An introduction to R for dynamic modeling

Stephen Ellner [*], modifications by Ben Bolker [†]

August 28, 2003

0 How to use this document

- These notes contain many sample calculations. It is important to do these yourselves - **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 (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) 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 Rlacks. However for our purposes 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 include extensive documentation, including an introductory manual ($\sim$ 100 pages) and a comprehensive reference manual (over 1000 pages).

1.1 Installing R on your computer

The main source for R is the CRAN home page `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 `rw1071.exe` *file on the CD. Note that this corresponds (slightly confusingly) to* R *version 1.7.1 (not 10.71 or 1.071). You will also find pre-compiled versions of a large number of libraries from CRAN on the CD.*

The standard distributions of R include several *libraries*, user-contributed suites of add-on functions. These Notes use some libraries that are not part of the standard distribution. In the Windows version additional libraries can be installed easily from within R using the **Packages** menu. Only some of the libraries are available pre-compiled for Unix/Linux and MacOS X. For others libraries in Unix/Linux you have to download and compile the source code. For MacOS 10.2, a complete pre-packaged version is maintained and distributed by Jan de Leeuw of the Department of Statistics at UCLA, at `http://gifi.stat.ucla.edu/pub`. The document `http://gifi.stat.ucla.edu/pub/R.pdf` gives installation instructions. The version with all the libraries required in this class is the "exit Gifi" option in those instructions. You'll need a good net hookup: the files are very large.

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 Ⓦ.)

Ⓦ We suggest that you edit the file `Rconsole` and change the line `MDI=yes` to `MDI=no`, and edit `Rprofile` to un-comment `options(chmhelp=TRUE)` by removing the # at the start of the line. These changes allow the command and graphics windows to move independently on the desktop, and selects the most powerful version of the help system. Some GUI options can be set from program menus when R is running (`Edit/GUI Preferences` in the Windows version).

Ⓦ It may also be a good idea to move R's default directory ...

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\rw1071\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, it 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 you try to run a command in R that seems to be stuck or taking longer than you're willing to wait, click on the stop sign on the menu bar or hit the `Escape` key.

2 Interactive calculations

Ⓦ When R is launched it opens the **console** window This has a few basic menus at the top, whose names and content are OS-dependent; check them out on your own. The console window 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 variables. For example: see

```
> a = 2 + 2
```

The variable `a` is automatically created and the result (4) is stored in it, but nothing is printed. 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 *not* to print the results. If you want R to print the value, just type the variable name by itself

```
> a

[1] 4
```

(`print(a)` also works). 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 `"a"`; 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 allow two or more commands to be typed 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 thinks 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).

Variable names in R must begin with a letter, followed by alphanumeric characters. Long names can be broken up using a period, as in `very.long.variable.number.3`, but (Windows users beware!) you **cannot** use the underscore character (_) or blank space as a separator in variable names. R is case sensitive: `Abc` and `abc` are **not** the same variable. (A general note on programming style: it helps to use descriptive variable names, such as `N.per.ha` or `pop.density` rather than `x` and `y` [although there is an obvious tradeoff with the amount of typing you'll have to do]. Don't underestimate how confusing something

that seemed crystal-clear at the time will seem when you come back to it a few months later . . .)

Calculations are done with variables as if they were numbers. R 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 the variable values for `x, y` were not displayed, R "remembers" that values have been assigned to them. Type `> x; y` to display the values.

If you mis-enter a command, it can be edited instead of starting again from scratch. The ↑ key (or `Control-P`) recalls previous commands to the prompt. For example, you can bring back the third-from-last command and edit it to

```
> z3 = 2 * x^y
```

You can do several operations in one calculation, such as

```
> A = 3
> C = (A + 2 * sqrt(A))/(A + 5 * sqrt(A))
> C
```

```
[1] 0.5543706
```

The parentheses specify the order of operations. (The command

```
> C = A + 2 * sqrt(A)/A + 5 * sqrt(A)
```

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

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) Exponentiation, (2) multiplication and division, (3) 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 find yourself desperately confused it's better to think through the order of operations rather than to just add parentheses at random.

R also has many **built-in mathematical functions** that operate on variables (see Table 1). 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) that is useful if you sort of maybe remember part of the the name of what it is you need help on.

Exercise 2.1 : Have R compute the values of

1. $\frac{2^7}{2^7-1}$and compare it with $\left(1-\frac{1}{2^7}\right)^{-1}$
2. $\sin(\pi/9), \cos^2(\pi/7) = (\cos(\pi/7))^2$ [Note that typing `cos^2(pi/7)` won't work!]
3. $\frac{2^7}{2^7-1} + 4\sin(\pi/9)$, using cut-and-paste to assemble parts of your past commands.

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).

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 *rmax* 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. (2000).

Light: 20, 20, 20, 20, 21, 24, 44, 60, 90, 94, 101
rmax: 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 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, `sd(rmax)` and `var(rmax)` return 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 `Light` (y) against `rmax` (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()` (**l**inear **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 *rmax* as a function of light").)

