An introduction to **R** for dynamic modeling

Stephen Ellner
Ecology and Evolutionary Biology, Cornell
March 19, 2003

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, 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
- 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 Gentlemen 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 **R** lacks. 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).

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.

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).

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, type `a=2+2; a` at the prompt and you see

```
> a=2+2; a
[1] 4
```

The variable `a` has been created, and assigned the value 4. The semicolon allows two or more commands to be typed on a single line; the second of these (`a` by itself) tells **R** to print out the value of `a`. By default, a variable created this way is a vector (an ordered list of numbers); in this case `a` is a vector length 1, which acts just like a number.

Variable names in **R** must begin with a letter, and 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.

Calculations are done with variables as if they were numbers. **R** uses `+, -, *, /, and ^` for addition, subtraction, multiplication, division and exponentiation, respectively. For example enter

```
> x=5; y=2; z1=x*y; z2=x/y; z3=x^y; z2; z3
```

and you should see

```
[1] 2.5
[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 recalls previous commands to the prompt. For example, you can bring back the next-to-last command and edit it to

```
> x=5 y=2 z1=x*y z2=x/y z3=x^y z2 z3
```

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.

so that commands are not separated by a semicolon. Then press Enter, and you will get an error message.

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 are specifying the order of operations. The command

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

gets a different result – the same as

```
> 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)` .

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. $\sin(\pi/9), \cos^2(\pi/7)$ [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);
```

The function `c()` *combines* the individual numbers into a vector.

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.

To see how the algal rate of increase is affected by light intensity,

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

creates a plot. 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()` function:

```
> fit = lm(rmax~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:

```
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 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 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 [Note: if you closed the plot of the data, you will need to create it again in order to add the regression line]

```
> abline(fit)
```

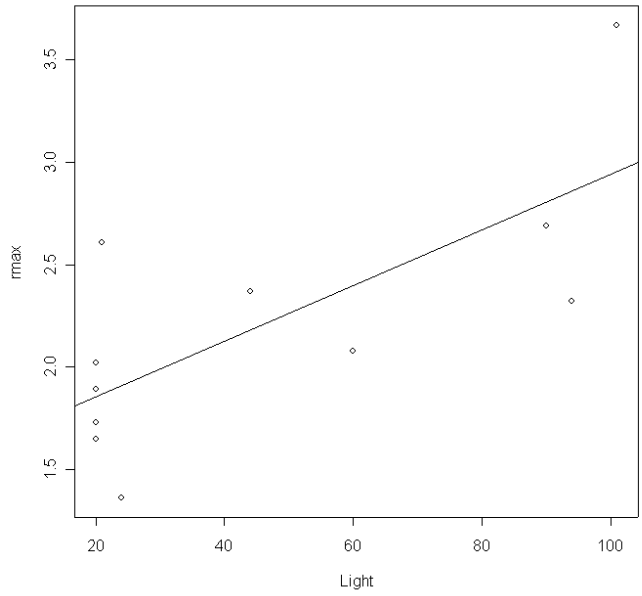

Figure 1: Graphical summary of regression analysis

You can also "interrogate" fit directly. Type `> names(fit)` to get a list of the components of `fit`.

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

Components of an object are extracted using the "$" symbol. For example, `> fit$coeffficients` yields the regression 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 or PFE).

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.

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. 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 course folder 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")
```

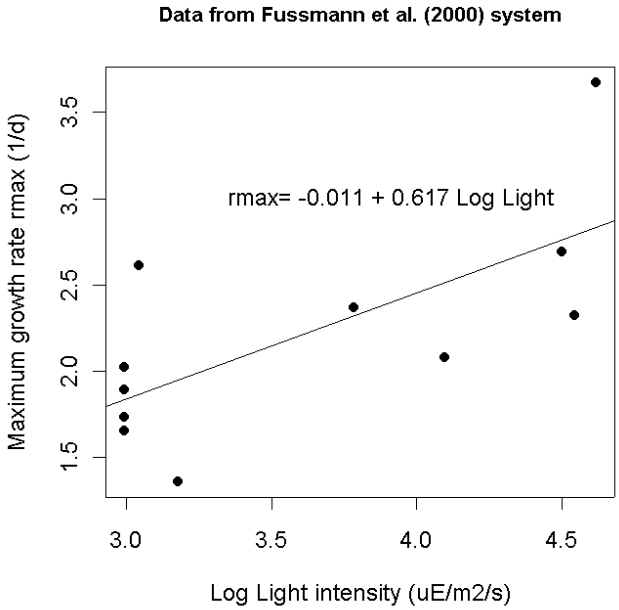

Figure 2: Graphical summary of regression analysis using log of light intensity

reads the file and puts the data values into the variable `X`. **NOTE** that in this line the path to the file has to be specified; you will need to edit the file so that it uses correct path to your temp folder. Also note that there are two different ways to specify paths: a single forward slash (as in the `source`) command just above, or a double backslash as in the `read.table` command. **R** understands either of these, but *a single backslash won't work.*

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 sort of like 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 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()`. 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`

aov, anova	Analysis of variance or deviance
glm	Generalized linear models
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 (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.

`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].

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 predefined 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; newsize;
[1]  2  4  6  8 10 12
[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; c(1:8);
[1] 1 2 3 4 5 6 7 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
```

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

`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`.

```
> 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.

Some of the main functions for creating and working with vectors are listed in Table 3. **R** also has numerous functions for creating vectors of random numbers with various distributions, that are useful in simulating stochastic models. Most of these have a number of **optional arguments**, which means in practice that you can choose to specify their value, or if you don't a default value is assumed. For example,

```
> rnorm(100)
```

yields 100 random numbers with a Normal (Gaussian) distribution with mean=0, standard deviation=1. But

```
> rnorm(100,2,5)
```

yields 100 random numbers from a Gaussian distribution with mean=2, standard deviation=5. The existence of default values for some arguments of a function is indicated by writing (for example) **rnorm(n,mu=0,sd=1)**. Since no default value is given for n, the user must supply one: `rnorm()` gives an error message.

