

An introduction to R for ecological modeling (lab 1)

Stephen Ellner ^{*}; modified by Ben Bolker [†]

August 30, 2005

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 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.

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

^{*}Ecology and Evolutionary Biology, Cornell

[†]Zoology, University of Florida

- 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 [2], 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 R also includes extensive documentation, including an introductory manual (≈ 100 pages) and a comprehensive reference manual (over 1000 pages). (There is a semi-inclusive graphical front-end for R, called Rcmdr, available at the R site, but we will not be using it in this class.)

1.1 Installing R on your computer

The main source for R is the CRAN home page <http://cran.r-project.org>. You can get the source code, but most users will prefer a precompiled version. To get one of these from CRAN, click on the link for your OS, continue to the folder corresponding to your OS version, and from there to the download file (e.g. `base/rwxxxx.exe` for Windows, `rmxxx.sit` for under MacOS, where `xxxx` is the version number).

For Windows, you can also use one of the CDs I've burned for the class: you'll find the `rw2011.exe` file on the CD. Note that this corresponds (slightly confusingly) to R version 2.1.1 (not 20.11 or 2.011). You will also find pre-compiled versions of a large number of packages from CRAN on the CD, as well as a variety of other tools and documents.

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 the Windows version additional packages can be installed easily from within R using the **Packages** menu. Only some of the packages are available pre-compiled for Unix/Linux and MacOS X. For other packages in Unix/Linux you have to download and compile the source code.

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. Installing versions for Linux or Unix is more complicated and idiosyncratic, which will not bother the corresponding users. (This introduction is generally moderately Windows-specific, although we've tried to mark Windows-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/rwxxxx/etc`, where `rwxxxx` stands for the version of R. You will then need to restart R.
- To select the most powerful version of the help system, go to the same directory (`My Computer/Program Files/R/rwxxxx/etc`) and use Notepad to edit the `Rprofile` file to un-comment `options(chmhelp=TRUE)` by removing the `#` at the start of the line.

1.2 Starting R

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

1.3 Stopping R

Lebanese proverb: “when entering, always look for the exit”. 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); for now (and in general), say “no” in order to avoid clutter.

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

2 Interactive calculations

Ⓜ 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
```

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

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 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 alphanumeric characters. 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. 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
> 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 `↓`, `→`, `←`, **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, or powers, (3) multiplication and division, (4) addition and subtraction (“**pretty please my dear Aunt Sally**”).

```
> 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). You can get help on any R function by entering

```
?functionname
```

in the console window (e.g., try `?sin`). You should also explore the items available on the Help menu, which include the manuals, 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.

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, replace them with regular parentheses ().)
2.
 - $1 + 0.2$
 - $1 + 0.2 + 0.2^2/2$
 - $1 + 0.2 + 0.2^2/2 + 0.2^3/6$

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

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

- $e^{0.2}$ (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(c(1,2))` should give you the values for the standard normal for $x = 1$ and $x = 2$.)

Exercise 2.2: 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). If you have a net connection, try `RSiteSearch("sin")` from the command line or the equivalent from the menu and see what happens.

3 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 Fussmann et al. [1].

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)
```

(don't try to enter the +, which is a continuation character as described above). 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 is affected by light intensity, type

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

to plot `rmax` (y) against `Light` (x). A linear regression seems reasonable. **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()` (linear **m**odel) function:

```
> fit = lm(rmax ~ Light)
```

(Note that the variables are in the *opposite order* from the `plot()` command, which is `plot(x,y)`, whereas the linear model is read as “model r_{\max} as a function of light”.)

The `lm` command produces no output whatsoever, but it has created `fit` as an **object**, i.e. a data structure consisting of multiple parts, holding the results of a regression analysis with `rmax` being modeled as a function of `Light`. Unlike most statistics packages, R rarely produces automatic summary output from an analysis. 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 $r_{\max} = 1.58 + 0.0136 \times \text{Light}$, and 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-squared 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)
```

