An introduction to R for modeling (lab 1)

Stephen Ellner[*], modified by Ben Bolker and Steve Walker [†]

September 4, 2014

Version: 2014-09-04 10:13:09

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 will 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,

[*]Ecology and Evolutionary Biology, Cornell University

[†]Department of Mathematics & Statistics and Biology, McMaster University

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 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 [1], 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 also includes extensive documentation, including an introductory manual ($\approx$ 100 pages) and a comprehensive reference manual (over 1000 pages).

1.1 Installing R and RStudio on your computer: basics

If R and RStudio are already installed on your computer, you can skip this section.

Here are the instructions:

- go to `http://www.rstudio.com/ide/download/desktop`
- if you don't yet have R, follow the link at the end of this sentence: *RStudio requires R 2.11.1 (or higher). If you don't already have R, you can download it here.*

  - click on the download corresponding to your operating system
  - click on R-3.1.1-snowleopard.pkg or R-3.1.1-mavericks.pkg if you are on a Mac and the link that says *install R for the first time* if you are on Windows.
  - read and follow the instructions (which are pretty much "click on the icon").

- go back to `http://www.rstudio.com/ide/download/desktop`
- click on the download most appropriate for your system
- read and follow instructions

R should work well on any reasonably modern computer. 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.)

2 Interactive calculations

When you start RStudio 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
```

(When cutting and pasting from this document to R, don't include the text for the command prompt (`>`).)

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 +
+  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 (or `Control-P`) 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

`abs`	absolute value
`cos`, `sin`, `tan`	cosine, sine, tangent of angle x in radians
`exp`	exponential function, e^x
`log`	natural (base-e) logarithm
`log10`	common (base-10) logarithm
`sqrt`	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.

```
> 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**xcuse **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.

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 : Do an Apropos on `sin` via the Help menu, to see what it does. Now enter the command

```
> help.search("sin")
```

and see what that does (answer: `help.search` pulls up all help pages that include 'sin' anywhere in their title or text. Apropos just looks at the name of the function).

4 A first interactive session: a "leaky bucket" model

To get a feel for working in R we'll construct a simple discrete-time dynamical system, sometimes known as the "leaky bucket" model.

Suppose that in a queue (a group of people or things waiting for service — e.g. people waiting at a bank, or jobs waiting for processing on a computer system, or cars in line at a toll booth), 25% (rounded) of the people waiting are served every hour and 10 new people arrive every hour. (For precision, we will assume that the 10 new people arrive at the *end* of the hour and are not counted in the fraction served.)

Suppose the queue is initially empty. We could run this model by brute force by typing

```
> N <- 0
```

to set the initial state and then typing (or cutting and pasting)

```
> N <- N - round(0.25 * N)
> N <- N + 10
```

over and over again.

Of course, this would be extremely tedious (we also wouldn't see the results unless we typed `N` by itself from time to time to see where we'd gotten). This approach doesn't save the results over time; we can only see the current state of the system.

Suppose that we decide we want to run the system for 20 steps. We can set aside space for a vector of 20 numbers, set the first to zero, and assign a variable `i` as a counter (if `i` is an integer in the correct range, then `N[i]` refers to the `i`th value from the vector `N` — more on this below):

```
> N <- numeric(20)
> N[1] <- 0
> i <- 1
```

Then we can do the following steps repeatedly:

```
> i <- i+1
> N[i] <- N[i-1]-round(0.25*N[i-1])+10  ## condense
```

Of course, this doesn't help much in the tedium department. We should instead use a `while` statement:

```
> i <- 1
> while (i<20) {
+    i <- i+1
+    N[i] <- N[i-1]-round(0.25*N[i-1])+10
+  }
> N

##  [1]  0 10 18 24 28 31 33 35 36 37 38 38 38 38 38 38 38 38
## [19] 38 38
```

R repeatedly tests the *condition* `i<20` given in parentheses after `while`; if it is true, it runs all of the code inside the curly brackets (`{}`), then starts again.

Exercise 4.1: Why did we use the condition `i < 20` rather than the condition `i <= 20`?