Exercise 6.2 Generate a vector of 5000 random numbers from a Gaussian distribution with mean=3, standard deviation=2. Use `hist` to visualize the distribution of values, and the functions `mean, sd` to estimate the population mean and standard deviation from the "data" values in the vector.

Some of the functions for creating length-n vectors of random numbers are listed in Table 4. Functions to evaluate the corresponding probability distribution functions are also available. For a listing use the Help system (?Normal, ?Uniform, ?Lognormal, etc. for lists of the available functions for each distribution family). We will see soon some examples of using these functions to simulate models with random components to their

`rnorm(n,mean=1,sd=1)`	Gaussian distribution(mean=mu, standard deviation=sd)
`runif(n,min=0,max=1)`	Uniform distribution on the interval (min,max)
`rbinom(n,size,prob)`	Binomial distribution with parameters #trials N=size, probability of success p=prob.
`rpois(n,lambda)`	Poisson distribution with mean=lambda
`rbeta(n,shape1,shape2)`	Beta distribution on the interval $[0, 1]$ with shape parameters shape1,shape2

Table 4: Some of the main **R** function for generating vectors of random numbers. To create random matrices, these can be reshaped using `matrix()`.

dynamics.

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 6.3 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: we'll get to it later). The answer is "both and neither". Vectors are printed out as row vectors, but if you use a vector in an operation that succeeds or fails depending on the vector's orientation, **R** will assume that you want the operation to succeed and will proceed as if the vector has the necessary orientation. For example, **R** will let you add a vector of length 5 to a 5×1 matrix or to a 1×5 matrix, in either case yielding a matrix of the same dimensions.

Exercise 6.4 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 6.5 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 $n = 50$.

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.

```
> X
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

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=T); 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

```
     [,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
```

`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.

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,]=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.

11 Solving differential equations

In addition to basic programming, **R** includes many pre-written functions that make it relatively simple (compared to languages such as **C** or fortran) to do some complicated things. A prime example is finding numerical solutions of systems of differential equations

$$dx/dt = f(t, x) \tag{4}$$

Here t is time, and x is the state vector of the system, i.e.

$$x(t) = (x_1(t), x_2(t), \cdots, x_n(t))$$

where the $x_i(t)$ are the state variables of the model.

The first step is to write an **R** function to evaluate f in (4). For example, here is an example of the Lotka-Volterra competition equations:

$$\begin{aligned} dx_1/dt &= x_1(2 - x_1 - ax_2) \\ dx_2/dt &= x_2(3 - x_2 - bx_1) \end{aligned} \tag{5}$$

and here is the function that you would write for it:

```
lotka=function(t,x,parms) {
    a=parms[1]; b=parms[2];
    xdot=rep(0,2);
    xdot[1]=x[1]*(2-x[1]-a*x[2]);
    xdot[2]=x[2]*(3-x[2]-b*x[1]);
    return(list(xdot));
}
```

There's a lot to digest in that. Here are the things to note:

1. The function must have input arguments (t,x,parms), even if only x is used to compute the derivative. When you pass in an argument that isn't used, **R** just ignores it.

2. **parms** is a vector of parameter values for function. This allows you to write a function with "free" parameters, so that you can see how solutions change as parameters are varied, without having to re-write the function for each new parameter set. It also simplifies the process of fitting differential equations to data (we'll be doing this soon).

3. The derivative has to be returned as a list. There's a good reason for this; if you're curious and have *way* too much free time on your hands, ?lsoda contains the explanation. You've been warned.

The second step is to use an **R** function that implements an algorithm for numerical solution of differential equations. These are in the **odesolve** library, which has to be loaded (at the command line or at the top of a script file) by

```
library(odesolve).
```

The "basic" solver is **rk4** which implements the 4th-order Runge Kutta method with a fixed time step. The format is

```
out=rk4(x0,times,func,parms)
```

- x0 is the value of the state vector at times[1]
- times is a vector of the times at which you want solution values
- func is the function specifying the model (such as the **lotka** function above)
- parms is the vector of parameter values passed to func

So here is an example using our **lotka** function:

```
x0=c(2,1);
times=(0:200)/10;
parms=c(0.5,0.5);
out=rk4(x0,times,lotka,parms)
matplot(out[,1],out[,2:3],type="l",ylim=c(0,3));
```

As the last line indicates, **rk4** returns a matrix in which the first column gives the times at which solution values were obtained, and the remaining columns give the (approximate) state vector values.

You can do the same more compactly once you get the hang of it:

```
out=rk4(c(2,1),(1:200)/10,lotka,c(0.5,0.5))
```

That is, the values of the input arguments to rk4 are specified within the call, but remember that values have to go in the order that the function expects. But try

```
out=rk4(c(2,1),lotka,(1:200)/10,c(0.5,0.5))
```

and the result is an error message, because rk4 is expecting a numeric vector of times in the second slot, and instead it gets a function name.

The "industrial strength" solver is **lsoda**. lsoda is a "front end" to a general-purpose differential equation solver (called, oddly enough, lsoda) that was developed at Lawrence Livermore National Laboratory. The full calling format for lsoda is

```
lsoda(y, times, func, parms, rtol=1e-06, atol=1e-06, tcrit=NULL,
      jacfunc=NULL, verbose=FALSE, dllname=NULL, hmin=0, hmax=Inf)
```

Don't panic. Because defaults are provided for everything but the case-specific arguments required by **rk4**, lsoda can be called in exactly the same way as rk4:

```
out=lsoda(x0,times,lotka,parms)
matplot(out[,1],out[,2:3],type="l",ylim=c(0,3));
```

For the most part you can get away with this, so for now we will.

Exercise 11.1 Write a script file that uses lsoda to replicate the results in Figure 3.7 of Bulmer chapter 3, for the classical Rosenzweig-MacArthur predator-prey model (n_1 is the prey, n_2 the predator, and we write the model in the usual form)

$$\begin{aligned} dn_1/dt &= r_1 n_1(1 - n_1/K_1) - n_2 \frac{a_1 n_1}{B + n_1} \\ dn_2/dt &= n_2 \frac{a_2 n_1}{B + n_1} - r_2 n_2 \end{aligned} \tag{6}$$

and parameter values $r_1 = r_2 = 1, a_1 = a_2 = 2, B = 200$ with initial values $n_1(0) = 300, n_2(0) = 50$. The result to be replicated is that for $K_1 = 500$ predator and prey converge to an equilibrium, while for $K_1 = 650$ they settle into a steady pattern of cyclic oscillations. To graphically show what happens in each case, have your script file solve the model from $t = 0$ to 100, and plot the densities of predator and prey populations as a function of time.

Exercise 11.2 Modify your script file from the previous exercise to incorporate **predator switching** in the form of a type-III functional response. Specifically,
(a) add a parameter $q > 1$ affecting how the predator responds to changes in prey density:

$$\begin{aligned} dn_1/dt &= r_1 n_1(1 - n_1/K_1) - n_2 \frac{a_1 n_1^q}{B^q + n_1^q} \\ dn_2/dt &= n_2 \frac{a_2 n_1^q}{B^q + n_1^q} - r_2 n_2 \end{aligned} \tag{7}$$

(b) Use numerical solutions of the model (at judiciously chosen sets of parameter values) to explore how the stability of the interaction is affected as the value of q increases from 1. At parameters giving a stable equilibrium for $q = 1$, can increasing q destabilize the equilibrium and give rise to cycles? Conversely, at parameters giving rise to cycles when $q = 1$, can increasing q stabilize the equilibrium and eliminate the cycling?

11.1 Additional arguments in lsoda

The most important of the additional arguments are rtol, atol, hmin, and hmax.

rtol and atol control the accuracy that the solver tries to achieve (they stand for relative and absolute error tolerances). lsoda gets from one time-value to another by taking small steps, and adaptively adjusting their size (and number) to achieve the necessary accuracy, in particular so that the error in computing each new value of state variable x_i is less than $rtol\,|x_i| + atol$. So small values of rtol and atol give more accurate solutions, but it takes longer to do the computing.

hmin,hmax control the range of possible time-step sizes that can be used. Setting a value for hmax is just a bit of added security, to avoid lsoda becoming over-confident about how large a step it can take. This is most important when the model is explicitly time-dependent, and you want to make sure that lsoda doesn't jump over brief "shocks" to the system. Setting hmin is a way to ensure that lsoda doesn't grind down to taking infinitely many, infinitely short time steps.

12 Operations with Matrices and Vectors

A numerical function applied to a matrix acts element-by-element.

```
> A=matrix(c(1,4,9,16),2,2); A; sqrt(A);
     [,1] [,2]
[1,]    1    9
[2,]    4   16
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

The same is true for scalar multiplication and division. **Try** 2*A, A/3 **and see what you get.**

If two matrices (or two vectors) are the same size, then you can do element-by-element addition, subtraction, multiplication, division, and exponentiation:

```
A+B, A-B, A*B, A/B, A^B
```

NOTE: $\mathbf{A} \wedge \mathbf{B}$ means each element in **A** raised to the power given by the corresponding element in **B**. **This is not what** $\mathbf{A} \wedge \mathbf{B}$ **means in Matlab.** So for example:

$$\text{if } \mathbf{C} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \text{ then } \mathbf{C} \wedge \mathbf{C} = \begin{bmatrix} 1^1 & 2^2 \\ 3^3 & 4^4 \end{bmatrix}$$

Matrix-matrix and matrix-vector multiplication are indicated by the special notation %*%. For example,

```
> v=1:2; A%*%v
     [,1]
[1,]   19
[2,]   36
```

It's important to remember that element-by-element is the default, because **R** 's eagerness to make things work can sometimes let errors get by without warning. So for example suppose you had entered above

```
A*v
     [,1] [,2]
[1,]    1    9
[2,]    8   32
```

Since you (inadvertently) "asked" for element-by-element multiplication, that's what **R** did. It ran out of entries in v before it ran out of entries in A, so it "recycled" through

`outer(v,w)`	outer product of vectors v and w
`solve(A)`	inverse of matrix A
`solve(A,B)`	solution x of the linear system $Ax = b$ for each column b of the matrix B
`det(A)`	determinant of the matrix A
`norm(A)`	matrix norm of A (several options)
`eigen(A)`	eigenvalues and eigenvectors
`t(A)`	transpose of A
`apply(A,margin,fun)`	apply a function `fun` to the rows (margin=1) or columns (margin=2) of matrix A, and return all resulting values
`sapply(v,fun)`	apply a function `fun` to all elements in vector v, returning a vector of values

Table 7: Some important functions for matrix-vector numerical calculations. Many of these have additional optional arguments; use the Help system for full details

the elements of v as needed to make your command work.

$$\begin{bmatrix} 1 & 9 \\ 4 & 16 \end{bmatrix} * \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 1*1 & 9*1 \\ 4*2 & 16*2 \end{bmatrix}$$

The matrix algebra functions in **R** are front-ends for the "industry standard" numerical libraries. Some of these functions are listed in Table 7. Some of these only work on square matrices and will return an error if **A** is not square, in particular functions for computing eigenvalues and eivenvectors.