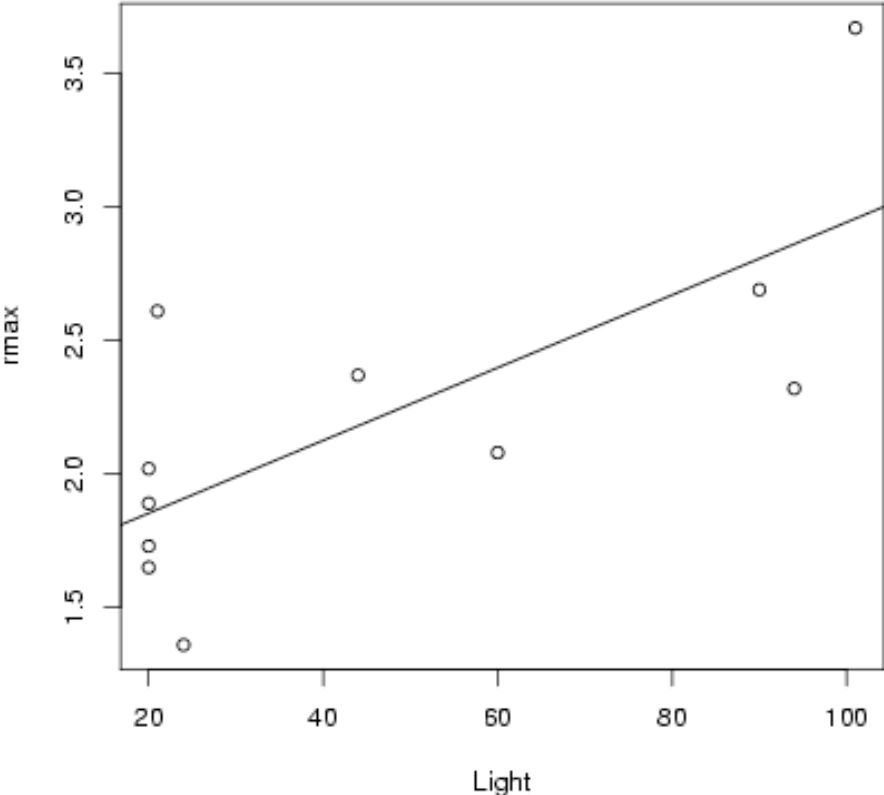


Figure 1: Graphical summary of regression analysis

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

You can 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"
```

For more information (perhaps more than you want) about `fit`, use `str(fit)` (for structure). You can get the regression coefficients this way:

```
> fit$coefficients

(Intercept)      Light
1.58095214  0.01361776
```

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`.

4 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, Xemacs, or Tinn-R: you **shouldn’t** use MS Word — 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 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 build 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 let R know where data and script files are located, you have three choices:
 1. spell out the *path*, or file location, explicitly.  There are two different ways to specify paths: a single forward slash (e.g. "c:/My Documents/R/script.R") or a double backslash (e.g. "c:\\My Documents\\R\\forest.txt"). R understands either of these, although you might as well just use the single forward slash, which works on all platforms.
 2. change your working directory to wherever the file(s) are located using **Change dir** in the **File** menu;
 3. change your working directory to wherever the file(s) are located using the `setwd()` (**set working directory**) function, e.g. `setwd("c:/temp")`

Changing your working directory may be 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.

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/R work**.

- it's important that data and script files be preserved 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 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 `Intro1.R` has the commands from the interactive regression analysis. **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*. 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 `Intro1.R`. In your editor, select and Copy the entire text of the file, and then Paste the text into the R console window (`Ctrl-C` and `Ctrl-V` as shortcuts). 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("c:/temp/Intro1.R")
```

`source()`ing can also be done by pointing and clicking via the **File** menu on the console window.

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

```
> X = read.table("ChlorellaGrowth.txt")
```

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.