This 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)` . Model objects are set up in R (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 $rmax = 1.580952 + 0.013618 \times Light$, 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 t. Below the table, the adjusted R-squared gives the estimated fraction of

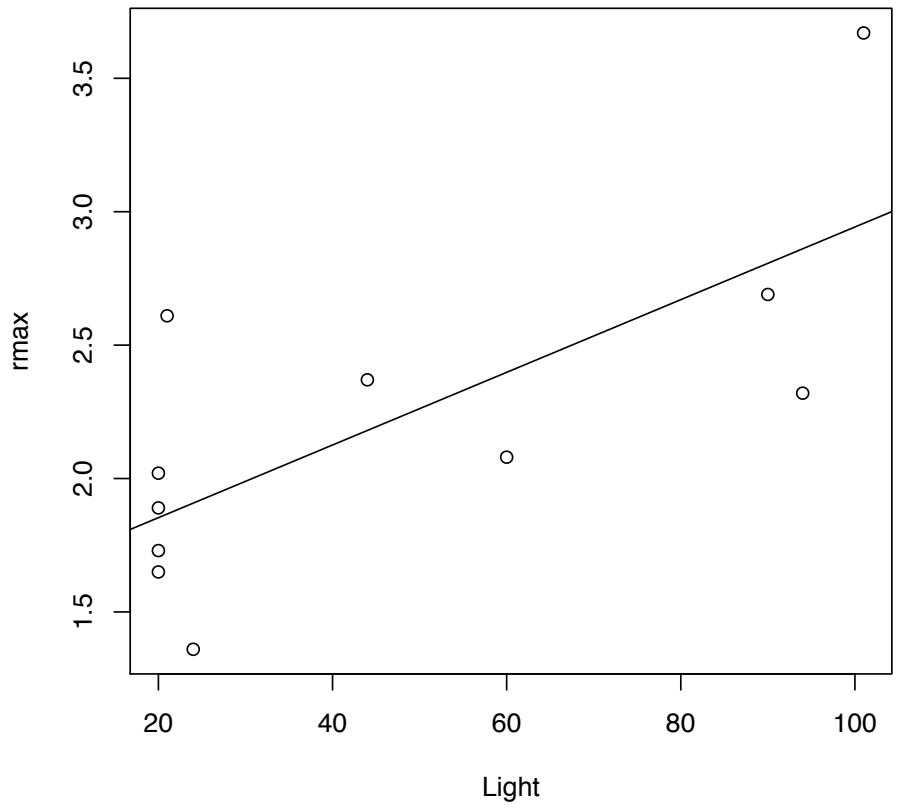

Figure 1: Graphical summary of regression analysis

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].

Adding the regression line to the plot of the data is similarly accomplished by 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: try `?abline`).

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

```
> coef(fit)

(Intercept)       Light
 1.58095214  0.01361776
```

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. You can also get the regression coefficients this way:

```
> names(fit)

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

> fit$coefficients

(Intercept)       Light
 1.58095214  0.01361776
```

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. As of this writing, only the MacOS version has a built-in script editor; with others you need to use an external text editing program (e.g. Windows Notepad, Wordpad, PFE, or Xemacs: you probably 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 libraries, or run another script file that creates some functions (more on functions later).
2. Read in from a text file the parameter values for a predator-prey model, and the numbers of predators and prey at time $t = 0$.
3. Calculate the population sizes at times $t = 1, 2, 3, \ldots, T$.
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 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 it may be better to get used to the single forward slash, which works across all platforms — but *a single backslash won't work*.
  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 w**orking **d**irectory) 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.

- 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), don't save the file as an Excel file; (3) **if at all possible don't use Microsoft Word to edit your data and script files — use Wordpad, or some other editor, instead**; it will try very hard to get you to save them as Word (rather than text) files, which will completely 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. Watch out for this.