Alternatively, since we know in advance how many times we want to run the model, we can instead use a "`for` loop":

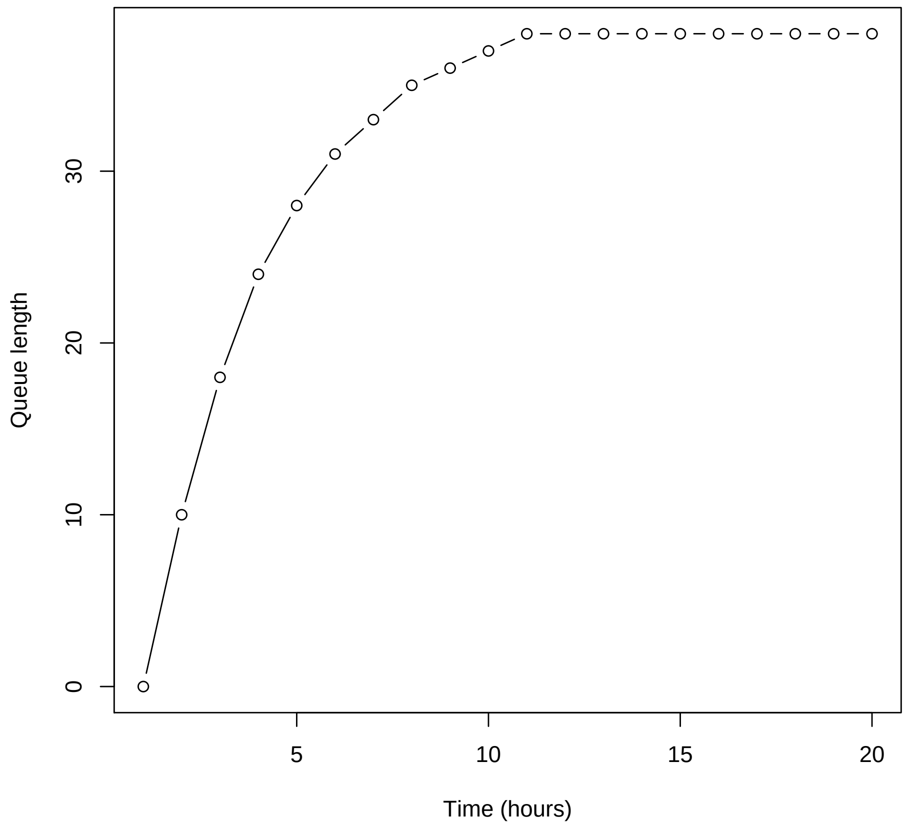
```
> for (i in 2:20) {
+    N[i] <- N[i - 1] - round(0.25 * N[i - 1]) + 10
+  }
> N

##  [1]  0 10 18 24 28 31 33 35 36 37 38 38 38 38 38 38 38 38
## [19] 38 38
```

The colon (:) operator creates a sequence of integers starting at 2 and ending at 20 (notice that we start from 1, not 2, because we have already set the value for `N[1]`). For each value in this vector, R sets the loop variable `i` to that value and runs the code inside the curly brackets.

Let's plot the results. `plot(N)` by itself will plot the values in `N` on the y-axis against their "index" (position in the vector) on the x-axis, but it might be better to specify the x values explicitly, as in `plot(1:20,N)`. Perhaps even better, we can assign the x values to a variable: at the same time, we can (1) specify that R should use both lines and points to plot the values by specifying `type="b"` and (2) specify more informative labels for the axes (see Figure 1).

```
> tvec <- 1:20
> plot(tvec, N, type = "b",
+       xlab = "Time (hours)",
+       ylab = "Queue length")
```



R's default plotting character is an open circle. Open symbols are good for plotting large data sets because it is easier to see where they overlap, but you could include `pch=16` in the `plot` command if you wanted closed circles instead. Figure 2 shows several more ways to adjust the appearance of lines and points in R.