Extract the variables from `X` with the commands

```
> Light = X[, 1]
> rmax = X[, 2]
```

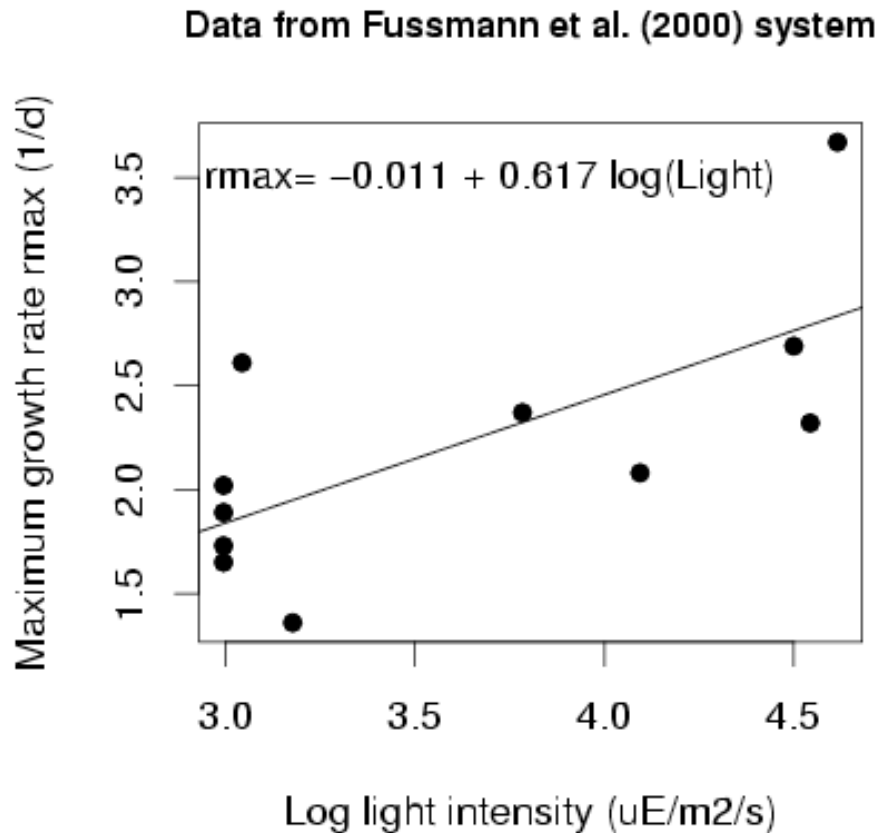


Figure 2: Graphical summary of regression analysis using log of light intensity (and annotating the plot)

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 4.1 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 Figure 2.

Exercise 4.2 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.

Exercise 4.3 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.

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 4.4 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))
```

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. `mfcow=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 `mfcow=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 `mfcow=c(1,2)`.

Exercise 4.5 * Use `?par` to read about other plot control parameters that can be set using `par()` (feel free to just 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 4.6 * Modify one of your scripts so that at the very end it saves the plot to disk. In Windows you can do this with `savePlot()`, otherwise with `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 data points (r_{\max} versus *Light* 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 once you get used to it and get in the habit of using it often.)

Some more tips on the help system:

- `help.start()` fires up a web browser pointing at all of the help files;
- `help()` or `?` only search through functions in the *currently loaded* packages (we'll get there); `help.search` looks through all of the *installed* packages;
- as mentioned above, `apropos()` just looks through all accessible R objects, which means it will match names of functions containing a given string
- `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)` gives information on all the objects in a particular package `pkg` (again, more about packages later)
- if you're connected to the Web, you can use the `RSiteSearch()` command (from the command line or the Help menu) to do a full-text search of all R documentation and the mailing list archives
- `example(function)` will run all of the examples in the help page, if any, for function `function`
- `demo(topic)` runs demonstration code on topic `topic`: type `demo()` by itself to list all available demos

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.

5 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* by Venables and Ripley [3] gives a good practical overview, and a list of available packages and their contents can be found at the main R website (<http://www.cran.r-project.org>, and click on Package sources). For the most part, we will not be concerned here with this side of R.

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

Table 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.

6 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

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

2. Using `read.table()`:

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

A vector can then be used in calculations as if it were a number (more or less)

```
> finalsize = initials + 1
> newsize = sqrt(initials)
> finalsize
```

```
[1] 2 4 6 8 10 12
```

```
> newsize
```

```
[1] 1.000000 1.732051 2.236068 2.645751 3.000000 3.316625
```

Notice that the operations were applied 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* manner; anything else (e.g. matrix multiplication) is done using special notation (discussed below).

6.1 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 6.1 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 .

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:

```
> 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 of the main functions for creating and working with vectors are listed in Table 3.