Several functions exist in two versions corresponding to older (LINPACK, EISPACK) and newer (BLAS,LAPACK,ATLAS) versions of these libraries. How you wind up using which is a lesson in how **R** works, and we leave it to the next section.

12.1 Eigenvalues and eigenvectors

In this section we consider only square matrices, for which eigenvalues and eigenvectors are defined. Recall that if $Aw = \lambda w$ (for A a square matrix, w a nonzero column vector, and λ a real or complex number) then λ is called an eigenvalue and w is the corresponding eigenvector of A.

If v is a row-vector such that $vA = \lambda v$, then v is called a *left eigenvector* of A. The left eigenvalues for a matrix are the same as the (right) eigenvalues. The left eigenvectors of **A** are the same as the right eigenvectors of the transpose $t(\mathbf{A})$.

`eigen` returns eigenvalues sorted by absolute value, with the dominant eigenvalue (meaning, the one with the largest absolute value) first. (the general definition of absolute value, which covers both real and complex numbers, is

$$|a + bi| = \sqrt{a^2 + b^2}).$$

For example,

```
A=matrix(1:9,3,3); vA=eigen(A); vA;
$values
[1]  1.611684e+01 -1.116844e+00 -9.357342e-17

$vectors
           [,1]       [,2]       [,3]
[1,] -0.5598757  0.8251730 -0.3767961
[2,] -0.6879268  0.2238583  0.7535922
[3,] -0.8159780 -0.3774565 -0.3767961
```

As with the output from `lm` (you remember `lm` ...) vA is an *object* composed of two *components*, whose names are `values` and `vectors`. The "$" is used to extract components of a compound object, for example `vA$values` is the `values` part of `vA`, a vector consisting of the sorted eigenvalues:

```
vA$values
[1]  1.611684e+01 -1.116844e+00 -9.357342e-17
```

The `vectors` component is a matrix whose columns are the corresponding eigenvectors, sorted in the same order as the eigenvalues. That is,

```
j=1; A%*%vA$vectors[,j]-vA$values[j]*vA$vectors[,j];
              [,1]
[1,]  0.000000e+00
[2,] -3.552714e-15
[3,] -3.552714e-15
```

Exercise 12.1 Verify that the output is also $(0, 0, 0)$, apart from numerical error, for j=2 and j=3. Explain in words what these calculations show.

Exercise 12.2 Enter the command `names(vA)` and see what results. From this, infer what the `names` function does. Enter the command `names(A)` and infer the meaning of the object `NULL` in **R** . Check your guess by using the Help menu on the console window: select R language (standard) and type NULL into the popup window that appears.

To get the left eigenvectors of a matrix, you use `eigen` to compute the eigenvectors of its transpose, `t(A)`. So

```
vLA=eigen(t(A))$vectors
```

gets you the left eigenvectors of A.

Exercise 12.3 Compute `vLA[,j]%*%A-vA$values[j]*vLA[,j]` to verify the last claim about left eigenvectors, for j=1 to 3.

For stability analysis of differential equation models, the "dominant eigenvalue" of the Jacobian matrix is the one with the largest real part. To find this we need to extract the real parts of each eigenvalue and identify the largest one.

The help system (`?complex`) lists the **R** functions for working with complex numbers. The one we need here is `Re`, which extracts the real parts of complex numbers.

Use `data.entry` to create the matrix

$$A = \begin{pmatrix} 3 & 0 & 0 \\ 2 & 2 & -3 \\ 0 & 3 & 1 \end{pmatrix}$$

and you should find that `eigen(A)$values` are

```
[1] 1.5+2.95804i 1.5-2.95804i 3.0+0.00000i.
```

The first two are a complex conjugate pair with absolute value 3.316625 (`mod(eigen(A)$values)` gets you the absolute values of the eigenvalues), but the third one has the largest real part.

To have **R** do this for you, we use the `which` function to find where the real part is maximized.

```
vA=eigen(A)$values; rmax=max(Re(vA));
j=which(Re(vA)==rmax);
lmax=vA[j]; vmax=eigen(A)$vectors[,j];
```

The first line uses `max` to compute the largest real part of any eigenvalue. In the second line, `which` finds the indices at which the logical condition (Re(vA)==rmax) is TRUE, in this case `j=3`. The third line then extracts the relevant entries from the lists of eigenvalues and eigenvectors.

Exercise 12.4 Try the above on `A=diag(c(3,1,3))` and see what you get for `lmax` and `vmax`. Why?

Exercise 12.5 In this exercise we take an eigenvalue-based look at a simple (and partially re-scaled) predator-prey model:

$$\begin{aligned} dx/dt &= x(1 - x/K) - yx^q \\ dy/dt &= cyx^q - my \end{aligned} \tag{8}$$

with x= prey density, y=scaled predator density, and K, q, c, m positive constants. The interior equilibrium for this model is at

$$\bar{x} = (m/c)^{1/q}, \qquad \bar{y} = (\bar{x}^{1-q})(1 - \bar{x}/K)$$

and the Jacobian matrix at a point (x, y) is

$$\begin{bmatrix} 1 - 2x/K - qyx^{q-1} & -x^q \\ cqyx^{q-1} & cx^q - m \end{bmatrix}$$

As usual with a type-II functional response ($0 < q < 1$) the model exhibits a transition from stability to cycles as K is increased (Figure 3).

(a) Write a function **domreal=function(K,q,c,m)** that computes and returns the largest real part of any eigenvalue for the Jacobian of model (8) evaluated at the interior equilibrium, as a function of the model parameters.
(b) Using your function, create a plot like that in Figure 3 panel (c), showing how the eigenvalues change as K is gradually increased from 200 to 350 with other parameters fixed at the values $q = 0.5, c = 0.1, m = 1$. ("gradually" means: compute enough values that you can plot a smooth curve as a function of K, rather than just two points connected by a line).

