AN INTRODUCTION TO R FOR ECOLOGICAL MODELING

STEPHEN P. ELLNER, BENJAMIN M. BOLKER, AND AARON A. KING

Contents

Date: December 17, 2014.

1. 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 or identified as a Project) are given 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). It also draws on the documentation supplied with R. Versions of this document have been used in courses taught by BB and AAK at Florida, Michigan, and in the Ecology and Evolution of Infectious Diseases Workshop over several years.

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) and targets biologists who are neither programmers nor statisticians.

2. What is R?

R is a computing environment 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, and

- extensive help and documentation facilities.

R is an open source project, available for free download via the Web (R Core Team, 2014b). 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 modellers) as a platform for making new methods available to users. The commercial implementation of S (called S-PLUS) offers an Office-style "point and click" interface that R lacks. For our purposes, however, the advantage of this front-end is outweighed by the fact that R is built on a faster and much less memory-hungry implementation of S and is easier to interface with other languages. A standard installation of of R also includes extensive documentation, including an introductory manual ($\approx$ 100 pages) and a comprehensive reference manual (over 1000 pages). [There are a number of graphical front-ends for R (see `http://www.r-project.org/GUI/` and `http://www.rstudio.com`). Before learning about these, however, you should learn a little about R itself.]

3. Getting started with R

3.1. **Installing R on your computer.** The main source for R is the Comprehensive R Archive Network (CRAN): `http://cran.r-project.org`. You can get the source code and compile it yourself, but you may prefer at this point to download and install a precompiled version. You can download precompiled binaries for most major platforms from any CRAN mirror. To do so:

- 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-win32.exe` for Windows, `R-x.y.z.dmg` for MacOS, where `x.y.z` is the version number). The binary files are large (30–60 megabytes) — you will need to find a fast internet connection.
- Read and follow the instructions.

Be sure to install the latest version, which is 3.1.2, nicknamed "Pumpkin Helmet".

R should work well on any reasonably recent computer.

For Windows, R is installed by launching the downloaded file and following the on-screen instructions. At the end you'll have an R icon on your desktop that can be used to launch the program. In the following, we've tried to mark Windows-specific items with a ⓦ and *nix (linux/unix/MacOSX)-specific items with a ⓧ.

ⓦ If you are using R on a machine where you have sufficient permissions, you may want to edit some of your graphical user interface (GUI) options.

- To allow command and graphics windows to move independently on the desktop (SDI, single-document interface, rather than MDI, multiple-document interface): go to `File/Edit/Preferences` and click the radio button to set `SDI` instead of `MDI`. This edits the `Rconsole` file. R will ask you where to save it; click through to `My Computer/Program Files/R/R-x.y.z/`, where `x.y.z` stands for the version of R. You will then need to restart R.

The standard distributions of R include several *packages*, user-contributed suites of add-on functions. These notes use some additional packages which you will have to install. In the Windows version additional packages can be installed easily from within R using the **Packages** menu. Under all platforms, you can use the commands `install.packages()` and `update.packages()` to install and update packages, respectively. Most packages are available pre-compiled for MacOS X and Windows; under *nix, R will download and compile the source code for you.

3.2. **Starting R.**

ⓧ Execute `R` from the command line.

ⓦ 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.X.X\bin\Rgui.exe`.

3.3. **Stopping R.**

> When entering, always look for the exit. —Lebanese proverb

You can stop R from the `File` menu (ⓦ), 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); in general, you should say "no" to avoid clutter and unintentional loading of old data.

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 `Esc` key (ⓦ), or `Ctrl-c` (ⓧ).

4. Interactive calculations

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
```

To do anything complicated, the results from calculations have to be stored in (*assigned to*) variables. For example:

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

R automatically creates the variable `a` and stores the result (4) in it, but R 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 *not* print the results. To ask R to print the value, just type the variable name by itself

```
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* (an ordered list), and it is *numeric* because we gave R a number rather than (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 (this often happens 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 `Ctrl-c` (ⓧ), the `Esc` key or the stop icon in the menu bar (ⓦ) to get out.

You can assign values to variables in R using the `<-` operator. There are several alternative forms for assignment. Each of these three commands accomplishes the same thing; the first is the preferred form, however.

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

```
a = 2+2
```

Variable names in R must begin with a letter, followed by alphanumeric characters. You can break up long names with a period, as in `long.variable.number.3`, an underscore (`very_very_long_variable_name`), or by using camel case (`quiteLongVariableName`); you cannot use blank spaces in variable names. R is case sensitive: `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`, and `T`, which are either built-in R functions or hard to tell apart.

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
z2 <- x/y
z3 <- x^y
z1; z2; z3

## [1] 10
## [1] 2.5
## [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` or `print(x); print(y)` to display the values.

You can retrieve and edit previous commands. The up-arrow (↑) key or `Ctrl-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 ↓, →, ←, `Home` and `End` keys too.

You can combine several operations in one calculation:

```
A <- 3
C <- (A+2*sqrt(A))/(A+5*sqrt(A)); C

## [1] 0.5543706
```

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, (3) multiplication and division, (4) addition and subtraction.

```
> 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 it's best to **use 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 if 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 1. Using editing shortcuts wherever you can, use R to 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, replace them with regular parentheses ().)
(2) (a) $1+0.2$
    (b) $1+0.2+0.2^2/2$
    (c) $1+0.2+0.2^2/2+0.2^3/6$
    (d) $e^{0.2}$ (remembering 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(x=c(1,2))` should give you the values for the standard normal for $x=1$ and $x=2$.)

5. 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.

Other (on-line) help resources. Paul Johnson's "R tips" web page (http://pj.freefaculty.org/R/Rtips.html) answers a number of "how do I ...?" questions, although it's out of date in some places. The R "wiki" (http://wiki.r-project.org) is a newer venue for help information. The R tips page is (slowly) being moved over to the wiki, which will eventually (we hope) contain a lot more information. You can also edit the wiki and add your own pages!

Also see:

`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.

- R reference card: http://cran.r-project.org/doc/contrib/Short-refcard.pdf
- Mathematica to R reference: http://wiki.r-project.org/rwiki/doku.php?id=getting-started:translations:mathematica2r
- Octave (free MATLAB clone) to R: http://wiki.r-project.org/rwiki/doku.php?id=getting-started:translations:octave2r
- R ecology ("environmetrics") task view: cran.r-project.org/web/views/Environmetrics.html
- contributed documentation at CRAN: http://cran.us.r-project.org/other-docs.html

Exercise 2. Do an Apropos on `sin` 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). Note that `??sin` is equivalent to `help.search("sin")`. If you have a net connection, try `RSiteSearch("sin")` from the command line or the equivalent from the menu and see what happens.

6. A FIRST INTERACTIVE SESSION: LINEAR REGRESSION

To get a feel for working in R we'll fit a straight-line model (linear regression) to data. Below are some data on the maximum growth rate $r_{\max}$ of laboratory populations of the green alga *Chlorella vulgaris* as a function of light intensity (μE per m^2 per second). These experiments were run during the system-design phase of the study reported by Fussman et al. (2000).

Light: 20, 20, 20, 20, 21, 24, 44, 60, 90, 94, 101
$r_{\max}$: 1.73, 1.65, 2.02, 1.89, 2.61, 1.36, 2.37, 2.08, 2.69, 2.32, 3.67

To analyze these data in R, first enter them as numerical *vectors*:

```
Light <- c(20,20,20,20,21,24,44,60,90,94,101)
rmax <- c(1.73,1.65,2.02,1.89,2.61,1.36,2.37,2.08,2.69,2.32,3.67)
```

The function `c()` *combines* the individual numbers into a vector. Try recalling (with ↑) and modifying the above command to

```
Light <- 20,20,20,20,21,24,44,60,90,94,101
```

and see the error message you get: in order to create a vector of specified numbers, you **must** use the `c()` function.

To see a histogram of the growth rates enter `hist(rmax)`, which opens a graphics window and displays the histogram. There are **many** other built-in statistics functions: for example `mean(rmax)` gets you the mean, and `sd(rmax)` and `var(rmax)` give the standard deviation and variance, respectively. Play around with these functions, and any others you can think of.

To see how the algal rate of increase varies with light intensity, type

```
plot(Light,rmax)
```

to plot `rmax` (y) against `Light` (x). Based on what you see, does it seem reasonable to hypothesize a linear relationship between these variables? **Don't close this plot window:** we'll soon be adding to it.

To perform linear regression we create a linear model using the `lm()` function:

```
fit <- lm(rmax~Light)
```

(Note that the variables in the formula `rmax Light` appear in an order *opposite* to that in `plot()` command above. In the formula, `rmax` is the response variable and `Light` is the predictor.)

The `lm` command created an *object* named `fit`. In R, an *object* is a data structure consisting of multiple parts. In this case, `fit` holds the results of the linear regression analysis. Unlike most statistics packages, R rarely summarizes an analysis for you by default. Statistical analyses in R are done by creating a model, and then giving additional commands to extract desired information about the model or display results graphically.

To get a summary of the results, enter the command `summary(fit)`. R sets up model objects (more on this later) so that the function `summary()` "knows" that `fit` was created by `lm()`, and produces an appropriate summary of results for an `lm()` object:

```
summary(fit)
```

```
##
## Call:
## lm(formula = rmax ~ Light)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -0.5478 -0.2607 -0.1166  0.1783  0.7431
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 1.580952   0.244519   6.466 0.000116 ***
## Light       0.013618   0.004317   3.154 0.011654 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.4583 on 9 degrees of freedom
## Multiple R-squared:  0.5251,Adjusted R-squared:  0.4723
## F-statistic: 9.951 on 1 and 9 DF,  p-value: 0.01165
```

If you've had (and remember) a statistics course the output will make sense to you. The table of coefficients gives the estimated regression line as $\texttt{rmax} = 1.58 + 0.0136 \times \texttt{Light}$. Associated with each coefficient is the standard error of the estimate, the t-statistic value for testing whether the coefficient is nonzero, and the p-value corresponding to the t-statistic. Below the table, the adjusted R^2 gives the estimated fraction of the variance explained by the regression line, and the p-value in the last line is an overall test for significance of the model against the null hypothesis that the response variable is independent of the predictors.

You can add the regression line to the plot of the data with a function taking `fit` as its input (if you closed the plot of the data, you will need to create it again in order to add the regression line):

```
abline(fit)
```

(`abline`, pronounced "a b line", is a general-purpose function for adding lines to a plot: you can specify horizontal or vertical lines, a slope and an intercept, or a regression model: `?abline`).

You can get the coefficients by using the `coef()` function:

```
coef(fit)
```

```
## (Intercept)       Light
##  1.58095214  0.01361776
```

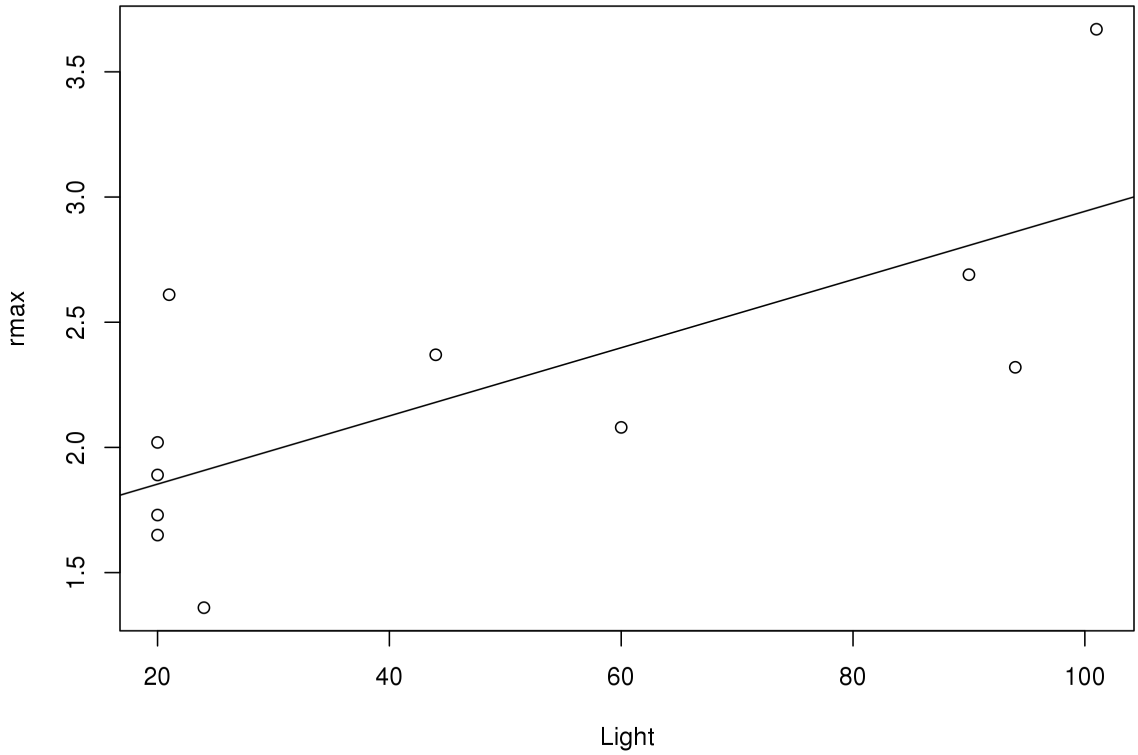


FIGURE 1. Graphical summary of regression analysis

You can also also "interrogate" `fit` directly. Type `names(fit)` to get a list of the components of `fit`, and then extract components according to their names using the `$` symbol.

```
names(fit)

##  [1] "coefficients"  "residuals"     "effects"       "rank"
##  [5] "fitted.values" "assign"        "qr"            "df.residual"
##  [9] "xlevels"       "call"          "terms"         "model"
```

You can get the regression coefficients this way:

```
fit$coefficients

## (Intercept)       Light
##  1.58095214  0.01361776
```

`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`	Linear and nonlinear mixed-effects models (repeated measures, block effects, spatial models): in package `nlme`
`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.

It's good to be able to look inside R objects when necessary, but all other things being equal you should prefer (e.g.) `coef(x)` to `x$coefficients`. For more information (perhaps more than you want) about `fit`, use `str(fit)` (for **str**ucture).

7. Statistics in R

Some of the important functions and packages (collections of functions) for statistical modeling and data analysis are summarized in Table 2. The book *Modern Applied Statistics with S* (Venables and Ripley, 2002) gives a good practical overview, and a list of available packages and their contents can be found at the main R website (http://cran.r-project.org, and click on `Packages`).

8. 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, e.g.,

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

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. It is safe to answer "yes" here.

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

9. Vectors

The most basic data-type in R is the vector. A vector is just a 1-dimensional array of values. Several different kinds of vectors are available: (1) numerical vectors, (2) logical vectors, (3) character-string vectors, (4) factors, (5) ordered factors, and (6) lists. Lists are treated a bit differently in R than the other kinds, so we'll postpone talking about them until later.

Besides its *class*—which kind of vector it is—a vector's only other necessary attribute is its length. Optionally, vectors in R can have a `names` attribute, which allows you to refer to entries by name.

We've already how to create vectors in R using the `c` function, e.g.,

```
x <- c(1,3,5,7,9,11)
y <- c(6.5,4.3,9.1,-8.5,0,3.6)
z <- c("dog","cat","dormouse","chinchilla")
w <- c(a=4,b=5.5,c=8.8)
```

```
length(x)

## [1] 6

mode(y)

## [1] "numeric"
```

```
mode(z)

## [1] "character"

names(w)

## [1] "a" "b" "c"
```

The nice thing about having vectors as a basic type is that many operations in R are efficiently *vectorized*. That is, the operation acts on the vector as a unit, saving you the trouble of treating each entry individually. For example:

```
x <- x+1
xx <- sqrt(x)
x; xx

## [1]  2  4  6  8 10 12
## [1] 1.414214 2.000000 2.449490 2.828427 3.162278 3.464102
```

Notice that the operations were applied to every entry in the vector. Similarly, commands like `x-5, 2*x, x/10`, and `x^2` apply subtraction, multiplication, and division to each element of the vector. The same is true for operations involving multiple vectors:

```
x+y

## [1]  8.5  8.3 15.1 -0.5 10.0 15.6
```

Exercise 3. What do the `%%` and `%/%` operators do?

In R the default is to apply functions and operations to vectors in an *element by element* manner; anything else (e.g. matrix multiplication) is done using special notation (discussed below).

Warning: element recycling. R has a very useful, but unusual and perhaps unexpected, behavior when two vector operands in a vectorized operation are of unequal lengths. It will effectively extend the shorter vector using element "re-cycling": re-using elements of the shorter vector. Thus

```
x <- c(1,2,3)
y <- c(10,20,30,40,50,60)
x+y

## [1] 11 22 33 41 52 63

y-x
```

```
## [1]  9 18 27 39 48 57
```

Exercise 4. What happens when the length of the longer vector is not a multiple of that of the shorter?

9.1. **Functions for creating vectors.** A set of regularly spaced values can be created with the `seq` function, whose syntax is `x <- seq(from,to,by)` or `x <- seq(from,to)` or `x <- seq(from,to,length.out)`. 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; and the third form creates a vector with the desired endpoints and length. There is also a shortcut for creating vectors with `by=1`:

```
1:8

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

Exercise 5. 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 .

Exercise 6. What happens when `to` is less than `from` in `seq`? This is one of the first "gotchas" R newbies run into.