5 R-markdown

R-markdown is a useful tool for reporting your findings. Essentially, R-markdown allows you to combine descriptions in English (or any other natural language) with your R scripts, and R output in a single document. Fortunately, RStudio makes R-markdown very easy! Currently, `html`, `pdf`, and `docx` formats are supported by RStudio (`html` is least likely to cause

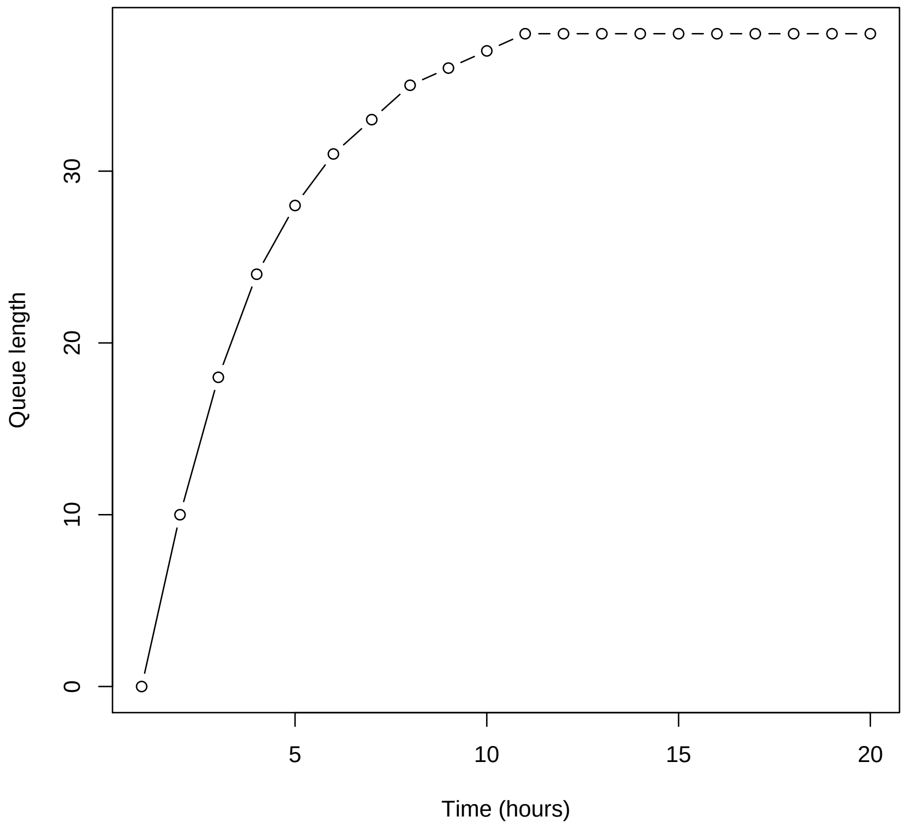


Figure 1: Queue length “simulation”: $N(t) = N(t-1) - 0.25 \cdot \mathrm{round}(N(t-1) + 10$.

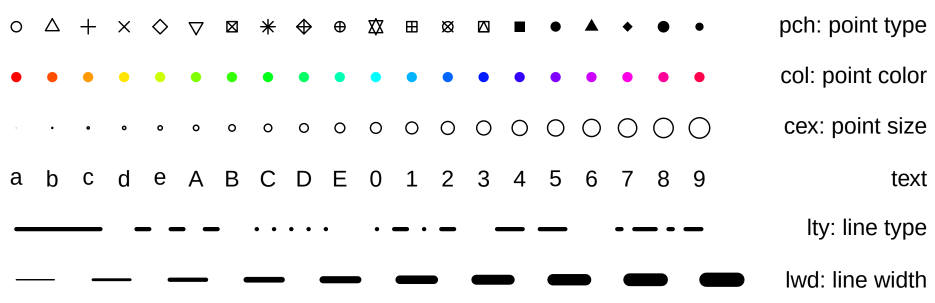


Figure 2: Some of R's graphics parameters. Color specification, `col`, also applies in many other contexts: colors are set to a rainbow scale here. See `?par` for (many more) details on graphics parameters, and one or more of `?rgb`, `?palette`, or `apropos("color")` for more on colors.

problems, so please use this unless otherwise instructed). **You will be required to submit all of your assignments using R-markdown, so please take a few moments to get it working on your system now.**

To begin, select `File: New: R Markdown...` and follow the instructions. Choose an `html` document for now. At this point, RStudio will probably recommend that you install several add-on packages – you should follow this advice. If everything goes well, a sample document will open up. To test that it works, click on the Knit HTML icon and an `html` document should appear. That's it for now. **Please email either David or Steve if something goes wrong with setting up R-markdown.**