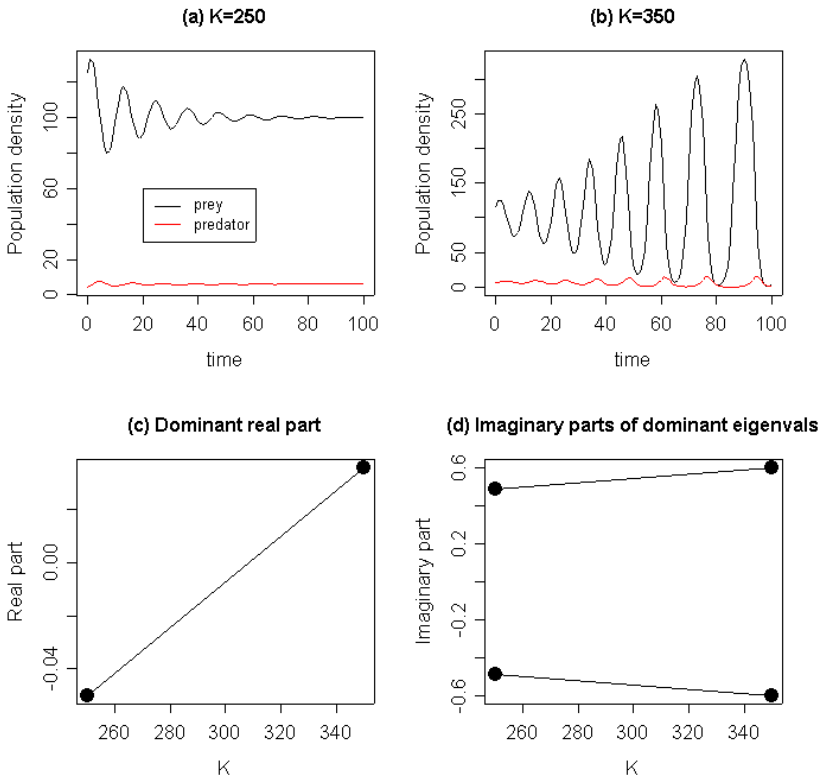

Figure 3: Hopf bifurcation in predator-prey model (8) with parameters $q = 0.5, c = 0.1, m = 1$. As K increases, a pair of complex conjugate eigenvalues crosses the imaginary axis in the complex plane.

12.2 Eigenvector scalings

For a transition matrix model, the dominant right eigenvector w (i.e. the eigenvector corresponding to the eigenvalue with largest absolute value) is the *stable stage distribution*, so we are most interested in relative proportions. To get those, `w=w/sum(w)`. The dominant left eigenvector **v** is the reproductive value, and it is conventional to scale those relative to the reproductive value of a newborn. If newborns are class 1: `v=v/v[1]`.

Exercise 12.6: Write a script file which applues the above to the matrices

$$A = \begin{pmatrix} 1 & -5 & 0 \\ 6 & 4 & 0 \\ 0 & 0 & 2 \end{pmatrix} \quad B = \begin{pmatrix} 0 & 1 & 5 \\ 0.6 & 0 & 0 \\ 0 & 0.4 & 0.9 \end{pmatrix}$$

finding **all** the eigenvalues and then extracting the dominant one and the corresponding left and right eigenvectors. For B, use the scalings defined above.

12.3 Eigenvalue sensitivities and elasticities

For an $n \times n$ matrix **A** with entries a_{ij}, the sensitivities s_{ij} and elasticities e_{ij} of the dominant (largest absolute value) eigenvector can be computed as

$$s_{ij} = \frac{\partial \lambda}{\partial a_{ij}} = \frac{v_i w_j}{\langle v, w \rangle} \qquad e_{ij} = \frac{a_{ij}}{\lambda} s_{ij} \tag{9}$$

where λ is the dominant eigenvalue, **v** and **w** are dominant left and right eigenvalues, and $\langle v, w \rangle$ is the inner product of **v** and **w**, computed in **R** as `sum(v*w)`. So once λ, **v**, and **w** have been found and stored as variables, it just takes some for-loops to compute the sensitivies and elasticities.

```
vA=eigen(A); lambda=vA$values[1];
w=vA$vectors[,1]; w=w/sum(w);
v=eigen(t(A))$vectors[,1]; v=v/v[1];
vdotw=sum(v*w);
s=A; n=dim(A)[1];
for(i in 1:n) {
    for(j in 1:n) {
        s[i,j]=v[i]*w[j]/vdotw;
    }
}
e=(s*A)/lambda;
```

Note how all the elasticities are computed at once in the last line. In **R** that kind of "vectorized" calculation is *much* quicker than computing entries one-by-one in a loop.

Even better is to use a built-in function that operates at the vector or matrix level. In this case we can use `outer` (see Table 7 or `?outer`) to completely eliminate the nested do-loops:

```
vA=eigen(A); lambda=vA$values[1];
w=vA$vectors[,1]; w=w/sum(w);
v=eigen(t(A))$vectors[,1]; v=v/v[1];
s=outer(v,w)/sum(v*w);
e=(s*A)/lambda;
```

Vectorizing code to avoid or minimize loops is an important aspect of efficient **R** programming. The functions `apply` or `sapply` are often useful in this regard and you should look into them if you have a script that is running too slow because of loops.

Exercise 12.7 Construct the transition matrix **A**, and then find λ, **v**, **w** for an age-structured model with the following survival and fecundity parameters.

Age-classes 1-6 are genuine age classes with survival probabilities $(p_1 p_2 \cdots p_6) = (0.3, 0.4, 0.5, 0.6, 0.6, 0.7)$

Note that $p_j = a_{j+1,j}$, the chance of surviving from age j to age $j+1$, for these ages. You can create a vector `p` with the values above and then use a for-loop to put those values into the right places in `A`.