A constant vector such as `(1 1 1 1)` can be created with `rep` function, whose 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:

`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` `b` times each
`hist(v)`	Histogram plot of value in v
`mean(v),var(v),sd(v)`	Estimate of population mean, variance, standard deviation based on data values in `v`
`cov(v,w)`	Covariance between two vectors
`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`.

```
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 you'll get used to it—and it will turn out to be useful.

Some useful functions for creating and working with vectors are listed in Table 3.

9.2. **Vector indexing.** It is often necessary 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 by giving a vector of indices:

```
v <- z[c(2,3,4,5)]
```

or

```
v <- z[2:5]; v

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

This has extracted the 2nd through 5th elements in the vector.

Exercise 7. If you enter `v <- z[seq(1,5,2)]`, what will happen? Make sure you understand why.

Extracted parts of a vector don't have to be regularly spaced. For example

```
v <- z[c(1,2,5)]; v

## [1] 1 3 9
```

Indexing is also used 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 1st, 3rd, and 5th values.

Elements in a named vector can be accessed and modified by name as well as by position. Thus

```
w

##   a   b   c
## 4.0 5.5 8.8

w["a"]

## a
## 4

w[c("c","b")]

##   c   b
## 8.8 5.5

w["b"] <- 0
w

##   a   b   c
## 4.0 0.0 8.8
```

Exercise 8. Write a *one-line* command to extract a vector consisting of the second, first, and third elements of `z` *in that order*.

Exercise 9. What happens when I set the value of an element that doesn't exist? For example, try

```
z[9] <- 11
```

Exercise 10. Write code 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.

Warning: unavoidable imprecision. Note that comparing very similar numeric values can be tricky: rounding can happen, and some numbers cannot be represented exactly on a digital computer. By default, R displays 7 significant digits (`options("digits")`).

```
x <- 1.999999; x; x-2

## [1] 1.999999
## [1] -1e-06

x <- 1.9999999999999; x; x-2

## [1] 2
## [1] -9.992007e-14

x <- 1.99999999999999999; x; x-2

## [1] 2
## [1] 0
```

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. What happened in the third case?

Exercise 11. The sum of the geometric series $1 + r + r^2 + r^3 + ... + r^n$ approaches the limit $1/(1-r)$ for $r < 1$ as $n \to \infty$. Take $r = 0.5$ and $n = 10$, and 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 $n = 50$.

9.3. **Logical operations.** Some operations return a logical value (i.e., `TRUE` or `FALSE`). For example, try:

```
a <- 1; b <- 3;
c <- a < b
d <- (a > b)
c; d

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

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.

The parentheses around `a > b` above are optional but do make the code easier to read. Be careful when you make comparisons with negative values: `a<-1` may surprise you by setting `a=1`, because `<-` is the assignment operator in R. Use `a< -1` or `a<(-1)` to make this comparison.

When we compare two vectors or matrices, comparisons are done element-by-element (and the recycling rule applies). For example,

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

## [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)` or `all.equal(x,y)`. You can use `?Logical` to read more about logical operations. **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 signify logical negation; `!=` (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.

More complicated conditions are built by using **logical operators** to combine comparisons:

`!`	logical NOT
`&`	logical AND, elementwise
`&&`	logical AND, first element only
`|`	logical OR, elementwise
`||`	logical OR, first element only
`xor(x,y)`	exclusive OR, elementwise

The two forms of logical OR (`|` and `||`) are *non-exclusive*, meaning that `x|y` is true if either `x` or `y` or both are true. 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. Use `xor` when exclusive OR is needed. 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. **More on vector indexing.** 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.

As a simple example, we might want to focus on just the low-light values of $r_{\max}$ in the *Chlorella* example:

```
Light <- X[,1]
rmax <- X[,2];
lowLight <- Light[Light<50]
lowLightrmax <- rmax[Light<50]
lowLight

## [1] 20 20 20 20 21 24 44

lowLightrmax

## [1] 1.73 1.65 2.02 1.89 2.61 1.36 2.37
```

What is really happening here (think about it for a minute) is that `Light<50` generates a logical vector the same length as `Light` (`TRUE TRUE TRUE ...`) which is then used to select the appropriate values.

If you want the positions at which `Light` is lower than 50, you could say `(1:length(Light))[Light<50]`, but you can also use a built-in function: `which(Light<50)`. If you wanted the position at which the maximum value of `Light` occurs, you could say `which(Light==max(Light))`. (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).

(What would happen if instead of setting `lowLight` you replaced `Light` by saying `Light <- Light[Light<50]`? Why would that be the wrong thing to do?)

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

```
Light[Light<50 & rmax <= 2.0]

## [1] 20 20 20 24

rmax[Light<50 & rmax <= 2.0]

## [1] 1.73 1.65 1.89 1.36
```

Exercise 12. `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 find the subset of those numbers that is less than the mean.

***Exercise 13.** 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

x[c('second','first')] <- c(8,9); x

##  first second  third
##      9      8      2
```

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

10. Matrices and arrays

10.1. **Creating matrices.** A matrix is a two-dimensional array of items. Most straightforwardly, we can create a matrix by specifying the number of rows and columns, and specifying the entries. For example

```
X <- matrix(c(1,2,3,4,5,6),nrow=2,ncol=3); X

##      [,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. Note that the values in the data vector are put into the matrix *column-wise*, by default. You can change this by using the optional parameter `byrow`:

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

##      [,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 need be, to fill a matrix of the specified size. So for example

```
matrix(1,nrow=50,ncol=50)
```

creates a 50×50 matrix, every entry of which is 1.

Exercise 15. 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
```

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

Exercise 16. Use `rnorm` and `matrix` to create a 5×7 matrix of Gaussian random numbers with mean 1 and standard deviation 2.

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 1s on the diagonal and 0 everywhere else.

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

```
A <- matrix(0,3,4); data.entry(A)
```

will create `A` as a 3×4 matrix, and then call up a spreadsheet-like interface in which the values can be edited directly.

`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.

10.2. **`cbind` and `rbind`.** If their sizes match, vectors can be combined to form matrices, and matrices can be combined with vectors or matrices to form other matrices. The functions that do this are `cbind` and `rbind`.

`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); C

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

Remember that 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); D

##      [,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 17. 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 (2^{nd} row, 3^{rd} column) entry of the matrix **A** that you recently created, and

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

## [1] 5 6

B <- A[2:3,1:2]; B

##      [,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,]; first.row

## [1] 1 2 3

second.column <- A[,2]; second.column;

## [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; A

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

The same can be done with blocks, rows, or columns, for example

```
A[1,] <- c(2,4,5); A

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

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
## [2,]   3   3
```

10.4. **Arrays.** The generalization of the matrix to more (or less) than 2 dimensions is the array. In fact, in R, a matrix is nothing other than a 2-dimensional array. How does R store arrays? In the simplest possible way: an array is just a vector plus information on the size of the array. Most straightforwardly, we can create an array from a vector:

```
X <- array(1:24,dim=c(3,4,2)); X

## , , 1
```

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

Note, again, that the arrays are filled in a particular order: the first dimension first, then the second, and so on. A one-dimensional array is subtly different from a vector:

```
y <- 1:5; y

## [1] 1 2 3 4 5

z <- array(1:5,dim=5); z

## [1] 1 2 3 4 5

y==z

## [1] TRUE TRUE TRUE TRUE TRUE

identical(y,z)

## [1] FALSE

dim(y); dim(z)

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

Exercise 18. What happens when we set the dimension attribute on a vector? For example:

```
x <- seq(1,27)
dim(x) <- c(3,9)
is.array(x)
is.matrix(x)
```

11. Factors

For dealing with measurements on the nominal and ordinal scales (Stevens, 1946), R provides vectors of type *factor*. A factor is a variable that can take one of a finite number of distinct *levels*. To construct a factor, we can apply the `factor` function to a vector of any class:

```
x <- rep(c(1,2),each=3); factor(x)

## [1] 1 1 1 2 2 2
## Levels: 1 2

trochee <- c("jetpack","ferret","pizza","lawyer")
trochee <- factor(trochee); trochee

## [1] jetpack ferret  pizza   lawyer
## Levels: ferret jetpack lawyer pizza
```

By default, `factor` sets the levels to the unique set of values taken by the vector. To modify that behavior, there is the `levels` argument:

```
factor(trochee,levels=c("ferret","pizza","cowboy","scrapple"))

## [1] <NA>   ferret pizza  <NA>
## Levels: ferret pizza cowboy scrapple
```

Note that the order of the levels is arbitrary, in keeping with the fact that the only operation permissible on the nominal scale is the test for equality. In particular, the factors created with the `factor` command are un-ordered: there is no sense in which we can ask whether, e.g., `ferret < cowboy`.

To represent variables measured on the ordinal scale, R provides *ordered factors*, constructed via the `ordered` function. An ordered factor is just like an un-ordered factor except that the order of the levels matters:

```
x <- ordered(sample(x=letters,size=22,replace=TRUE)); x

##  [1] q a v u d o j p w z e k l t g f g t r g f c
## 18 Levels: a < c < d < e < f < g < j < k < l < o < p < q < r < t < ... < z
```

Here, we've relied on `ordered`'s default behavior, which is to put the levels in alphabetical order. It's typically safer to specify explicitly what order we want:

```
x <- ordered(x,levels=rev(letters))
x[1:5] < x[18:22]

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

Exercise 19. Look up the documentation on the `sample` function used above.