The template R-markdown file provided by RStudio looks something like this,

```
---
title: "testing"
output: html_document
---

This is an R Markdown document. Markdown is a simple formatting syntax
for authoring HTML, PDF, and MS Word documents. For more details on
using R Markdown see <http://rmarkdown.rstudio.com>.

When you click the **Knit** button a document will be generated that
includes both content as well as the output of any embedded R code
chunks within the document. You can embed an R code chunk like this:
```

~~~
```{r}
summary(cars)
```

You can also embed plots, for example:

```{r, echo=FALSE}
plot(cars)
```

Note that the `echo = FALSE` parameter was added to the code chunk to
prevent printing of the R code that generated the plot.
~~~

Notice how this document combines English sentences (e.g. "This is an...") with R code (e.g. `summary(cars)`). When you fit `knit HTML`, the R code is evaluated and displayed with the documentation provided. To learn about the R-markdown format, you can find help by clicking the ? icon. You will be presented with two options, `Using R Markdown` and `Markdown Quick Reference` – both are useful. The former takes you to a website with a tutorial, and the latter to a pane within RStudio giving tips on syntax. An example R-markdown file can be found at FIXME, which illustrates how to write up a homework assignment. Note that this example also illustrates how to use the LaTeX language within R-Markdown in order to produce beautiful mathematics. If you have never heard of LaTeX, it is a system for typesetting mathematics (see FIXME for more details).

6 Script files and data files

When you are just trying to figure something out, rather than reporting on your findings, R scripts can be very useful. Scripts are series' of commands stored in text files. Open a new script file in RStudio using the `File: New: R Script` menu option.

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

1. Initialization 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:
  - use the RStudio magic of pointing and clicking!
  - 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).
  - change your working directory to wherever the file(s) are located using `Change dir` in the `File` menu (Windows: on Mac it's `Misc/Change Working Directory`);
  - change your working directory to wherever the file(s) are located using the `setwd` (**set** **w**orking **d**irectory) function, e.g. `setwd("c:/temp")`

  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.

- 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.

As a first example, the file `bucketModel.R` has the commands from the leaky bucket model, which can be found at FIXME. **Important:** before working with an example file, create a personal copy in some location on your own computer. We will refer to this location as your *temp folder*. The `File` tab in RStudio will help you keep track of the directory structure of your projects. At the end of a lab session you *must* move files onto your personal disk (or email them to yourself).

Now open **your copy** of `bucketModel.R`. In RStudio, select the entire text of the file and click on the Run icon (pro-tip: hovering over this icon will reveal a hot key). This has the same effect as entering the commands by hand into the console: they will be executed and so a graph is displayed with the results. Highlighting specific pieces of the file allows you to execute script files one piece at a time (which is useful for finding and fixing errors). The `source` function allows you to run an entire script file, e.g.

```
> source("c:/temp/bucketModel.R")
```

Exercise 6.1 Make a copy of `bucketModel.R` under a new name, and modify the copy so that it picks the number of new arrivals from a Poisson distribution with mean 10 (`rpois(1, lambda = 10)`) and plots the data appropriately. (We will learn about the Poisson distribution later.) In the "setup" phase of your code, use the command `set.seed(101)` to initialize the random number generator so that you get the same answer as I did. You should end up with a graph that resembles Figure 3.

Exercise 6.2 The axes in plots are scaled automatically, but the outcome is not always ideal (e.g. if you want several graphs with exactly the same axis limits). You can use the `xlim` and `ylim` arguments in `plot` to control the limits:

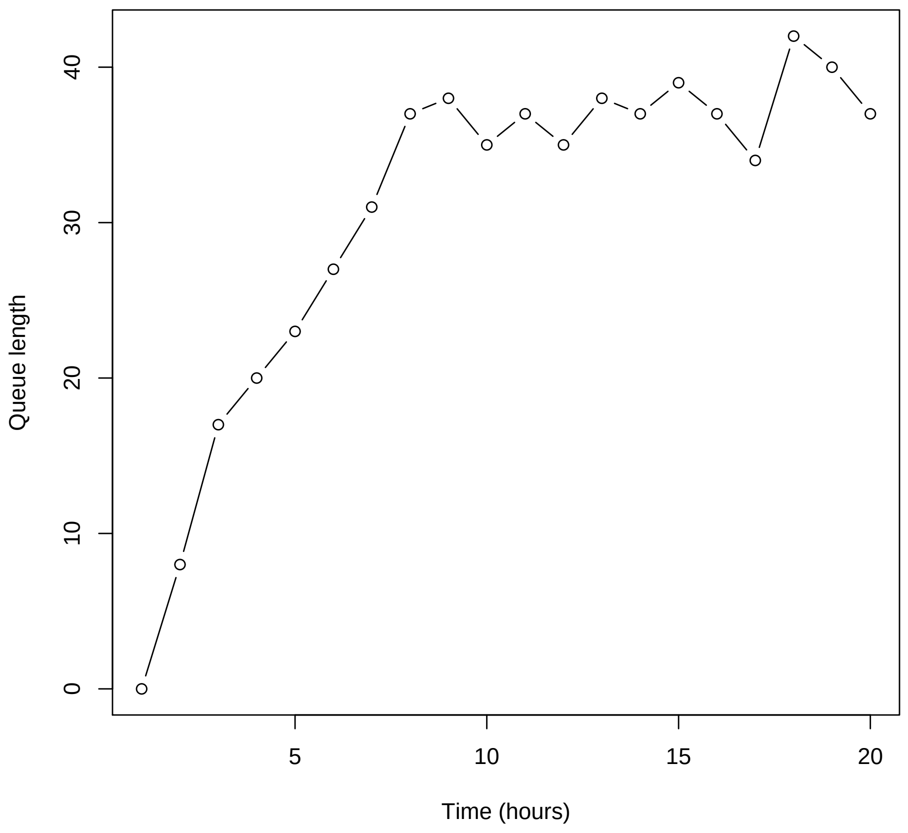


Figure 3: Queue simulation with random arrivals.

```
> plot(x, y, xlim = c(x1, x2), [other stuff])
```

will draw the graph with the x-axis running from `x1` to `x2`, and using `ylim = c(y1, y2)` within the `plot` command will do the same for the y-axis.

Create a plot of queue length versus time with the x-axis running from 0 to 50 and the y-axis running from 0 to 50.

Exercise 6.3 You can place several graphs within a single figure by using the `par` function (short for "parameter") to adjust the layout of the plot. For example, the command

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

divides the plotting area into 2 rows and 3 columns. As R draws a series of graphs, it places them along the top row from left to right, then along the next row, and so on. `mfcol = c(2, 3)` has the same effect except that R draws successive graphs down the first column, then down the second column, and so on.

Save `bucketModel.R` with a new name and modify the program as follows. Use `mfrow = c(1, 2)` to create side-by-side graphs of queue length vs. time for the deterministic (exactly 10 new people per hour) and stochastic ($X \sim \text{Poisson}(\lambda = 10)$ new people per hour). Use the `ylim` argument to make sure the y-axis scales are the same in each plot.

Exercise 6.4 Use `?par` to read about other plot control parameters that you use `par` to set (you should definitely skim — this is one of the longest help files in the whole R system!). Then draw a 2×2 set of plots, each showing the line $y = 5x + 3$ with x running from 3 to 8, but with 4 different line styles and 4 different line colors.

Exercise 6.5 Modify one of your scripts so that at the very end it saves the plot to disk. Use either `savePlot` or `dev.print`. Use `?savePlot`, `?dev.print` to read about these functions. Note that the argument `filename` can include the path to a folder; for example, in Windows you can use `filename="c:/temp/Intro2Figure"`.

(These are really exercises in using the help system, with the bonus that you learn some things about `plot`. (Let's see, we know `plot` can graph a simulation (`N` versus `tvec` and all that). Maybe it can also draw a line to connect the points, or just draw the line and leave out the points. That would be useful. So let's try `?plot` and see if it says anything about lines ... Hey, it also says that `graphical parameters can be given as arguments to plot`, so maybe I can set line colors inside the plot command instead of using `par` all the time ...) The help system can be quite helpful (amazingly enough) once you get used to it and get in the habit of using it often.)

The main point is not to be afraid of experimenting; if you have saved your previous commands in a script file, there's almost nothing you can break by trying out commands and inspecting the results.