Age-class 7 are adults, with survival 0.9 and fecundity 12.

Results: $\lambda = 1.0419$

$$A = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 12 \\ .3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & .4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & .5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & .6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & .6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & .7 & .9 \end{pmatrix}$$

$w = (0.6303, 0.1815, 0.0697, 0.0334, 0.0193, 0.0111)$
$v = (1, 3.4729, 9.0457, 18.8487, 32.7295, 56.8328, 84.5886)$

13 matrix or Matrix? Classes in R

This section can be skipped for now. It will make sense if you have previous experience with object-oriented languages. Othewise, come back to it after you've worked for a while with objects in **R** .

Variables and other kinds of objects in **R** can be assigned a *class* attribute which tells **R** what kind of object it is. As an example:

```
> g=matrix(0,5,5); class(g)
NULL
```

shows that the object g has not been assigned a class. On the other hand,

```
> library(Matrix); g=Matrix(0,5,5); class(g)
[1] "Matrix"
```

`library(Matrix)` loaded the `Matrix` library (which has some additional and higher-accuracy routines for matrix algebra). `g=Matrix(0,5,5)` created a 5-by-5 matrix of zeros, just as in the previous example, but this time the function `Matrix` assigned it a class.

Class attributes allow **R** functions to exist in multiple versions (called "methods"), that do different things to different classes of objects. You can list them using the `methods` function, for example

```
> methods(det);
[1] "det.default"             "det.LowerTriangular"
[3] "det.Matrix"              "det.UnitLowerTriangular"
[5] "det.UnitUpperTriangular" "det.UpperTriangular"
```

So the effect of the command `det(g)` depends on g's class. If g's class is Matrix, det(g) invokes det.Matrix to compute the determinant. If g's class is LowerTriangular (a kind of matrix) then deg(g) invokes det.LowerTriangular to compute the determinant, and so on. If g doesn't have a class for which a specific method exists, then det(g) invokes det.default.

Hopefully you now understand how **R** can have two sets of matrix algebra routines, as we mentioned above. The methods above for det are all part of the Matrix library (and if you try `methods(det)` without loading the Matrix library, you won't see them listed). With the Matrix library loaded, a plain old matrix will have its determinant calculated by det.default, while a matrix of class Matrix will be handled by det.Matrix.

The convenience of classes and methods is illustrated better `methods(plot)`. The list shows that **R** has a series of methods that generate suitable plots for a variety of classes (including statistical models, time series, functions, etc.) and the appropriate one is usually invoked "under the hood" without the user having to specify which one you want.

The inconvenience of is that some **R** functions require arguments of a specific class, and you may then have to devote some effort to converting things from one class to another

(using functions like `as.matrix`). If you get an error message indicating that you're in this kind of trouble, use the help system to see if a function `as.whatever` exists that can convert into the necessary class; usually there will be one, if what you're trying to do is legitimate.

14 Optimization and fitting models to data

Optimization is **extremely important** and **extremely difficult**. A typical problem for modelers is the following: given a set of data and a model, we want to choose model parameters to fit the data as well as possible. Here is an example: a data set $(x(t), t = 1, 2, \cdots, 14)$ is stored in vectors **tvals,xvals** and we want to fit those data with a differential equation, the so-called θ-logistic model

$$dx/dt = rx(1 - (x/K)^{\theta}).$$

```
tvals=1:14;
xvals=c(15.58,30.04,66.05,141.6,274.6,410.0,468.8,526.4,472.5,
496.6,489.5,492.0,496.8,473.0);

ThetaLogist=function(t,x,parms) {
    r=parms[1]; K=parms[2]; theta=parms[3]
    xdot=r*x*(1-(x/K)^theta);
    return(list(xdot));
}
```

The first step is to choose an **objective function** $F(p)$ - a measure of how close the model is to the data, as a function of the parameter vector p. The objective function must be set up so that **small is beautiful** – a small value of F means a good fit to the data. One commonly used measure is the *least squares* criterion: choose parameters to minimize the sum of squared deviations between data and model output

```
TLfit=function(p) {
parms=p[1:3]; x0=p[4]; times=1:14;
out=lsoda(x0,times,ThetaLogist,parms,hmin=0.001);
mse=mean((out[,2]-xvals)^2);
return(mse);
}
```

Note that the model has 3 parameters, but the objective function **TLFit** has 4 arguments (i.e. p is a vector of length 4). The additional argument is the initial value $x0$. Were we to use xvals[1] as $x0$ we would be giving that value special treatment, insisting that the model solution pass exactly through that value. To put all data points on an equal footing, we have to treat $x0$ as another unknown value to estimate.

Now we want to minimize TLFit as a function of p. Calculus is no help: we can compute values of TLfit but there is no formula, so we can't use a "set derivatives to 0" approach. Instead we hand the problem over to an **optimizer**, a routine that searches for the minimum of a function. Because optimization is important for lots of things – not just theoretical ecology, but really important things like devising delivery routes to

provision each McDonalds at minimal expense – considerable effort has gone into developing software that can hunt efficiently for minima of functions. In **R** these are accessed through the functions **optim** and **nlm**. The former has the advantage of providing a common interface to several methods, so we will discuss **optim** here, but use of **nlm** is very similar. For the example above,

```
p0=c(1,5,1,15);
fit=optim(p0,TLfit);
fit2=optim(fit$par,TLfit);
fit$par; fit2$par;
```

p0 is our initial guess, based on eyeballing the data for $r, K, x0$ and starting with a standard logistic ($\theta = 1$). The next line does a minimal call to **optim**, providing the objective function name and the initial guess, and accepting default values for everything else.