Exercise 20. Can I make a matrix or an array out of a factor variable?

***Exercise 21.** What is the internal representation of factors in R? Try converting factors to integers using `as.integer`. Try converting an integer vector to a factor using `factor`.

12. Other structures: Lists and data frames

12.1. **Lists.** While vectors and matrices may seem familiar, lists may be 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 is a little different too: use `[[ ]]` 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=trochee,C=c("a","b","c"))
```

Then `L$A`, `L[["A"]]`, and `L[[1]]` will each return the first element of the list. To extract a sublist, use the ordinary single square brackets: `[]`:

```
L[c("B","C")]

## $B
## [1] jetpack ferret  pizza   lawyer
## Levels: ferret jetpack lawyer pizza
##
## $C
## [1] "a" "b" "c"
```

12.2. **Data frames.** Vectors, matrices, and lists of one sort or another are found in just about every programming language. The *data frame* structure is (or was last time I checked) unique to R, and is central to many of R's useful data-analysis features. It's very natural to want to store data in vectors and matrices. Thus, in the example above, we stored measurements of two variables ($r_{\max}$ and light level) in vectors. This was done in such a way that the observations of the first replicate were stored in the first element of each vector, the second in the second, and so on. To explicitly bind together observations corresponding to the same replicate, we might join the two vectors into a matrix using `cbind`. In the resulting data structure, each row would correspond to an observation, each column to a variable. This is possible, however, only because both variables are of the same type: they're both numerical. More commonly, a data set is made up of several different kinds of variables. The data frame is R's solution to this problem.

Data frames are a hybrid of lists and vectors. Internally, they are a list of vectors which can be of different types but must all be the same length. However, they behave somewhat like matrices, in that you can do most things to them 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. You can turn a data frame into a matrix (using `as.matrix()`, but only if all variables are of the same class) and a matrix into a data frame (using `as.data.frame()`).

When data are read into R from an external file using one of the `read.xxx` commands (`read.csv`, `read.table`, `read.xls`, etc.), the object that is created is a data frame.

```
course.url <- "http://kinglab.eeb.lsa.umich.edu/ICTPWID/SaoPaolo_2015/"
dat <- read.csv(file.path(course.url,"data/ChlorellaGrowth.csv"),
                comment.char='#')
dat
```

```
##    light rmax
## 1     20 1.73
## 2     20 1.65
## 3     20 2.02
## 4     20 1.89
## 5     21 2.61
## 6     24 1.36
## 7     44 2.37
## 8     60 2.08
## 9     90 2.69
## 10    94 2.32
## 11   101 3.67
```

Exercise 22. Download the `hurricanes.csv` file from the above course URL. Examine the resulting data frame by printing it and using the `str` command. Note the class type of each variable.

13. Probability distributions in R

R contains a great deal of distilled knowledge about probability distributions. In particular, for each of a large class of important distributions, methods to compute probability distribution functions (p.d.f., i.e., density or mass functions), cumulative distribution functions (c.d.f.), and quantile functions are available, as are methods for simulating these distributions (i.e., drawing random deviates with the desired distribution). Conveniently, these are all named using the same scheme:

`dxxx(x, ...)`	probability distribution function
`pxxx(q, ...)`	cumulative distribution function
`qxxx(p, ...)`	quantile function (i.e., inverse of `pxxx`)
`rxxx(n, ...)`	simulator

In the above `xxx` stands for the abbreviated name of the specific distribution (the "R name" as given in Table 6). In each case, the `...` indicates that additional, distribution-specific, parameters are to be supplied. Table 6 lists some of the more common distributions built in to R. A complete list of distributions provided by the base stats package can be viewed by executing `?Distributions`.

TABLE 6. Some of the probability distributions built in to R. For complete documentation, execute e.g., `?dbinom` in an R session.

Distribution	R name	Parameters	Range
Discrete distributions			
Binomial	`binom`	size, prob	$0, 1, \ldots,$ size
Bernoulli	`binom (size=1)`	prob	$0, 1$
Poisson	`pois`	lambda	$0, 1, \ldots, \infty$
Negative binomial	`nbinom`	size, (prob or mu)	$0, 1, \ldots, \infty$
Geometric	`geom`	rate	$0, 1, \ldots, \infty$
Hypergeometric	`hyper`	m, n, k	$0, 1, \ldots, m$
Multinomial	`multinom`	size, prob $\in [0,1]^m$	$\{(x_1, \ldots, x_m) : \sum x_i =$ size$\}$
Continuous distributions			
Uniform	`unif`	min, max	$[\min, \max]$
Normal	`norm`	mean, sd	$(-\infty, \infty)$
Gamma	`gamma`	shape, (scale or rate)	$[0, \infty)$
Exponential	`exp`	mu	$[0, \infty)$
Beta	`beta`	shape1, shape2	$[0, 1]$
Lognormal	`lnorm`	meanlog, sdlog	$[0, \infty)$
Cauchy	`cauchy`	location, scale	(∞, ∞)
χ^2	`chisq`	df	$[0, \infty)$
Student's T	`t`	df	$[0, \infty)$
Weibull	`weibull`	shape, scale	$[0, \infty)$

Exercise 23. Write codes to plot the c.d.f. of the binomial distribution (`pbinom`). Verify the relationship between the p.d.f. (`dbinom`) and the c.d.f.

Exercise 24. Plot the p.d.f. and c.d.f. of the Poisson distribution for several values of the parameter λ.

Exercise 25. Plot the p.d.f. and c.d.f. of the gamma distribution for several values of the shape and scale (or rate) parameters. Put in a vertical line to indicate the mean. Be sure to explore both large and small values of the shape parameter.

Exercise 26. Use Rstudio's `manipulate` facility to plot the p.d.f. and c.d.f. of the beta distribution for several values of the first and second shape parameters. Put in a vertical line to indicate the mean. Be sure to explore the full ranges of the parameters.

14. Script files and data files

Modeling and complicated data analysis are often accomplished more efficiently using *scripts*, which are a series of commands stored in a text file. The Windows and MacOS versions of R both have basic script editors. You can also use Windows Notepad or Wordpad, or a more featureful editor like PFE, emacs (with ESS, "emacs speaks statistics"), or Tinn-R: you **should not** use MS Word or other fancy editors (see below).

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

(1) Setup statements.
(2) Input some data from a file.
(3) Carry out the calculations that you want.
(4) Print the results, graph them, and/or save them to a file.

For example, a script file might

(1) Load some packages, or "source" 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 paper.

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. They are also **essential** for making research reproducible.

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

- To let R know where data and script files are located, you have a few choices:
  (1) change your working directory to wherever the file(s) are located before running R.
  (2) change your working directory within an R session to wherever the file(s) are located using the `setwd()` (**set** **w**orking **d**irectory) function, e.g. `setwd("c:/temp")`.
  (3) spell out the path to the file explicitly. Use a single forward slash to separate folders (e.g., ⓦ `"c:/My~Documents/R/script.R"` or ⓧ `"/home/kingaa/projects/hurricanes"`): this works on all platforms.

  The first option is far preferable, since it makes your codes portable: as long as you copy the whole directory, you can copy it whereever you like and R will find what it needs.

ⓦIf you have a shortcut defined for R on your desktop (or possibly in the Start menu) you can *permanently* change your default working directory by right-clicking on the shortcut icon, selecting `Properties`, and changing the starting directory to somewhere like (for example) `My Documents/projectX`.

- It's important that script files be preserved as *plain text* and data files also as plain text, using comma- or tab-separated format. 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 or translate them into something other than plain text.
  (2) if your web browser uses "file associations" (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 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 are careful to send them as plain text, lines will occasionally get broken in different places, which can lead to confusion. Beware.

As a first example, the file `Intro1.R` has the commands from the interactive regression analysis.

Download `Intro1.R` from the course dropsite,

kinglab.eeb.lsa.umich.edu/ICTPWID/SaoPaolo_2015,

and save it to your computer. Open **your copy** of `Intro1.R`. In your editor, select and Copy the entire text of the file, and then Paste the text into the R console window. 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. Cut-and-Paste 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("Intro1.R")
```

Another important time-saver is loading data from a text file. Grab copies of `Intro2.R` and `ChlorellaGrowth.csv` from the dropsite to see how this is done. In `ChlorellaGrowth.csv` the two variables are entered as columns of a data matrix. Then instead of typing these in by hand, the command

```
X <- read.csv("ChlorellaGrowth.csv",comment.char='#')
```

reads the file (from the current directory) and puts the data values into the variable `X`. **Note** that as specified above you need to make sure that R is looking for the data file in the right place: either move the data file to your current working directory, or change the

line so that it points to the actual location of the data file. Note also the `comment.char` option in the call to `read.csv`; this tells R to ignore lines that begin with a `#` and allows us to use self-documenting data files (a very good practice).

