vectors as row or column vectors according to what you're doing with them. Here it treats them as column vectors so that columns exist to be bound together. On the other hand,

```
(D=rbind(1:3,4:6))
```

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

treats them as rows. Now we have two matrices that can be combined.

Exercise 9.3 Verify that `rbind(C,D)` works, `cbind(C,C)` works, but `cbind(C,D)` doesn't. Why not?

9.3 Matrix indexing

Matrix indexing is like vector indexing except that you have to specify both the row and column, or range of rows and columns. For example `z=A[2,3]` sets `z` equal to 6, which is the (2nd row, 3rd column) entry of the matrix **A** that you recently created, and

```
A[2,2:3]
```

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

```
(B=A[2:3,1:2])
```

```
##      [,1] [,2]
## [1,]    4    5
## [2,]    7    8
```

There is an easy shortcut to extract entire rows or columns: leave out the limits, leaving a blank before or after the comma.

```
(first.row=A[1,])
## [1] 1 2 3

(second.column=A[,2])
## [1] 2 5 8
```

(What does `A[,]` do?)

As with vectors, indexing also works in reverse for assigning values to matrix entries. For example,

```
(A[1,1]=12)
## [1] 12
```

You can do the same with blocks, rows, or columns, for example

```
(A[1,]=c(2,4,5))
## [1] 2 4 5
```

If you use `which` on a matrix, R will normally treat the matrix as a vector — so for example `which(A==8)` will give the answer 6 (figure out why). However, `which` does have an option that will treat its argument as a matrix:

```
which(A==8,arr.ind=TRUE)
##      row col
## [1,]   3   2
```

10 Other structures: Factors, lists and data frames

10.1 Factors

Factors are R's way of storing categorical variables. When R reads data with `read.table` or `read.csv`, they are the default way of storing non-numeric

data. Factors are stored internally as a list of integer codes, along with a list of “levels” — character labels for each factor level. Typically, a data frame (see below) for a statistical analysis will be composed entirely of numeric vectors and factors — numeric vectors representing continuous predictors and responses, factors representing categorical predictors and responses.

Unless you have very specific needs, it’s best to leave categorical variables coded by their names (“high”, “low”) rather than converting them into dummy variables yourself (0,1) or (1,2). R will do much of this conversion automatically as necessary, and in the meantime you won’t have to remember the meaning of your dummy variables. In addition, R usually arranges plots in order of factor levels, so putting your factor levels in a meaningful order (see below) is good practice.

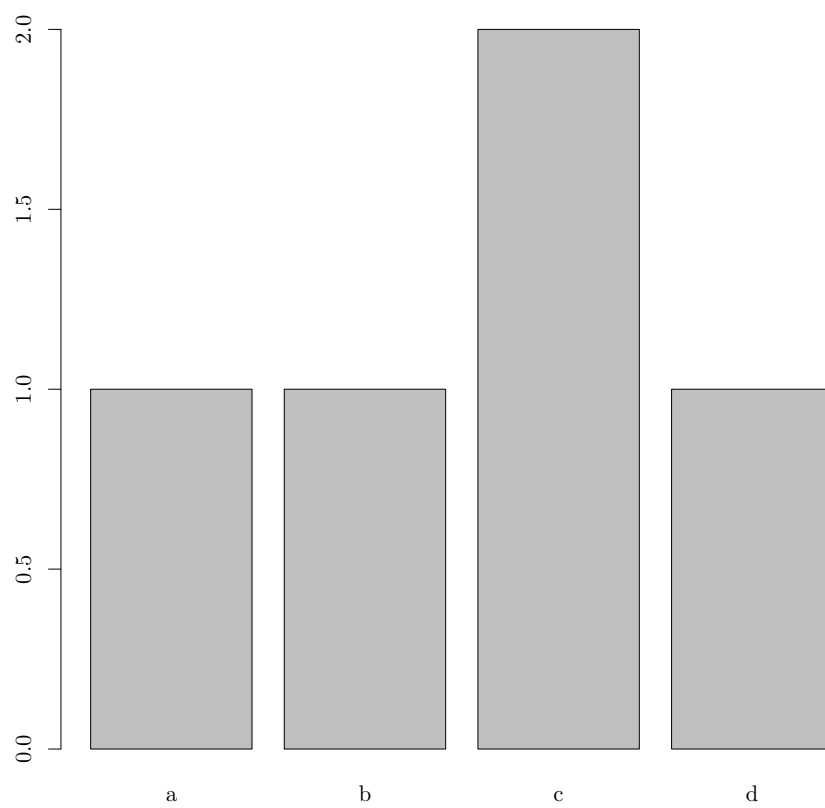
```
xx <- factor(c("a", "b", "c", "c", "d"))
str(xx)

##  Factor w/ 4 levels "a","b","c","d": 1 2 3 3 4

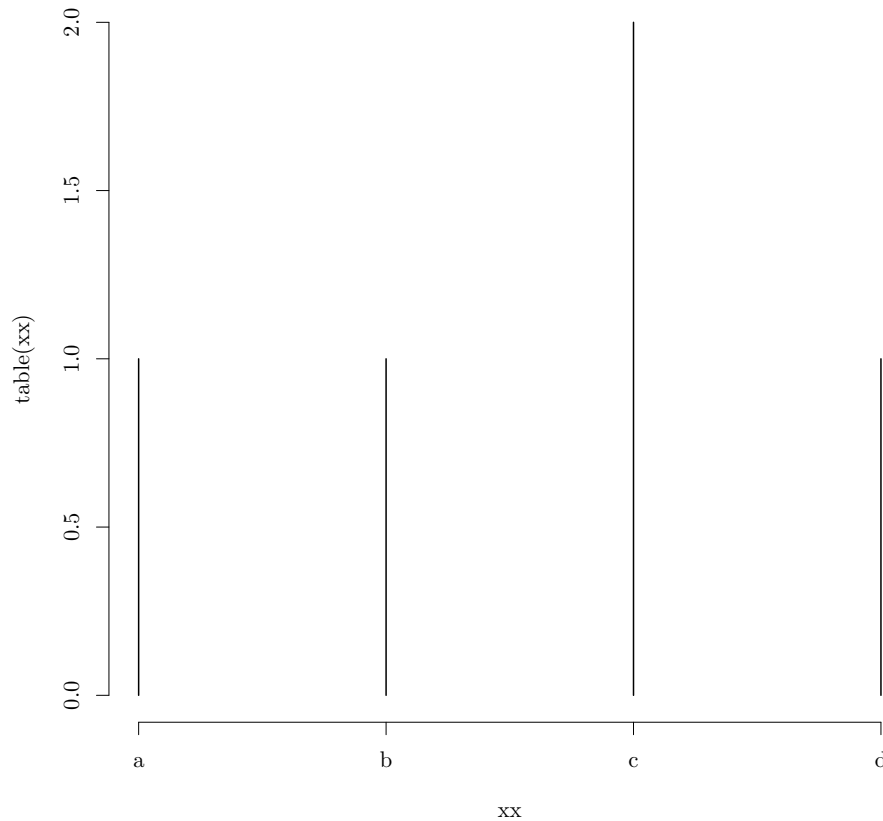
table(xx)

## xx
## a b c d
## 1 1 2 1

plot(xx)
```



```
plot(table(xx))
```