The default optimization method is the Nelder-Mead Simplex Algorithm, discussed below. This is a robust method intended for functions where it is costly to compute derivatives of the objective function, so it is best to search based only on values of the objective function. As a safeguard, we re-optimize starting from the supposed minimum identified on the first call, and check to see that we converged back to the same parameter estimates. This is relatively quick if the first search found the minimum, and if not you want to know about it so that you can try again or switch to another search method. In this case the re-optimization makes no significant adjustments to parameters, so we're done and can plot the results (Figure 4) – see **FitThetaLogist.R** for the complete code.

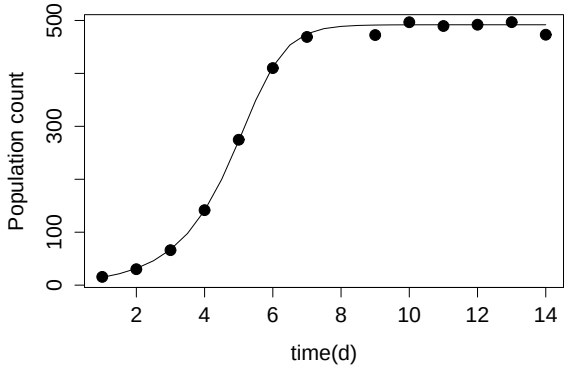

Figure 4: Results from least-squares fitting of the θ-logistic model. Points are the data, curve is the model solution with best-fitting parameters

Exercise 14.1 Type `names(fit)` at the console, and then use **?optim** to find out what all the parts are of the object returned by **optim**.

Exercise 14.2 If we reduce the number of parameters by assuming that $\theta = 1$ (a standard logistic model) can you fit the data as well? Modify the script file for the example so that $\theta = 1$, i.e. $dx/dt = rx(1 - x/K)$, and see if the model can fit the data as well as when we also allow θ to be fitted.

Exercise 14.3 Write a script file that uses optim to fit the same data as in the example above, by the difference equation model

$$x_{t+1} = \frac{\lambda x_t}{1 + bx_t}$$

[in plant ecology this is known as the Reciprocal Yield Law model]. Have your script produce a plot of data and fitted model like Figure (4). Note that instead of calling **lsoda** to solve the model, you will have to write a for-loop (inside the objective function) to compute solutions of the model. You'll also need an initial guess for the parameter values. The initial growth when x is small gives you an idea of what λ ought to be, and then you can estimate b so that the model's equilibrium corresponds to what you see in the data.

14.1 What to optimize?

This is not an easy question, because it depends on what you believe about the actual system generating the data. In our θ-logistic example above, we asked for a single, deterministic solution of the model to pass close to all the data points. This is called *calibration* or *trajectory matching*. It assumes that the real system is deterministic (noiseless), and that all differences between data and model output are due to measurement error (i.e. that the model is perfect but the data are not). In that case the objective function can be chosen to correspond to the statistical principle of **maximum likelihood**: parameters are chosen to maximize the likelihood of the observed data, based on the statistical distribution of the measurement errors.

If the errors have a Normal distribution with constant variance, it is a standard result from statistics that maximum likelihood is equivalent to the least-squares objective function, as in the example above. Other assumptions about measurement errors lead to different objective functions. This is really a topic in statistics, so we will mention only one other important case. If the errors are Poisson (which is often true or close-enough for population samples) then maximum likelihood is *approximately* equivalent to least-squares after transforming data and model to square-root scale. That is, we change the objective function by redefining

```
mse=mean((sqrt(out[,2])-sqrt(xvals))^2);
```

Exercise 14.4 Does this improved objective function make a difference in our θ-logistic example – that is, does the change in objective function result in any meaningful change in

parameter estimates or the fitted model solution? Modify the script file for the example, compute the parameter estimates with the modified objective function, and compare the two sets of parameter estimates.

Trajectory matching can run into problems if the true system is not deterministic. For example, consider a discrete-time model with random noise

$$X_{t+1} = F(X_t) + \sigma Z_t \tag{10}$$

where X_t is a vector of state variables, and Z_t is a vector of random perturbations to the state variable dynamics. The presence of random noise can change qualitative properties of model solutions. For example, a model that (in the absence of noise) would exhibit oscillatory convergence to a steady state, may instead exhibit fairly regular cycles. Fitting such data by trajectory matching will lead to incorrect parameters such that the model has stable periodic solutions in the absence of noise. Other fitting criteria are therefore more appropriate.

If error-free data are available on all state variables, then (10) is a nonlinear regression model, which can be fitted by any modern statistics package using nonlinear least squares (in **R** this is function **nls**). If the data are not totally accurate it's a more difficult regression problem but still doable (functions in the **nlme** library).

More typically one only has data on one or a few state variables. You can then try to choose parameters (including the noise level σ) so that model output matches overall features of the data such as the range of variation, periodicity of cycles if they occur, and the pattern of correlation between successive values. This is called "moment matching" or "method of moments" in the statistical literature. It suffers from the fact that the choice of which features to match is subjective, but has the advantage of that very few assumptions are needed about the underlying model, in order for parameter estimates to have good statistical properties.

Exercise 14.5 Here again is the Rosenzweig-MacArthur model

$$\begin{aligned} dn_1/dt &= r_1 n_1(1 - n_1/K_1) - n_2 \frac{a_1 n_1}{B + n_1} \\ dn_2/dt &= n_2 \frac{a_2 n_1}{B + n_1} - r_2 n_2 \end{aligned} \tag{11}$$

Suppose that parameter values $r_1 = r_2 = 1, a_1 = 2, a_2 = 1.8, B = 200$ are known, and you have data showing the predators cycling regularly between a minimum value of about 95 and a maximum of about 205; you have no data on the prey. Write a script file to estimate K_1 by matching these features as closely as possible. That is, your objective function will take a single argument K_1, generate solutions of the model, and return some measure of how well the model solutions match the observed features of the data. Since there is only one parameter to be fitted, for this exercise it will be sufficient to plot the objective function versus the value of K_1, and see where it is minimized (for a more accurate result, see **?optimize**).