Extract the variables from `X` with the commands

```
Light <- X[,1]
rmax <- X[,2]
```

Think of these as shorthand for "`Light` = everything in column 1 of `X`", and "`rmax` = everything in column 2 of `X`" (we'll learn about working with data matrices later). From there on out it's the same as before, with some additions that set the axis labels and add a title.

Exercise 27. Make a copy of `Intro2.R` under a new name, and modify the copy so that it does linear regression of algal growth rate on the natural log of light intensity, `LogLight=log(Light)`, and plots the data appropriately. You should end up with a graph that resembles Fig. 2.

Exercise 28. Run `Intro2.R`, then enter the command `plot(fit)` in the console and follow the directions in the console. Figure out what just happened by entering `?plot.lm` to bring up the help page for the function `plot.lm()` that carries out a `plot()` command for an object produced by `lm()`. (This is one example of how R uses the fact that statistical analyses are stored as model objects. `fit` "knows" what kind of object it is (in this case an object of type `lm`), and so `plot(fit)` invokes a function that produces plots suitable for an `lm` object.) **Answer:** R produced a series of diagnostic plots exploring whether or not the fitted linear model is a suitable fit to the data. In each of the plots, the 3 most extreme points (the most likely candidates for "outliers") have been identified according to their sequence in the data set.

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 axes limits). You can control scaling using the `xlim` and `ylim` arguments in `plot`:

```
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.

Exercise 29. Create a plot of growth rate versus light intensity with the x-axis running from 0 to 120, and the y-axis running from 1 to 4.

Exercise 30. Several graphs can be placed 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(m,n))
```

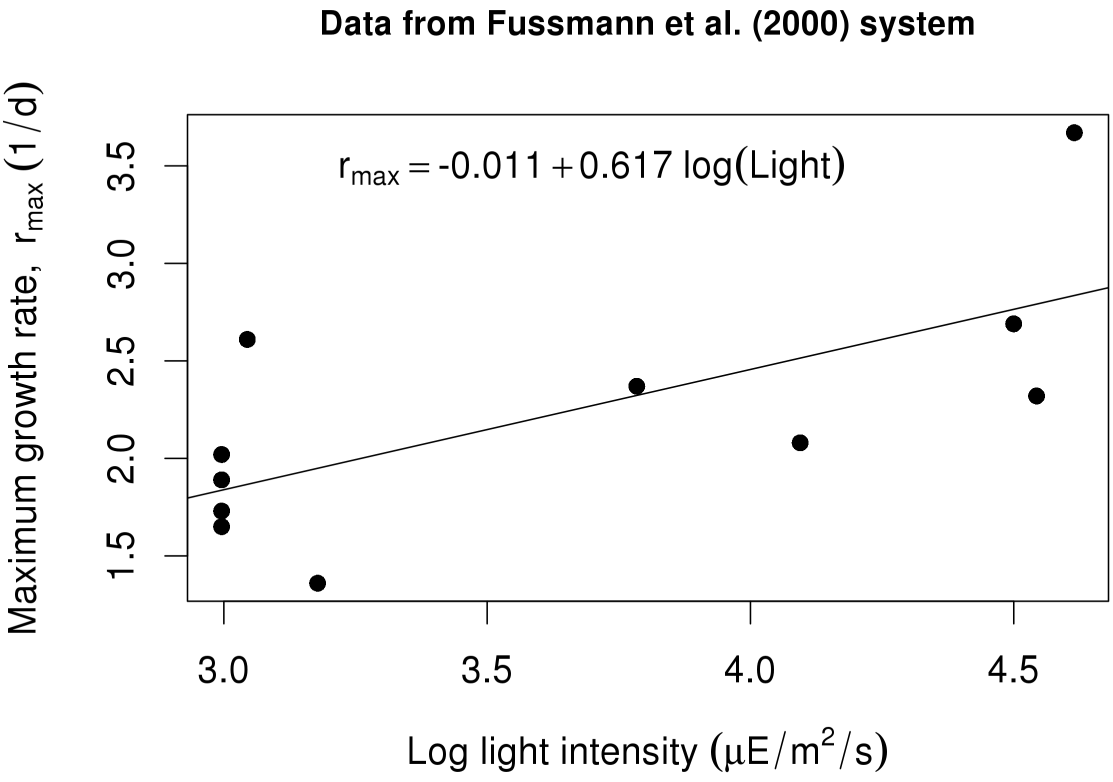


FIGURE 2. Graphical summary of regression analysis using log of light intensity (with plot annotations).

divides the plotting area into m rows and n columns. As a series of graphs is drawn, they are placed along the top row from left to right, then along the next row, and so on. `mfcol=c(m,n)` has the same effect except that successive graphs are drawn down the first column, then down the second column, and so on.

Save `Intro2.R` with a new name and modify the program as follows. Use `mfcol=c(2,1)` to create graphs of growth rate as a function of `Light`, and of `log(growth rate)` as a function of `log(Light)` in the same figure. Do the same again, using `mfcol=c(1,2)`.

***Exercise 31.** Use `?par` to read about other plot control parameters that can be set using `par()`. 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 32.** Modify one of your scripts so that at the very end it saves the plot to disk. You can accomplish this using the `dev.print()` function. Do `?dev.print` to read about this function. Note that you need to specify the `file` argument to write your

graphics to a file; if you don't, it will try and send it to a printer. Also note that many alternative formats are available via the `dev` argument.

15. Looping in R

Very frequently, a computation involves iterating some procedure across a range of cases, and every computer language I've ever come across has one or more facilities for producing such *loops*. R is no exception, though judging by their code, many R programmers look down their noses at loops. The fact remains that, at some point in life, one simply has to write a `for` loop. Here, we'll look at the looping constructs available in R.

15.1. `for` loops.

Execute the following code.

```
phi <- 1
for (k in 1:1000) {
  phi <- 1+1/phi
  print(c(k,phi))
}
```

What does it do? Sequentially, for each value of `k` between 1 and 1000, `phi` is modified. More specifically, at the beginning of the `for` loop, a vector containing all the integers from 1 to 1000 in order is created. Then, `k` is set to the first element in that vector, i.e., 1. Then the R expression from the `{` to the `}` is evaluated. When that expression has been evaluated, `k` is set to the next value in the vector. The process is repeated until, at the last evaluation, `k` has value 1000.

As an aside, note that the final value of `phi` is the Golden Ratio, 1.618034.

As an example of a situation where a loop of some sort is really needed, suppose we wish to iterate the Beverton-Holt map (one of the simplest discrete-time models of population growth),

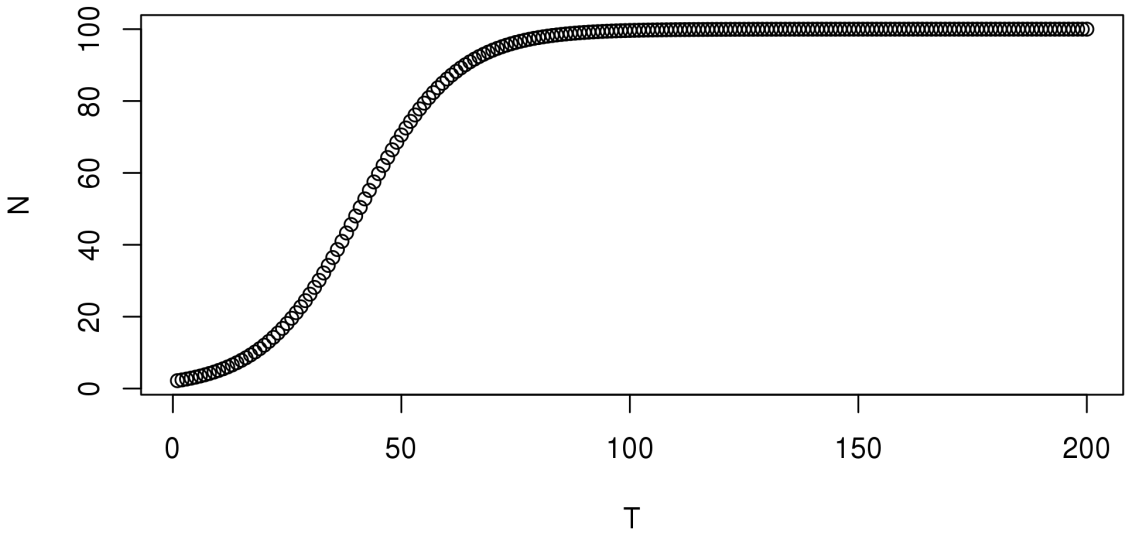
$$N_{t+1} = \frac{a\,N_t}{1 + b\,N_t}.$$

We simply have no option but to do the calculation one step at a time. Here's an R code that does this

```
a <- 1.1
b <- 0.001
T <- seq(from=1,to=200,by=1)
N <- numeric(length(T))
n <- 2
for (t in T) {
  n <- a*n/(1+b*n)
  N[t] <- n
}
```

Spend some time to make sure you understand what happens at each line of the above. We can plot the population sizes N_t through time via

```
plot(T,N)
```



Gotcha: An alternative way to do the above might be something like

```
N <- numeric(length(T))
for (t in 1:length(T)) {
  n <- a*n/(1+b*n)
  N[t] <- n
}
```

Exercise 33. Check that this works with different vectors `T`. What happens when `T` has length 1? What happens when `T` has length 0? Why? To avoid this trap, it's preferable to use `seq_len`, `seq_along`, or `seq.int` instead of the `1:n` construction used above. This and many other R gotchas are described in the useful *R Inferno* (Burns, 2012), a free version of which can be downloaded from the course readings site.