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** 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 in point-and-click fashion 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("c:\\temp\\ChlorellaGrowth.txt")
```

reads the file 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 ...

The variables are then extracted 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 matrices later). From there out it's the same as before, with some additions that put 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`

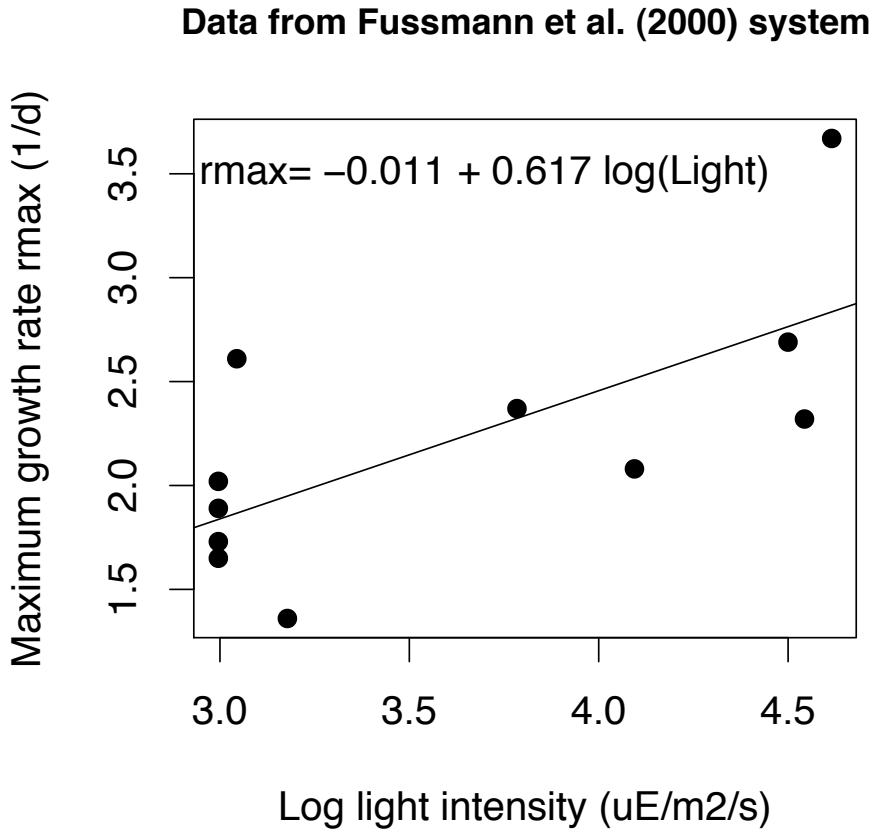

Figure 2: Graphical summary of regression analysis using log of light intensity

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 are 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 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".
```

(Remember, how you specify paths is OS-dependent ...)

[**Note:** 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 (*rmax* 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 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* libraries (we'll get there); `help.search` looks through all of the *installed* libraries; `help.search` uses "fuzzy matching" — for example, `help.search("log")` finds 528 entries 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" . . .)
- `library(help=lib)` gives information on all the objects in a particular library `lib` (again, more about libraries later)

Finally, here's a (funny, but perhaps discouraging) commentary on the help system in R from Graham Lawrence:

> I hate to say this, but what really helped me the most, after the initial feet-wetting, was to abandon the help manuals. Searching manuals for the answer to a specific question is frustrating because one does not know the key term for the search engine to deliver the item needed.
>
> So I stopped being serious and production oriented and simply played with R for a couple of months. Open the Base package, scan the list of contents for titles that pique my curiosity, paste their examples into R and see what happens. And those examples use other functions and their documentation has more examples and so on and on; and pretty soon after, whoops, there's another afternoon gone down the tubes. But all this apparent time wasting had a most happy result. I now find, much more often than not, that I know what I need to look up to answer a specific question, and that does wonders for my disposition.
>
> I think of R as a vast jungle, criss-crossed with myriad game trails (the documentation to each function in the packages). And I can explore each trail and see what animal it leads to in its native habitat, by pasting the examples into R and examining the result. So when I see an interesting spoor or paw print, I take a stroll down that trail and see where it leads. Not the most efficient way of learning the language, no doubt, but a pleasant and interesting entertainment rather than a chore.

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 library **mgcv**)
nls	Fit nonlinear models by least-squares (in library **nls**)
lme,nlme	Linear and nonlinear mixed-effects models (repeated measures, block effects, spatial models: in library **nlme**)
boot	Library: bootstrapping functions
modreg	Library: nonparametric regression (more such in libraries **fields, KernSmooth, logspline, sm** and others)
multiv	Library: multivariate analysis
survival	Library: survival analysis
tree	Library: tree-based regression

Table 2: A few of the functions and libraries in **R** for statistical modeling and data analysis. There are **many** more, but you will have to learn about them somewhere else.

5 Statistics in R

Some of the important functions and libraries (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 gives a good practical overview, and a list of available libraries and their contents can be found at the main R website (`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**.

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.

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

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

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

2. Using `read.table()`:

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

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

```
> finalsize = initialsize + 1
> newsize = sqrt(initialsize)
> 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). Note: this is the **opposite** of **Matlab**, where matrix operations are the default and element-by-element requires special notation.

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)`

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. There are also two shortcuts for creating vectors with `by=1`:

```
> 1:8

[1] 1 2 3 4 5 6 7 8

> c(1:8)
```

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

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

```
[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
```

created a vector in which the value 3 was repeated 5 times. `rep(values,lengths)` can also be used with a vector of values and their associated lengths, for example

```
> 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. This can be a little bit mind-blowing as you get started, but you'll get used to it ...