14.2 Controlling the optimization

Parameters affecting how **optim** proceeds can be set using the optional argument **control**, which is a list of options and their values. For example,

```
fit=optim(p0,TLfit,control=list(method="BFGS",maxit=1000,trace=6));
```

Here we have: changed the optimization method to BFGS, set the maximum number of search steps to 1000, and set the level of tracing to 6 (tracing means printing information to the console window as optimization proceeds, which is useful if things are going badly and you want to know why).

There are a lot of optional arguments – see **?optim**. Default values are well chosen except that the iteration limit `maxit` is suitable for "easy" problems where the function is not too wiggly or convoluted. If you often find that your safeguard optimization (fit2 in the example above) is different from the first, try increasing the number of iterations. If optimization is generally working but takes a long time, you might try decreasing the error tolerances (`abstol` and `reltol` in `control`). Optim tries to find the minimum with the maximum possible accuracy (given the number of digits used to represent real numbers). Sometimes that results in a lot of time being spent making tiny, meaningless adjustments to parameter estimates (e.g. involving more significant digits than are reliable in the data).

Exercise 14.6 We need optimization for reasons besides parameter estimation. Consider 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 gain function $g_2(t) = t^{0.7}$. The code snippet below (from an old lab solution) defines the gain rate as function:

```
g1=function(t){2*t^0.5}
g2=function(t){t^0.7}
gainR=function(t1,t2) {
  return((0.7*g1(t1)+0.3*g2(t2))/(3+0.7*t1+0.3*t2))
}
```

Write a script that uses **optim** to find the values (t_1^*, t_2^*) at which the rate of energy gain $R(t_1, t_2)$ is maximized, using the BFGS method. Recall that for **optim** the objective function must have one input argument, not two. Also recall that **optim** finds where a function is minimized; to find where a function $F(x)$ is maximized you can use $-F(x)$ as the objective function.

14.3 How to optimize

The basic decision in optimization is how you want to get from your initial guess to a final value: a few clever steps, or a lot of simple steps. This discussion is based on Kelley

(1999), the optimization chapter in Numerical Recipes, and much painful experience trying to fit models to data.

"Clever" means using a lot of information about the function, either computed at each new evaluation point (e.g. its derivatives and possibly second derivatives with respect to each argument) or else built up gradually over the search for the minimum. Using this information the algorithm tries to find the nearest local minimum of the function, typically by constructing a well-informed guess as to the direction to the minimum from the current evaluation point, and then searching for the smallest function value *in that direction*. For example, the local first and second derivatives define a quadratic surface approximation, and the search direction points at the minimum of that surface.

Such cleverness is often counterproductive, since your function may have many local minima that are much less good than the global minimum. To find the global minimum you then have to repeat the optimization many times, starting from different initial points. The efficiency of each repetition can easily be more than outweighed by the need to run many repetitions.

Less clever method often fare better against multiple local minima, because they work by gradually narrowing in on a minimum, and so initially they don't pay much attention to the function's fine-scale structure (Kelley 1999). In our example above we used one such method, the Nelder-Mead Simplex Algorithm. Initially, values of the function are computed at a set of points (a *simplex*) surrounding the initial guess (n points for a function of n variables). At each step, the algorithm tries to replace the worst point (the one with the highest function value) with a better one, along the line between the worst point and the average of all the others, by evaluating the function at a few pre-chosen points along that line. If that fails, it may shrink the simplex down around the best point on the simplex.

More clever methods in **optim** include **BFGS** and **L-BFGS-B**. Both of these use derivatives of the objective function, either supplied by the user as a function or else computed numerically by forming difference quotients. As the function is repeatedly evaluated in the search for the minimum, values of the derivatives at those points are used to build up an approximation to the matrix of second derivatives, which is then used as the basis for a quadratic approximation. If your function is not too wiggly, this intermediate level of cleverness is often very effective, so long as it is coupled with a strategy for junking the approximate second derivative matrix and starting over from scratch if you get into trouble.

15 Schedule of lab sessions for Spring 2003

Week 1, January 20

- Read sections 1 through 5
- Do in lab: Exercises 2.1, 2.2, 4.1, 4.2, 4.3.
- Homework: Exercises 4.4 and 4.5; Install R on your computer, and then download and install the **boot** library (in Windows, you can do this easist via the console window menus **Packages/Install Package from CRAN**.

Week 2, January 27

- Read sections 6 and 7 (vectors and matrices)
- Do in lab: all exercises in the sections
- Homework: Write a script file to compute $R(n)$ for any one value of n in Lecture Exercise 1.8.

Week 3, Feb 3

- Section 8 (Loops), omitting **A simulation project**.
- Do in lab:
- Homework: exercises 8.5,8.6

Week 4, Feb 10

- Section 9 (Branching) and 10 (functions)
- Do in lab:
- Homework: exercises 8.5,8.6

Week 5, Feb 17
In class: simulation project, exercise 8.8
HW: work with the program, exercise 8.9

Week 6, Feb 24
In lab: ode solutions, 11.1 HW: 11.2

Week 7 In lab: matrix operations, eigenvalues/eigenvectors

16 References

G. Fussmann, S.P. Ellner, K.W. Shertzer, and N.G. Hairston, Jr. 2000. Crossing the Hopf bifurcation in a live predator-prey system. Science 290: 1358-1360.

Ihaka, R., and R. Gentleman. 1996. R: a language for data analysis and graphics. Journal of Computational and Graphical Statistics 5: 299314.

Kelley, C.T. Iterative Methods for Optimization. SIAM (Society for Industrial and Applied Mathematics), Philadelphia PA.