15.2. **`while` loops.** A second looping construct is the `while` loop. Using `while`, we can compute the Golden Ratio as before:

```
phi <- 20
k <- 1
while (k <= 1000) {
```

```
  phi <- 1+1/phi
  print(c(k,phi))
  k <- k+1
}
```

What's going on here? First, `phi` and `k` are initialized. Then the `while` loop is started. At each iteration of the loop, `phi` is modified, and intermediate results printed, as before. In addition, `k` is incremented. The `while` loop continues to iterate until the condition `k <= 1000` is no longer `TRUE`, at which point, the `while` loop terminates.

Note that here we've chosen a large number (1000) of iterations. Perhaps we could get by with fewer. If we wanted to terminate the iterations as soon as the value of `phi` stopped changing, we could do:

```
phi <- 20
conv <- FALSE
while (!conv) {
  phi.new <- 1+1/phi
  conv <- phi==phi.new
  phi <- phi.new
}
```

Exercise 34. Verify that the above works as intended. How many iterations are needed?

Another way to accomplish this would be to use `break` to stop the iteration when a condition is met. For example

```
phi <- 20
while (TRUE) {
  phi.new <- 1+1/phi
  if (phi==phi.new) break
  phi <- phi.new
}
```

While this `while` loop is equivalent to the one before, it does have the drawback that, if the `break` condition is never met, the loop will go on indefinitely. An alternative that avoids this is to use a `for` loop with a large (but necessarily finite) number of iterations, together with `break`:

```
phi <- 3
for (k in seq_len(1000)) {
  phi.new <- 1+1/phi
  if (phi==phi.new) break
  phi <- phi.new
}
```

Exercise 35. Recompute the trajectory of the Beverton-Holt model using a `while` loop. Verify that your answer is exactly equivalent to the one above.

15.3. **`repeat` loops.** A third looping construct in R involves the `repeat` keyword. For example,

```
phi <- 12
repeat {
  phi.new <- 1/(1+phi)
  if (phi==phi.new) break
  phi <- phi.new
}
```

In addition, R provides the `next` keyword, which, like `break`, is used in the body of a looping construct. Rather than terminating the iteration, however, it aborts the current iteration and leads to the immediate execution of the next iteration.

For more information on looping and other control-flow constructs, execute `?Control`.

16. Functions and environments

16.1. **Definitions and examples.** An extremely useful feature in R is the ability to write arbitrary *functions*. A function, in this context, is an algorithm that performs a specific computation that depends on inputs (the function's *arguments*) and produces some output (the function's *value*) and/or has some *side effects*. Let's see how this is done.

Here is a function that squares a number.

```
sq <- function (x) x^2
```

The syntax is `function (arglist) expr`. The one argument in this case is `x`. When a particular value of `x` is supplied, R performs the squaring operation. The function then *returns* the value `x^2`:

```
sq(3); sq(9); sq(-2);

## [1] 9
## [1] 81
## [1] 4
```

Here is a function with two arguments and a more complex *body*, as we call the expression that is evaluated when the function is called.

```
f <- function (x, y = 3) {
  a <- sq(x)
```

```
  a+y
}
```

Here, the body is the R expression from `{` to `}`. Unless the `return` codeword is used elsewhere in the function body, the value returned is always the last expression evaluated. Thus:

```
f(3,0); f(2,2); f(3);

## [1] 9
## [1] 6
## [1] 12
```

Note that in the last case, only one argument was supplied. In this case, `y` assumed its default value, 3.

Note that functions need not be assigned to symbols; they can be *anonymous*:

```
function (x) x^5

## function (x) x^5

(function (x) x^5)(2)

## [1] 32
```

A function can also have side effects, e.g.,

```
hat <- "hat"
hattrick <- function (y) {
  hat <<- "rabbit"
  2*y
}
hat; hattrick(5); hat

## [1] "hat"
## [1] 10
## [1] "rabbit"
```

However, the very idea of a function insists that we should never experience *unintentional* side effects. We'll see how R realizes this imperative below.

An aside. If we want the function not to automatically print, we can wrap the return value in `invisible()`:

```
hattrick <- function (y) {
  hat <<- "rabbit"
```

```
  invisible(2*y)
}
hattrick(5)
print(hattrick(5))

## [1] 10
```

A function in R is defined by three components: (1) its *formal parameters*, i.e., its argument list, (2) its body, and (3) its *environment*, i.e., the context in which the function was defined. R provides simple functions to interrogate these function components:

```
formals(hattrick)

## $y

body(hattrick)

## {
##     hat <<- "rabbit"
##     invisible(2 * y)
## }

environment(hattrick)

## <environment: R_GlobalEnv>
```

16.2. **Function scope.** [1]As noted above, a paramount consideration in the implementation of functions in any programming language is that unintentional side effects should never occur. In particular, I should be free to write a function that creates temporary variables as an aid to its computations, and be able to rest assured that no variables I create temporarily will interfere with any other variables I've defined anywhere else. To accomplish this, R has a specific set of *scoping rules*.

Consider the function

```
f <- function (x) {
  y <- 2*x
  print(x)
  print(y)
  print(z)
}
```

[1]This section draws heavily, and sometimes verbatim, on §10.7 of the *Introduction to R* manual (R Core Team, 2014a).

In this function's body, `x` is a formal parameter, `y` is a local variable, and `z` is a free, or *unbound* variable. When `f` is evaluated, each of these variables must be *bound* to some value. In R, the free variable bindings are resolved—each time the function is evaluated—by first looking in the environment where the function was created. This is called *lexical scope*. Thus if we execute

```
f(3)
```

we get an error, because no object named `z` can be found. If, however, we do

```
z <- 10
f(3)

## [1] 3
## [1] 6
## [1] 10
```

we don't get an error, because `z` is defined in the environment, `<environment: R_GlobalEnv>`, of `f`. Similarly, when we do

```
z <- 13
g <- function (x) {
  2*x+z
}
f <- function (x) {
  z <- -100
  g(x)
}
f(5)

## [1] 23
```

The relevant value of `z` is the one in the environment where `g` was *defined*, not the one in the environment wherein it is *called*.

16.3. **Nested functions and environments.** In each of the following examples, make sure you understand exactly what has happened.

Consider this:

```
y <- 11
f <- function (x) {
  y <- 2*x
  y+x
}
f(1); y

## [1] 3
```

```
## [1] 11
```

Exercise 36. Why hasn't `y` changed its value?

What about this?

```
y <- 0
f <- function (x) {
  2*x+y
}
f(1); y

## [1] 2
## [1] 0
```

Exercise 37. Why is the return value of `f` different?

How about the following?

```
g <- function (y) y
f <- function (x) {
  g <- function (y) {
    2*y
  }
  11+g(x)
}
f(2); g(2)

## [1] 15
## [1] 2
```

Exercise 38. Be sure you understand what's happening at each line and why you get the results you see.

Another example:

```
y <- 11
f <- function (x) {
  y <- 22
  g <- function (x) {
    x+y
  }
  g(x)
}
f(1); y
```

```
## [1] 23
## [1] 11
```

Exercise 39. Which value of `y` was used?

Compare that with this:

```
y <- 11
f <- function (x) {
  y <- 22
  g <- function (x) {
    y <<- 7
    x+y
  }
  c(g(x),y)
}
f(1); y

## [1] 8 7
## [1] 11
```

and this:

```
y <- 11
f <- function (x) {
  g <- function (x) {
     x+y
  }
  g(x)
}
f(1); y

## [1] 12
## [1] 11
```

and this:

```
f <- function (x) {
  y <- 37
  g <- function (x) {
    h <- function (x) {
      x+y
    }
    h(x)
  }
  g(x)
}
```

```
f(1); y

## [1] 38
## [1] 11
```

Finally, consider the following:

```
rm(y)
f <- function (x) {
   g <- function (x) {
     y <<- 2*x
     y+1
   }
   g(x)+1
 }
f(11); y

## [1] 24
## [1] 22

f(-2); y

## [1] -2
## [1] -4
```

Exercise 40. In the above five code snippets, make sure you understand the differences.