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

6.2 Vector addressing

Often it is necessary to extract a specific entry or other part of a vector. This is done using subscripts, for example

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

[1] 5
```

`q[3]` extracts the third element in the vector `q`. You can also access a block of elements using the functions for vector construction, e.g.

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

[1] 3 5 7 9
```

This has extracted 2^{nd} through 5^{th} elements in the vector. If you enter `v=q[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 = q[c(1, 2, 5)]
> v

[1] 1 3 9
```

Addressing is also used to **set specific values within a vector**. For example,

```
> q[1] = 12
```

changes the value of the first entry in `q` while leaving all the rest alone, and

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

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

Exercise 6.2 write a **one-line** command to extract a vector consisting of the second, first, and third elements of `q` **in that order**.

You may be wondering if vectors in R are row vectors or column vectors (if you don't know what those are, don't worry). The answer is "both and neither". Vectors are printed

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

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

out as row vectors, but if you use a vector in an operation that succeeds or fails depending on the vector's orientation, R will assume that you want the operation to succeed and will proceed as if the vector has the necessary orientation. For example, R will let you add a vector of length 5 to a 5×1 matrix or to a 1×5 matrix, in either case yielding a matrix of the same dimensions.

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

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

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

```
> x = 1:5
> b = (x <= 3)
> 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 running multiple simulations and seeing how often one outcome occurred rather than another.

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 x is true, if y is true, or if both x and y 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 addressing 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 *rmax* 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
```

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 in the matrix (e.g. if your vector were `(0.1 0.9. 0.7 0.3)` the answer would be `(1 4)`).

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), 2, 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, 3, 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 out a matrix of the specified size. So for example

```
matrix(1,50,50)
```

creates a 50×50 matrix of all 1's.

Exercise 7.1 Use a command of the form `X=matrix(v,2,4)` where v is a data vector, to create the following matrix X

`matrix(v,m,n)`	$m \times n$ matrix using the values in v
`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
`cbind(a,b,c,...)`	combine compatible objects by binding them along columns
`rbind(a,b,c,...)`	combine compatible objects by binding 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 $(i,j)^{th}$ element is `v[i]*w[j]`
`iden(n)`	$n \times n$ identity matrix (in `boot` library)
`zero(n,m)`	$n \times m$ matrix of zeros (in `boot` library)
`dim(X)`	dimensions of matrix X. `dim(X)[1]`=# rows, `dim(X)[2]`=# columns

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] [,4]
[1,]    1    1    1    1
[2,]    2    2    2    2
```

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
```

```
     [,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 addressing

Matrix addressing is like vector addressing except that you have to specify both the row and column, or range of rows and columns. For example `q=A[2,3]` sets q 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.

```
> first.row = A[1, ]
> first.row
```

```
[1] 1 2 3
```

```
> second.column = A[, 2]
> second.column
```

```
[1] 2 5 8
```

As with vectors, addressing works in reverse to assign 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
```

There is an easy shortcut to extract entire rows or columns: leave out the limits.

```
> first.row=A[1,]; first.row
[1] 1 2 3
> second.column=A[,2]; second.column;
[1] 2 5 8
```

As with vectors, addressing works in reverse to assign 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,]=runif(3); A
          [,1]       [,2]     [,3]
[1,] 0.1911789 0.07919515 0.798139
[2,] 4.0000000 5.00000000 6.000000
[3,] 7.0000000 8.00000000 9.000000
```

(see Table 4 to remind yourself about `runif`).

Exercise 7.4 Write a script file to do the following. (a) Use **runif** to construct a 5×5 matrix **B** of random numbers with a uniform distribution between 0 and 1. (b) Extract from it the second row, the second column, and the 3×3 matrix of the values that are not at the margins (i.e. not in the first or last row, or first or last column). (c) Use `seq` to replace the values in the first row of **B** by `2 5 8 11 14`.

8 Iteration ("Looping")

8.1 For-loops

Loops make it easy to do the same operation over and over again, for example:

- Solving a model to make population forecasts 1 year ahead, then 2 years ahead, then 3, etc.
- Simulating a model multiple times with different parameter values.

There are two kinds of loops in **R** : **for** loops, and **while** loops. A **for** loop runs for a specified number of steps:

```
for (variable in vector) {
      commands
}
```

Here's an example (in **Loop1.R**):

```
# initial population size
initsize=4;

# create vector to hold results and store initial size
popsize=rep(0,10); popsize[1]=initsize;

# calculate population size at times 2 through 10, write to Command Window
for (n in 2:10) {
      popsize[n]=2*popsize[n-1];
      x=log(popsize[n]);
      cat(n,x,"\n");
}
plot(1:10,popsize,type="l");
```

The first time through the loop, n=2. The second time through, n=3. When it reaches n=10, the for-loop is finished and **R** starts executing any commands that occur after the end of the loop. The result is a table of the log population size in generations 2 through 10.

Note also the `cat` function (short for "concatenate") that prints results to the console window. `cat` converts its arguments to character strings, concatenates them, and then prints them. The `"\n"` argument is a line-feed character (as in **C**) so that each (n,x) pair is put on a separate line.

If this discussion of looping doesn't make sense to you, **stop now and get help**. Loops are essential for modeling.

Exercise 8.1 Write a script file that uses two for-loops, one after the other, to create the following 5×5 matrix A.