6.2 Vector indexing

Often it is 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
```

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

Table 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`.

(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.

```
> v = z[2:5]
> v
```

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

This has extracted the 2nd through 5th elements in the vector. If you enter `v=z[seq(1,5,2)]`, what will happen? Try it and see, and make sure you understand what happened.

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 \mathbf{z} while leaving all the rest alone, and

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

changes the 1st, 3rd, and 5th values.

Exercise 6.2 Write a *one-line* command to extract a vector consisting of the second, first, and third elements of \mathbf{z} *in that order*.

Exercise 6.3 Write a script file that computes values of $y = \frac{(x-1)}{(x+1)}$ for $x = 1, 2, \dots, 10$, and plots y versus x with the points plotted and connected by a line.

Exercise 6.4 The sum of the geometric series $1+r+r^2+r^3+\dots+r^n$ approaches the limit $1/(1-r)$ for $r < 1$ as $n \rightarrow \infty$. Take $r = 0.5$ and $n = 10$, and write a **one-statement** command that creates the vector $G = (r^0, r^1, r^2, \dots, r^n)$. Compare the sum (using `sum()`) of this vector to the limiting value $1/(1-r)$. Repeat for $n = 50$.

6.3 Logical operators

These operators return a logical value of TRUE or FALSE. For example, try:

```
> a = 1
> b = 3
> c = a < b
> d = (a > b)
> c
```

```
[1] TRUE
```

```
> d
```

```
[1] FALSE
```

The parentheses around `(a>b)` are optional but can be used to improve readability. One special case where you *do* need parentheses (or spaces) is comparing to negative values; `a<-1` will surprise you by assigning the value 1 to `a`, because `<-` (representing a left-pointing arrow) is an alternative way of assigning a value 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,

<code>x < y</code>	less than
<code>x > y</code>	greater than
<code>x <= y</code>	less than or equal to
<code>x >= y</code>	greater than or equal to
<code>x == y</code>	equal to

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

```
> 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)` 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
```

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 3 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:

```

!      Negation
&, && AND
|, || OR

```

OR is *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. 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.

6.4 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 low-light values of r_{\max} in the *Chlorella* example:

```

> X = read.table("ChlorellaGrowth.txt")
> 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]
```

```
[1] 20 20 20 24
```

```
> rmax[Light < 50 & rmax <= 2]
```

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

There are a huge number of variations on this theme.

Exercise 6.5 `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 6.6 * 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 6.7* Specify two ways to take only the odd elements of a vector of arbitrary length.

7 Matrices

7.1 Creating matrices

Like vectors, matrices can be created by reading in values from a data file using `read.table`. Matrices of numbers can also be entered 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(c(1, 2, 3, 4, 5, 6), nrow = 2, ncol = 3)
```

takes the values 1 to 6 and reshapes them into a 2 by 3 matrix. Note that values in the data vector are put into the matrix *column-wise*, by default. You can change this by using the optional parameter `byrow`. For example

```
> 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 of all 1's.

Exercise 7.1 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

```

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

Exercise 7.2 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 1's on the diagonal and 0 everywhere else.

Finally, in Windows 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 changed to whatever you need.

7.2 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

```

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

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.

```

      [,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 7.3 Verify that `rbind(C,D)` works, `cbind(C,C)` works, but `cbind(C,D)` doesn’t. Why not?

7.3 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]
```

```
> 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

```

8 Other structures: Lists and data frames

8.1 Lists

While vectors and matrices may seem pretty 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 is a little different too, use `[[]]` (rather 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.

8.2 Data frames

Data frames are the solution to the problem that vectors and matrices (which might seem to be the most natural way to store data) can only contain a single type of data, but most data sets have several different kinds of variables for each observation. Data frames are a hybrid of lists and vectors; internally, they are a list of vectors that can be of different types but all have to 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] G. Fussmann, S. P. Ellner, K. W. Shertzer, , and Jr. N. G. Hairston. Crossing the Hopf bifurcation in a live predator-prey system. *Science*, 290:1358–1360, 2000.
- [2] Ross Ihaka and Robert Gentleman. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, 1996.
- [3] Venables and Ripley. *Modern Applied Statistics with S*. Springer, New York, 3d edition, ????