- If a numeric vector gets converted to a factor by accident, use `as.numeric(as.character(x))` to convert it back to numeric — *not* `as.numeric(x)`. The latter will pull out the integer codes, which is not what you want ...
- to order the levels of a factor, use `factor(x, levels=L)`. For example, if you read in a data set with a predictor whose categories are `small`, `medium`, `large`, R will by default order the categories alphabetically.

```
transform(data, sizefac=factor(sizefac, levels=c("small", "medium", "large")))
```

or, if the rows of the data have the factors in the correct order,

```
transform(data, sizefac=factor(sizefac, levels=unique(as.character(sizefac))))
```

- to reorder the levels of a factor according to the values of another variable, use `reorder`.
- when you subset data and eliminate all the values corresponding to some factor, the empty levels remain in the data. Use `droplevels` when you want to get rid of them. For example:

```
table(xx[1:3])

##
## a b c d
## 1 1 1 0

table(droplevels(xx[1:3]))

##
## a b c
## 1 1 1
```

10.2 Lists

While vectors and matrices may seem familiar, lists are probably new to you. Vectors and matrices have to contain elements that are all the same type: lists in R can contain anything — vectors, matrices, other lists ... Indexing lists is a little different too: use double square brackets `[[ ]]` (rather than single square brackets as for a vector) to extract an element of a list by number or name, or `$` to extract an element by name (only). Given a list like this:

```
L = list(A=x, B=y, C=c("a", "b", "c"))
```

Then `L$A`, `L[["A"]]`, and `L[[1]]` will all grab the first element of the list.

You won't use lists too much at the beginning, but many of R's own results are structured in the form of lists.

10.3 Data frames

Data frames are the solution to the problem that most data sets have several different kinds of variables observed for each sample (e.g. categorical site location and continuous rainfall), but matrices can only contain a single type of data. Data frames are a hybrid of lists and vectors; internally, they are a list of vectors that may be of different types but must all be the same length, but you can do most of the same things with them (e.g., extracting a subset of rows) that you can do with matrices. You can index them either the way you would index a list, using `[[ ]]` or `$` — where each variable is a different item in the list — or the way you would index a matrix. Use `as.matrix` if you have a data frame (where all variables are the same type) that you really want to be a matrix, e.g. if you need to transpose it (use `as.data.frame` to go the other way).

Exercise 10.1 * Do problem 1.6 on p. 17 of Dobson & Barnett. The data are in a file called `applemoths.txt`.

You can do part (c) (estimating θ numerically) with the `mle2` function in R: fill in the dots below appropriately.

```
library(bbmle)
mle2(y~dbinom(prob=p,size=total),data=...,start=list(p=...))
```

where `total` is the name of a variable in your data set that contains the total number of moths per group (and the dots need to be filled in appropriately by you).

In addition, plot the frequencies of females in order from smallest to largest frequency. For an extra challenge, figure out how to make the point sizes proportional to the size of the group.

Exercise 10.2 * Using the `dnbinom`, `rgamma`, `rpois` (and `curve` and `hist`) functions, confirm by simulation that compounding a Poisson distribution with a Gamma, i.e.

$$X \sim \text{Poisson}(\lambda)$$

$$\lambda \sim \text{Gamma}(r = k/\mu, a = k)$$

results in a negative binomial distribution with mean μ and shape parameter k . Draw a picture with $\mu = 2$, $k = 2$.

References

- Ihaka, R. and R. Gentleman (1996). R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics* 5(3), 299–314.
- Venables and Ripley (2002). *Modern Applied Statistics with S* (4th ed.). New York: Springer.