0	1	2	3	4
0.1	0	0	0	0
0	0.2	0	0	0
0	0	0.3	0	0
0	0	0	0.4	0

Exercise 8.2 Suppose that while doing fieldwork in some distant land you and your assistant have picked up a parasite that grows exponentially until treated. Your case is more severe than your assistant's: on return to Ithaca there are 400 parasites in you, and only 120 in your assistant. However, your field-hardened immune system is more effective. In you the number of parasites grows by 10 percent each day, while in your assistant they increase by 20 percent each day. That is,

$$n(0) = 400, n(j+1) = 1.1n(j) \text{ (you)}$$
$$m(0) = 120, m(j+1) = 1.2m(j) \text{ (your assistant)}$$

Write a script file **Parasite1.R** that uses a for-loop to compute the number of parasites in your body and your assistant's over the next 30 days, and then draws a single plot of both on log-scale (i.e. $\log(n(j))$ and $\log(m(j))$ versus time for 30 days). [This could be done without a loop, but we'll soon be extending this script to do things that require loops]. Notes: (1) For efficiency, compute all values of n and m first, then take logs. (2) One way to plot two curves on one graph is with `matplot`. It works like `plot` but `matplot(x,A)` plots each column of a matrix `A` as a separate curve. In this case `A=cbind(log(n),log(m))` will create the matrix you need, assuming n and m are column vectors.

Exercise 8.3 Modify **Parasite1.R** so that the parasite growth rates are random, in particular so that on day j the parasite loads increase by factors

$$r_i(j) = \tilde{r}_i e^{0.1Z_i(j)}$$

(here r_1 is for you, and r_2 is for your assistant), where each $Z_i(j)$ is random with a Normal distribution, mean=0 and sd=1, and $\tilde{r}_1 = 0.1$ and $\tilde{r}_2 = 0.2$. The model equations are then

$$n(0) = 400, \quad n(j+1) = (1 + r_1(j))n(j) \quad \text{(you)}$$
$$m(0) = 120, \quad m(j+1) = (1 + r_2(j))m(j) \quad \text{(assistant)}$$

Have the script file plot n and m over time, on log scale. if you've done this right, the results will be different each time you run the script. Save this script file as **Parasite2.R** for use later.

Exercise 8.4 Write a script file that uses a for-loop to calculate solutions of the difference equation model

$$N(j+1) = r(j)N(j)/(1 + N(j)), \quad N(1) = 1$$

for $j = 2, 3, 4, \cdots, 13$, where $r(j) = 2e^{-0.2j}$. Write your script so that all values of $r(j)$ are computed and stored in a vector before the start of the for-loop.

Exercise 8.5 Modify your script file from the previous exercise so that it reads a list of 12 positive numbers from a text file named **r.txt** into an vector named **r**, and then uses a for-loop to calculate the values of $N(j)$. You should find **r.txt** in the course folder.

8.2 Nested loops

Several `for` loops can be nested within each other, as in the example below. It is important to notice that the second loop is **completely** within the first. Loops must be either **nested** (one completely inside the other) or **sequential** (one starts after the previous one ends).

```
A=matrix(0,3,3);                (1)
for (row in 1:3) {              (2)
      for (col in 1:3) {        (3)
          A[row,col]=row*col    (4)
      }                         (5)
}                               (6)
A;                              (7)
```

Type the example above into a script file (without the line numbers!) and run it; the result should be

```
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    2    4    6
[3,]    3    6    9
```

Line 1 creates **A** as a matrix of all zeros, which is then filled in with meaningful values as the program runs. Line 2 starts a loop over the rows of **A**, and immediately in line 3 a loop over the columns is started. To fill in the matrix we need to consider all possible values for the pair (row, col). So for row=1, we need to consider col=1,2,3. Then for row=2 we also need to consider col=1,2,3, and the same for row=3. That's what the nested for-loops accomplish. For row=1 (as requested in line 2), the loop in lines 3-5 is executed until it ends. Then we get to the end in line 6, at which point the loop in line 2 moves on to row=2, and so on.

Nested loops are useful for running a simulation many times, for example with different parameters or to look at the average over many runs of a stochastic model. For example (**Loop2.R**),

```
p=rep(0,5);                          (1)
for (init in c(1,5,9)){              (2)
      p[1]=init;                     (3)
      for (n in 2:5) {               (4)
         p[n]=2*p[n-1]               (5)
         cat(init,n,p[n],"\n");      (6)
      }                              (7)
}                                    (8)
```

Line 1 creates the vector p. Line 2 starts a loop over initial population sizes Lines 4-7 do a "population growth" simulation Line 8 then closes the loop over initial sizes

The result that the population growth iteration is done repeatedly, for a series of values of the initial population size. To make the output a bit nicer we can add some headings as the program runs - source **Loop3.R** and then look at the file to see how that was done.

Exercise 8.6 Modify your script file **Parasite2.R** so that it
(1) uses nested for-loops to compute 10 simulations of the parasite load in your body (drawing different values of the Z's for each time t in each solution), and stores the results in a matrix such that the j^{th} column of each matrix holding the results from the j^{th} simulation of the model.
(2) then uses another loop to compute, and store in vectors, the mean and standard deviation of the 10 simulations as a function of time
(3) produces a 2×1 plot (i.e. with `par(mfrow= ...)`) showing the mean and standard deviation as a function of time.
Save this script as **Parasite3.R**.