As mentioned above, each function is associated with an environment: the environment within which it was defined. When a function is evaluated, a new temporary environment is created, within which the function's calculations are performed. Every new environment has a parent, the environment wherein it was created. The parent of this new environment is the function's environment. To see this, try

```
f <- function () {
  g <- function () {
    h <- function () {
      cat("inside function h:\n")
      cat("current env: ")
      print(sys.frame(sys.nframe()))
      cat("parent env: ")
      print(parent.frame(1))
      cat("grandparent env: ")
      print(parent.frame(2))
      cat("great-grandparent env: ")
      print(parent.frame(3))
      invisible(NULL)
```

```
    }
    cat("inside function g:\n")
    cat("environment of h: ")
    print(environment(h))
    cat("current env: ")
    print(sys.frame(sys.nframe()))
    cat("parent env: ")
    print(parent.frame(1))
    cat("grandparent env: ")
    print(parent.frame(2))
    h()
  }
  cat("inside function f:\n")
  cat("environment of g: ")
  print(environment(g))
  cat("current env: ")
  print(sys.frame(sys.nframe()))
  cat("parent env: ")
  print(parent.frame(1))
  g()
}
cat("environment of f: "); print(environment(f))

## environment of f:
## <environment: R_GlobalEnv>

cat("global env: "); print(sys.frame(sys.nframe()))

## global env:
## <environment: R_GlobalEnv>

f()

## inside function f:
## environment of g: <environment: 0x27603f8>
## current env: <environment: 0x27603f8>
## parent env: <environment: R_GlobalEnv>
## inside function g:
## environment of h: <environment: 0x2757bf0>
## current env: <environment: 0x2757bf0>
## parent env: <environment: 0x27603f8>
## grandparent env: <environment: R_GlobalEnv>
## inside function h:
## current env: <environment: 0x27629b8>
## parent env: <environment: 0x2757bf0>
## grandparent env: <environment: 0x27603f8>
## great-grandparent env: <environment: R_GlobalEnv>
```

Each variable referenced in the function's body is bound, first, to a formal argument if possible. If a local variable of that name has previously been created (via one of the assignment operators `<-`, `->`, or `=`, this is the variable that is affected by any subsequent assignments. If the variable is neither a formal parameter nor a local variable, then the parent environment of the function is searched for that variable. If the variable has not been found in the parent environment, then the grand-parent environment is searched, and so on.

If the assignment operators `<<-` or `->>` are used, a more extensive search for the referenced assignee is made. If the variable does not exist in the local environment, the parent environment is searched. If it does not exist in the parent environment, then the grand-parent environment is searched, and so on. Finally, if the variable cannot be found anywhere along the lineage of environments, a new global variable is created, with the assigned value.

17. The `apply` family of functions

As mentioned above, there are circumstances under which looping constructs are really necessary. Very often, however, we wish to perform some operation across all the elements of a vector, array, or dataset. In such cases, it is faster and more elegant (to the R afficiando's eye) to use the `apply` family of functions.

17.1. **List apply: `lapply`.** `lapply` applies a function to each element of a list or vector, returning a list.

```
  x <- list("teenage","mutant","ninja","turtle",
            "hamster","plumber","pickle","baby")
  lapply(x,nchar)
```

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

```
##
## [[7]]
## [1] 6
##
## [[8]]
## [1] 4
```

```
  y <- c("teenage","mutant","ninja","turtle",
         "hamster","plumber","pickle","baby")
  lapply(y,nchar)
```

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

17.2. **Sloppy list apply: `sapply`.** `sapply` isn't content to always return a list: it attempts to simplify the results into a non-list vector if possible.

```
  x <- list("teenage","mutant","ninja","turtle",
            "hamster","plumber","pickle","baby")
  sapply(x,nchar)
```

```
## [1] 7 6 5 6 7 7 6 4
```

```
  y <- c("teenage","mutant","ninja","turtle",
         "hamster","plumber","pickle","baby")
  sapply(y,nchar)
```

```
## teenage  mutant   ninja  turtle hamster plumber  pickle    baby
##       7       6       5       6       7       7       6       4
```

17.3. **Multiple-list apply: `mapply`.** `mapply` is a multiple-argument version of `sapply`:

```
  x <- c("teenage","mutant","ninja","turtle")
  y <- c("hamster","plumber","pickle","baby")
  mapply(paste,x,y,sep="/")

##           teenage            mutant             ninja            turtle
## "teenage/hamster"  "mutant/plumber"    "ninja/pickle"     "turtle/baby"
```

As usual, the recycling rule applies:

```
  mapply(paste,x,y[2:3])

##           teenage            mutant             ninja            turtle
## "teenage plumber"   "mutant pickle"   "ninja plumber"   "turtle pickle"

  mapply(paste,x[c(1,3)],y)

##           teenage             ninja              <NA>              <NA>
## "teenage hamster"   "ninja plumber"  "teenage pickle"      "ninja baby"
```

17.4. **Array apply: `apply`.** `apply` is very powerful and a bit more complex. It allows an arbitrary function to applied to each slice of an array, where the slices can be defined in all possible ways. Let's create a matrix:

```
  A <- array(data=seq_len(15),dim=c(3,5)); A

##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    4    7   10   13
## [2,]    2    5    8   11   14
## [3,]    3    6    9   12   15
```

To apply an operation to each row, we *marginalize* over the first dimension (rows). For example, to sum the rows, we'd do

```
  apply(A,1,sum)

## [1] 35 40 45
```

To sum the columns (the second dimension), we'd do

```
  apply(A,2,sum)

## [1]  6 15 24 33 42
```

Now suppose we have a 3-dimensional array:

```
  A <- array(data=seq_len(30),dim=c(3,5,2)); A

## , , 1
##
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    4    7   10   13
## [2,]    2    5    8   11   14
## [3,]    3    6    9   12   15
##
## , , 2
##
##      [,1] [,2] [,3] [,4] [,5]
## [1,]   16   19   22   25   28
## [2,]   17   20   23   26   29
## [3,]   18   21   24   27   30
```

To sum the rows within each slice, we'd do

```
  apply(A,c(1,3),sum)

##      [,1] [,2]
## [1,]   35  110
## [2,]   40  115
## [3,]   45  120
```

while to sum the slices, we'd do

```
  apply(A,3,sum)

## [1] 120 345
```

For each of the above, make sure you understand exactly what has happened.

Of course, we can apply an anonymous function wherever we apply a named function:

```
  apply(A,c(2,3),function (x) sd(x)/sqrt(length(x)))

##           [,1]      [,2]
## [1,] 0.5773503 0.5773503
## [2,] 0.5773503 0.5773503
## [3,] 0.5773503 0.5773503
```

```
## [4,] 0.5773503 0.5773503
## [5,] 0.5773503 0.5773503
```

Additional arguments are passed to the function:

```
  apply(A,c(1,2),function (x, y) sum(x>y),y=8)

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

  apply(A,c(1,2),function (x, y) sum(x>y),y=-1)

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

17.5. **Table apply: `tapply`.** `tapply` is, in a way, an extension of `table`. The syntax is `tapply(X,INDEX,FUN,...)`, where `X` is a vector, `INDEX` is a list of one or more factors, each the same length as `X`, and `FUN` is a function. The vector `X` will be split into subvectors according to `INDEX`, and `FUN` will be applied to each of the subvectors. By default, the result is simplified into an array if possible. Some examples:

```
  x <- seq(1,30,by=1)
  b <- rep(letters[1:10],times=3)
  data.frame(x,b)

##     x b
## 1   1 a
## 2   2 b
## 3   3 c
## 4   4 d
## 5   5 e
## 6   6 f
## 7   7 g
## 8   8 h
## 9   9 i
## 10 10 j
## 11 11 a
## 12 12 b
## 13 13 c
## 14 14 d
## 15 15 e
## 16 16 f
## 17 17 g
```

```
## 18 18 h
## 19 19 i
## 20 20 j
## 21 21 a
## 22 22 b
## 23 23 c
## 24 24 d
## 25 25 e
## 26 26 f
## 27 27 g
## 28 28 h
## 29 29 i
## 30 30 j
```

```
  tapply(x,b,sum)
```

```
##  a  b  c  d  e  f  g  h  i  j
## 33 36 39 42 45 48 51 54 57 60
```

```
  b <- rep(letters[1:10],each=3)
  data.frame(x,b)
```

```
##     x b
## 1   1 a
## 2   2 a
## 3   3 a
## 4   4 b
## 5   5 b
## 6   6 b
## 7   7 c
## 8   8 c
## 9   9 c
## 10 10 d
## 11 11 d
## 12 12 d
## 13 13 e
## 14 14 e
## 15 15 e
## 16 16 f
## 17 17 f
## 18 18 f
## 19 19 g
## 20 20 g
## 21 21 g
## 22 22 h
## 23 23 h
## 24 24 h
```