Exercise 6.6 * Read about `lines` and `legend`. Using everything you have learned so far, recreate Figure 1.5 on pp. 15 of Mooney and Swift's book. Do not worry if your graph does not look *exactly* the same, just that it conveys the same information and uses the same graphical approaches (e.g. line graph with points; legend). However, students who come closest to an exact reproduction will receive a higher mark.

7 The R package system

R has many extra packages that provide extra functions. You are able to install new packages from a menu within RStudio (`Tools: Install Packages...`). You can also 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) 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.

You will frequently 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

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

Table 2: 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.

main help system index files couldn't be updated because of file permissions problems.

8 Statistics in R

Some of the important functions and packages (collections of functions) for statistical modeling and data analysis are summarized in Table 2. [2] 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`). For the most part, we will not be concerned here with this side of R.

9 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 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.

9.1 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 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 9.1* 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
```

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

Table 3: 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`.

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

9.2 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 extracted 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 9.2 * Write a *one-line* command to extract a vector consisting of the second, first, and third elements of `z` *in that order*.

Exercise 9.3 Write a script file that computes values of $y = \frac{(x-1)}{(x+1)}$ for $x = 1, 2, \cdots, 10$, and plots y versus x with the points plotted and connected by a line (hint: in `?plot`, search for `type`).

Exercise 9.4 * The sum of the geometric series $1 + r + r^2 + r^3 + ... + r^n$ approaches the limit $1/(1-r)$ for $r < 1$ as $n \to \infty$. Set the values $r = 0.5$ and $n = 10$, and then write a **one-line** command that creates the vector $G = (r^0, r^1, r^2, ..., r^n)$. 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 = 1.9999999999999
> 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.)

9.3 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
```

`x < y`	less than
`x > y`	greater than
`x <= y`	less than or equal to
`x >= y`	greater than or equal to
`x == y`	equal to
`x != y`	*not* equal to

Table 4: Some comparison operators in R. Use `?Comparison` to learn more.

```
> 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.

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.

9.4 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!)

As a simple example, we might want to focus on just the values N near the maximum in the leaky bucket example:

```
> (largeN <- N[N > 0.9 * max(N)])

## [1] 38 38 39 42 40

> (largeN_tvec = tvec[N > 0.9 * max(N)])

## [1]  9 13 15 18 19
```

What is really happening here (think about it for a minute) is that `N>0.9*max(N)` generates a logical vector the same length as `N` (`FALSE FALSE FALSE ...`) which is then used to select the appropriate values.

If you want the positions at which `N` is greater than 90% of its maximum, you could say `(1:length(N))[N > 0.9 * max(N)]`, but you can also use a built-in function: `which(N > 0.9 * max(N))`. If you wanted the position at which the maximum value of `N` occurs, you could say `which(N == max(N))`. (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 9.5: What would happen if instead of setting `largeN` you replaced `N` by saying `N <- N[N > 0.9 * max(N)]`, and then `tvec <- tvec[N > 0.9 * max(N)]`? Why would that be wrong? Try it with some temporary variables — set `N2 <- N` and `tvec2 <- tvec` and then play with `N2` and `tvec2` 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):

```
> N[N > 0.9 * max(N) & tvec >= 15]

## [1] 39 42 40

> tvec[N > 0.9 * max(N) & tvec >= 15]

## [1] 15 18 19
```

If we were going to do this a lot, we could save typing by assigning a logical vector

```
> bigN <- N > 0.9 * max(N) & tvec >= 15
```

and using it to do the indexing.

Exercise 9.6 `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 9.7 * 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 9.8 *: Specify two ways to take only the elements in the odd positions (first, third, . . .) of a vector of arbitrary length.

10 Matrices

10.1 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 11.2.) 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 10.1 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 10.2 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.)

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.

10.2 `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,

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

Table 5: Some important functions for creating and working with matrices. Many of these have additional optional arguments; use the help system for full details.

```
> (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 10.3 Verify that `rbind(C, D)` works, `cbind(C, C)` works, but `cbind(C, D)` doesn't. Why not?

10.3 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
```

11 Other structures: Lists and data frames

11.1 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.

11.2 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).

References

[1] Ross Ihaka and Robert Gentleman. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, 1996.

[2] W. N. Venables and B. D. Ripley. *Modern Applied Statistics with S.* Springer, New York, fourth edition, 2002. ISBN 0-387-95457-0.