8.3 While-loops

A **while** loop lets an iteration continue until some condition is satisfied. For example, we can solve a model until some variable reaches a threshold. The format is

```
while(condition){
commands
}
```

The loop repeats as long as the condition remains true. **Loop4.R** contains an example: read it and then source it. Notice:

1. Although the condition in the while loop said `while(popnow<1000)` the last population value was > 1000. That's because the loop condition is checked **before** the commands in the loop are executed. In generation 6 the population size is 640, so the condition is satisfied and the loop runs through one more time. After that the population size is 1280, so the condition is not satisfied, and the program moves on to statements after the loop.

2. Since we don't know in advance how many iterations will occur, we can't create in advance a vector to hold the results. Instead, a vector of results (called `popsize`) is built up one step at a time. Before the loop starts, it is initialized to consist of the initial population size – (`popsize=initsize;`). Then at each step of the loop, the population "now" is added to the end of textttpopsize by the line `popsize=c(popsize,popnow)`.

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

Table 6: Some comparison operators in **R** . Use ?Comparison to learn more.

3. When the loop ends and we want to plot the results, the "y-values" are `popsize`, and the x values need to be `0:something`. To find "something", the `length` function is used to find the length of `popsize`.

Another way to accomplish the same thing is with a **counter**, a variable that is used to keep track of how many times the loop has executed. Look at **Loop5.R** to see an example. The key bit is:

```
# set initial size and initialize counter for length of popsize
popsize=10; j=1;

# calculate new population size, and update the counter
while(popsize[j]<1000) {
        popsize[j+1]=2*popsize[j];
        j=j+1;
}
```

Here `j` is used to keep track of the length of `popsize`: how many values have we computed so far? What happens inside the loop should be illegal, but **R** lets you get away with it. The first time through, `j`=1, so the first line of the loop is computing `popsize[2]`. **There is no such thing as popsize[2]!** But **R** assumes you want there to be such a thing, so it turns `popsize` into a vector of length 2. The same thing happens each time through the loop: as you compute a new population size and try to stick it into a nonexistent location in `popsize`, **R** expands `popsize` so that the location you need does exist. This is useful but dangerous, since in most other programming languages it can create unexpected errors or program crashes if you refer to a nonexistent part of a vector or matrix.

The conditions controlling a **while** loop are built up from operators that compare two variables (Table 6). These operators return a logical value of TRUE or FALSE. For example, try:

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

The parentheses around (a>b) are optional but can be used to improve readability in script files.

Exercise 8.7 Modify your script file **Parasite1.R** so that it uses a while-loop to compute the number of parasites in you and your assistant so long as you are sicker than your assistant (i.e. so long as $n > m$) and stops when your assistant is sicker than you.

8.4 More on comparison operators

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

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

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 running multiple simulations and seeing how often one outcome occurred rather than another.

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 x is true, if y is true, or if both x and y are true. For example, try
`>> a=c(1,2,3,4); b=c(1,1,5,5); (a<b)&(a>3); (a<b)|(a>3);`
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.

8.5 A simulation project

Project Exercise 8.8 Write a script file that simulates geometric population growth with spatial variation, defined as follows. The **state variables** for the model are the numbers of individuals in a series of $\mathbf{L} = 20$ patches along a line (L stands for "length of the habitat") .

1	2	3	4	...					...	L-1	L

Let $N_j(t)$ denote the number of individuals in patch j ($j = 1, 2, \ldots, L$) at time t ($t = 1, 2, 3, \ldots$), and let λ_j be the geometric growth rate in patch j. The **dynamic equations** for this model consists of two steps:

1. Geometric growth within patches:

$$M_j(t) = \lambda_j N_j(t) \quad \text{for all } j. \tag{1}$$

2. Dispersal between neighboring patches:

$$N_j(t+1) = (1-2d)M_j(t) + dM_{j-1}(t) + dM_{j+1}(t) \quad \text{for } 2 \le j \le L-1 \tag{2}$$

where $2d$ is the "dispersal rate". We need special rules for the end patches. For this exercise assume *reflecting boundaries*: ones who go out into the void have the sense to come back. That is, there is no leftward dispersal out of patch 1 and no rightward dispersal out of patch L:

$$\begin{aligned} N_1(t+1) &= (1-d)M_1(t) + dM_2(t) \\ N_L(t+1) &= (1-d)M_L(t) + dM_{L-1}(t) \end{aligned} \tag{3}$$

◇ Write your program to start with 5 individuals in each patch at time t=1, iterate the model up to t=50, and graph the total population size (total number in all patches) over time. Use the following growth rates: $\lambda_j = 0.9$ in the left half of the patches, and $\lambda_j = 1.2$ in the right.
◇ Write your program so that d and L are parameters, in the sense that the first line of your script file reads `d=0.1; L=20;` and the program would continue to run if these were changed other values.