```
## 25 25 i
## 26 26 i
## 27 27 i
## 28 28 j
## 29 29 j
## 30 30 j
```

```
  tapply(x,b,sum)
```

```
##  a  b  c  d  e  f  g  h  i  j
##  6 15 24 33 42 51 60 69 78 87
```

```
  course.url <- "http://kinglab.eeb.lsa.umich.edu/ICTPWID/SaoPaolo_2015/"
  datafile <- "data/seedpred.dat"
  seeds <- read.table(paste0(course.url,datafile),header=TRUE,
                      colClasses=c(station='factor',dist='factor',date='Date'))
  x <- subset(seeds,available>0)
  with(x, tapply(tcum,list(dist,station),max,na.rm=TRUE))
```

```
##     1  10 100 101 102 103 104 105 106 107 108 109 11 110 111 112 113 114
## 10 24 248 122 248  10  17  39  17  46  10   3  17 NA  10  10  17 248 248
## 25 18 123  11 249 249  11  11 249  81  11 123 249  4  25 249  47 249 249
##    115 116 117 118 119  12 120 121 122 123 124 125 126 127 128 129  13 130
## 10  67 248 248  24  10 248  17 248 248   3 248 248  10  67  17 248 248 248
## 25  40  68  11  32  18 249  32  11 249  68 249 249  18  NA  18 249 249  18
##    131 132 133 134 135 136 137 138 139 14 140 141 142 143 144 145 146 147
## 10   3 248 248  10  10   3 248 248   3 24 248 248 248 248 248 248 248  24
## 25  18  40  54  18   4  40  61 249   4 61 249  68  11  NA   4 249 209   4
##    148 149 15 150 151 152 153 154 155 156 157 158 159  16 160  17  18  19
## 10 248 248  3 248  39 248 248 248  NA  53 248 248  53 248  NA 248 136  24
## 25 249  81 11 249 249 249 249 249 249 123 249 249  32  25 249   4 249 123
##     2  20  21 22 23  24  25  26 27  28  29  3 30 31  32 33 34  35 36 37
## 10 248 248 248 24 24 248  24 248 53  31  80 31 NA NA 248 17 24 248 39 17
## 25  47  54 249 61 40 249 249 249  4 249 249 11  4 NA  18  4  4 159  4 11
##    38  39   4  40  41  42 43 44  45 46 47 48 49   5 50  51 52 53  54 55 56
## 10  3  10  31 248 122 248 NA  3  17 NA NA NA NA 248  3 248  3 10 248  3  3
## 25 NA 249 249   4  NA 123 NA  4 249 11 18  4 11 249 40  18 11 11 249  4 18
##    57  58 59  6 60  61 62 63 64  65  66 67 68 69  7 70 71 72  73  74 75
## 10 24   3 NA 10  3   3  3  3  3 248 248 NA 10 17 10  3 NA 10 248 248 NA
## 25 11 159 25 11 11 209  4  4 11 249  11 40 11 61 25  4  4  4 249 159 32
##     76 77 78 79   8 80  81 82 83  84  85 86 87 88 89   9 90 91 92 93 94 95
## 10 248  3 10 NA  NA  3 248 80 NA 241 122 24 10 10 24 115 17 NA 17 10  3  3
## 25 249 47 18  4 249 11  NA 11 11  11 249 11 NA 11 11 123 11 11 25 11  4 11
##    96 97 98 99
## 10 10 10 10 17
## 25 18 18 11  4
```

17.6. **sapply with expected result: `vapply`.** When we could use `sapply` and we know exactly what the size and class of the value of the function will be, it is sometimes faster to use `vapply`. The syntax is like that of `sapply`: `vapply(X,FUN,FUN.VALUE,...)`, where `X` and `FUN` are as in `sapply`, but we specify the size and class of the value of `FUN` via the `FUN.VALUE` argument. For example, suppose we define a function that, given a number between 1 and 26 will return the corresponding letter of the alphabet:

```
alph <- function (x) {
  if (x < 1 || x > 26) stop("bad value of x")
  LETTERS[as.integer(x)]
}
```

This function will return a vector of length 1 and class `character`. To apply it to a randomly sampled set of integers, we might do

```
  x <- sample(1:26,50,replace=TRUE)
  y <- vapply(x,alph,character(1))
  paste(y,collapse="")

## [1] "HZYNIJFRXXOBJYDTNUBNBAWHDLHAOOSNBKGHVZTDSFVXVBFMRV"
```

18. Vectorized functions vs. loops

As Ligges and Fox (2008) point out, the idea that one should avoid loops wherever possible in R, using instead vectorized functions like those in the `apply` family, is quite widespread in some quarters. The belief, which probably dates back to infelicities in early versions of S and Splus but is remarkably persistent, is that loops are very slow in R. Let's have a critical look at this.

Consider the following loop code that can be vectorized:

```
x <- runif(n=1e6,min=0,max=2*pi)
y <- numeric(length(x))
for (k in seq_along(x)) {
  y[k] <- sin(x[k])
}
```

To time this, we can wrap the vectorizable parts in a call to `system.time`:

```
x <- runif(n=1e6,min=0,max=2*pi)
system.time({
  y <- numeric(length(x))
  for (k in seq_along(x)) {
    y[k] <- sin(x[k])
  }
})
```

```
##    user  system elapsed
##   1.180   0.000   1.182
```

We can compare this with a simple call to `sin` (which is vectorized):

```
system.time(z <- sin(x))

##    user  system elapsed
##   0.032   0.000   0.035
```

Clearly, calling `sin` directly is much faster. What about using `sapply`?

Exercise 41. Compare the time spent on the equivalent calculation, using `sapply`. What does this result tell you about where the computational cost is incurred?

```
##    user  system elapsed
##   1.708   0.016   1.731
```

The above example is very simple in that there is a builtin function (`sin` in this case) which is capable of the fully vectorized computation. In such a case, it is clearly preferable to use it. Frequently, however, no such builtin function exists, i.e., we have a custom function of our own we want to apply to a set of data. Let's compare the relative speeds of loops and `sapply` in this case.

```
x <- seq.int(from=20,to=1e6,by=10)
f <- function (x) {
  (((x+1)*x+1)*x+1)*x+1
}
system.time({
  res1 <- numeric(length(x))
  for (k in seq_along(x)) {
    res1[k] <- f(x[k])
  }
})

##    user  system elapsed
##   0.228   0.020   0.251

system.time(res2 <- sapply(x,f))

##    user  system elapsed
##   0.176   0.000   0.175
```

[Actually, in this case, `f` is vectorized automatically. Why is this?]

```
system.time(f(x))
```

```
##    user  system elapsed
##       0       0       0
```

Another example: in this case function g is not vectorized.

```
g <- function (x) {
  if ((x[1] > 30) && (x[1] < 5000)) {
    1
  } else {
    0
  }
}

system.time({
  res1 <- numeric(length(x))
  for (k in seq_along(x)) {
    res1[k] <- g(x[k])
  }
})

##    user  system elapsed
##   0.228   0.000   0.226

system.time(res2 <- sapply(x,g))

##    user  system elapsed
##   0.148   0.000   0.150
```

Acknowledgments

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). It also draws on the documentation supplied with R. Versions of this document have been used in courses taught by BB and AAK at Florida, Michigan, and in the Ecology and Evolution of Infectious Diseases Workshop over several years.

References

Burns, P. (2012) *The R Inferno* (lulu.com), 2nd ed. URL http://www.burns-stat.com/pages/Tutor/R_inferno.pdf.

Fussman, G. F., Ellner, S. P., Sherzer, K. W., Nelson G. Hairston, J. (2000) Crossing the Hopf bifurcation in a live predator-prey system. *Science* **290**:1358–60.

Ihaka, R., Gentleman, R. (1996) R: a language for data analysis and graphics. *Journal of Computational and Graphical Statistics* **5**:299–314.

Ligges, U., Fox, J. (2008) R help desk : How can i avoid this loop or make it faster? *R News* **8**:46–50. URL http://www.r-project.org/doc/Rnews/Rnews_2008-1.pdf.

R Core Team (2014a) *An Introduction to R*. URL http://cran.r-project.org/doc/manuals/r-release/R-intro.html.

R Core Team (2014b) *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL http://www.R-project.org/.

Stevens, S. S. (1946) On the theory of scales of measurement. *Science* **103**:677–680. doi:10.1126/science.103.2684.677.

Venables, W. N., Ripley, B. D. (2002) *Modern Applied Statistics with S. Fourth Edition* (Springer, New York). URL http://www.stats.ox.ac.uk/pub/MASS4/. ISBN 0-387-95457-0.

Stephen P. Ellner, Department of Ecology and Evolutionary Biology, Cornell University

Benjamin Bolker, Departments of Mathematics & Statistics and Biology, McMaster University

Aaron A. King, Departments of Ecology & Evolutionary Biology and Mathematics, University of Michigan