Notes and hints:

1. This is a real programming problem. Think first, then start writing your code.

2. One thing to notice is that this model is not *totally* different from **Loop1.R**, in that you start with a founding population at time 1, and use a loop to compute successive populations at times 2,3,4, and so on. The difference is that the population is described by a vector rather than a number. Therefore, to store the population state over time you will need a matrix `njt` with L columns. Then `njt[t,]` is the population state vector at time t.

3. **Vectorize**. **R** does vector/matrix operations much faster than loops. Set up your calculations so that computing $M_j(t) = \lambda_j N_j(t)$ for $j = 1, 2, \cdots, L$ is a **one-line** statement of the form `a=b*c` . Then for the dispersal step, if $M_j(t), j = 1, 2, \ldots, L$ is stored as a vector `mjt` of length L, then what (for example) are $M_j(t)$ and $M_{j\pm1}(t)$ for $2 \le j \le (L-1)$?

Exercise 8.9 . Use the model (modified as necessary) to ask whether the spatial arrangement of good versus bad habitat patches makes a difference for population growth rate. For example, does it matter if all the good sites ($\lambda > 1$) are at one end or in the middle? What if they aren't all in one clump, but are spread out evenly (in some sense) across the entire habitat? **Be a theoretician**: (a) Patterns will be easiest to see if good sites and bad sites are very different from each other. (b) Patterns will be easiest to see if you come up with a nice way to compare growth rates across different spatial arrangements of patches. (c) Don't confound the experiment by also changing the proportion of good versus bad patches at the same time you're changing the spatial arrangement.

Exercise 8.10 . Modify your script file for the model (or write it this way to begin with...) so that the dispersal phase (equations 2 and 3) is done by calling a function **reflecting=function(Mt,d)** whose arguments are the pre-dispersal population vector $M(t)$ and the dispersal parameter d, and which returns $N(t+1)$, the population vector after dispersal has taken place.

9 Branching

Another use of comparison operators is to let the "rules" for state variable dynamics depend on the current values of state variables. The **if** statement lets us do this; the basic format is

```
if(condition) {
    some commands
}else{
    some other commands
}
```

An `if` block can be set up in other ways, but the layout above, with the }else{ line to separate the two sets of commands, can always be used.

If the "else" is to do nothing, you can leave it out:

```
if(condition) {
    commands
}
```

Exercise 9.1 Look at and source **a copy of Branch1.R** to see an `if` statement which makes the population growth rate depend on the current population size.

Exercise 9.2 Modify your **Parasite2.R** script so that the random variation in parasite success is "good/bad" rather than Gaussian. Specifically, on "bad days" the parasites increase by 10% while on "good days" they are beaten down by your immune system

and they go down by 10%, and similarly for your assistant. That is,

$$\text{Bad days: } n(j+1) = 1.1n(j), \quad m(j+1) = 1.2m(j)$$
$$\text{Good days: } n(j+1) = 0.9n(j), \quad m(j) = 0.8m(j)$$

Do this by using `runif(1)` and an `if` statement to "toss a coin" each day: if the random value produced by `runif` for that day is < 0.25 it's a good day, and otherwise it's bad.

9.1 Nested if statements

More complicated decisions can be built up by nesting one `if` block within another, i.e. the "other commands" under `else` can include an `if` block. **Branch2.R** uses **elseif** to have population growth tail off in several steps as the population size increases:

```
for (i in 1:50) {                    (1)
    if(popnow<250){                  (2)
        popnow=popnow*2;             (3)
    }else{                           (4)
        if(popnow<500){              (5)
            popnow=popnow*1.5        (6)
        }else{                       (7)
            popnow=popnow*0.95       (8)
        }                            (9)
    }                                (10)
    popsize=c(popsize,popnow);       (11)
}                                    (12)
```

What does this accomplish?

- If `popnow` is still < 250, then line 3 is executed growth by a factor of 2 occurs. Since the `if` condition was satisfied, the entire `else` block (line numbers 5-10 above) isn't looked at; **R** jumps line (11) and continues from there.
- If popnow is not < 250, **R** moves on to the `else` on line 4, and immediately encounters the `if` on line 5.
- If popnow is < 500 the growth factor of 1.5 applies, and **R** then jumps to the **end** and continues from there.
- If neither of the two `if` conditions is satisfied, the final `else` block is executed and population declines by 5% instead of growing.

Exercise 9.3 Use nested if's to write a script that draws a random number U between 0 and 1 using `runif(1)` and then writes to the console window "Small", "Medium", or "Large" depending on whether U is $\leq 1/3$, between $1/3$ and $2/3$, or $\geq 2/3$.

10 Writing your own functions

Functions (often called subroutines in other programming languages) allow you to break a program into subunits. Each function is an independent little program, performing a few related tasks and returning the results. This makes complex problems easier to program, and makes it easier to see the logical flow of a large program. In addition, each function can be written and tested independently, before you try to put all the pieces together.

Also, many **R** functions for simulation and data analysis require that you specify your model in the form of a function – for example, a system of differential equations that you want to solve numerically, or a nonlinear model that you want to fit to experimental data. The **R** function (for solving differential equations, doing nonlinear least squares, etc.) is then "told" the name of your function, and does its job on your particular model.

The basic syntax for creating a function is as follows. Suppose [for the sake of an example] you want a function `mysquare` that produces sums of squares: given vectors `v` and `w`, it returns a vector consisting of the element-by-element sums of the squares of the elements in the two vectors. The syntax is then:

```
mysquare=function(v,w) {
    u=v^2+w^2;
    return(u)
}
```

This code adds `mysquare` as an **R** command, just like `sin` or `log`. The variable `u` is *internal* to the function; if you use `mysquare` in a program, the program won't "know" the value of `u`. You can save some typing by computing the final value within the return statement:

```
mysquare=function(v,w) {return(v^2+w^2)}
```

Exercise 10.1 Type the above into a script file and run it, and then do `q=mysquare(1:4,1:4); q` in the console window.

Schematically, a function is defined by a code block like this:

```
function.name=function(argument1,argument2,...) {
     command;
     command;
       ...
     command;
     return(value)
}
```

Functions can return several different values, by combining them into a list with named parts.

```
mysquare2=function(v,w) {
    q=v^2; r=w^2
    return(list(v.squared=q,w.squared=r))
}
```

You can then extract the components in the usual way.

```
> x=mysquare2(1:4,2:5); names(x);
[1] "v.squared" "w.squared"
> x$v.squared
[1]  1  4  9 16
```

Functions can be placed **anywhere** in a script file. Once the code defining the function has been executed within a session, the new function can be treated like any other **R** command. However, **user-defined functions "vanish" when you end a session.** To use them again in another session, the function code needs to be run again. Alternatively, you can set things up (on your **R** at home) so that functions you use consistently are automatically loaded whenever **R** starts; the next subsection describes how.

Exercise 10.2 Write a function `stats(v)` that takes as input a single vector, and returns a list with named components `average` (mean of the values in the vector), and `variance` (population variance estimated from the values in the vector, using `var`). Verify that once you've sourced the function definition, you get

```
> stats(1:21);
$average
[1] 11
$variance
[1] 38.5
```

Exercise 10.3 Write a function to compute a forager's expected rate of energy gain $R(t_1, t_2)$ in a Patch Model with two patch types, travel time $T = 3$, 70% patches of type 1 with gain function $g_1(t) = 2t^{0.5}$, and 30% of type 2 with $g_2(t) = t^{0.7}$. Recall that the gain rate as a function of the GUTs t_i in a Patch Model with multiple patch types is $R = \frac{\sum_i P_i g_i(t_i)}{T + \sum_i P_i(t_i)}$. Save this script as **TwoPatchRate.R**, we'll be using it later.

Exercise 10.4 Write a script to simulate the random growth model

$$n(t+1) = \lambda(t)n(t), n(0) = 1$$

with $\lambda(t) = e^{Z(t)}$ and $Z(t)$ having a normal distribution with mean=0.05, sd=0.5. Have your script do this using a function `new.pop(nt)` which computes and returns $\lambda(t)nt$ (here `nt` is a number, representing a value of $n(t)$).

Exercise 10.5 (a) Modify your script from the last exercise so that the input to `new.pop` can be a vector rather than a single number, say $nt = (n_1, n_2, n_3, \cdots, n_d)$, and the output is the vector with components $n_i\lambda_i(t)$ with each λ_i having the distribution for λ specified in the last exercise. Do this in such a way that the function does not use any loops. If `new.pop` needs to know how long a vector nt is, remember the `length` function.
(b) Having done this, you can now modify your script so that it does many replicate simulations of the random growth model, i.e.

$$n_i(t+1) = \lambda_i(t)n(t), n_i(0) = 1, i = 1, 2, \cdots, d,$$

using a single loop on time t, rather than a nested loop over time t and population number i (this results in much faster execution compared to a nested loop). [How: set up n as a matrix with d columns, and 1's in the first row. One call to `new.pop` with the first row as input, returns values for the second row: `n[2,]=new.pop(n[1,])`. And so on, as often as needed). Use `matplot` to display the results for $t = 0, 1, \cdots 50$ and $d = 20$ replicate populations.

10.1 Autoloading functions at startup

If you've written some functions that you would like to have available every time you use **R** , there's a mechanism for having them load automatically at startup. It takes a bit of work, but not much.

First, take all the functions you want auto-loaded, and copy them into a single file, for example, `c:/src/R/MyFunctions.R`. Then look in the `etc` subfolder of your **R** folder, and find the file **Rprofile**. Commands in **Rprofile** are executed whenever **R** is started. You can edit this file, and at the end add a line like

```
source("c:/src/R/MyFunctions.R");
```

Then any functions in `MyFunctions.R` will be run at the start of each session.