An introduction to programming in R

Compiled by Paul Hurtado[1]
Mathematical Biosciences Institute, and
Department of Evolution, Ecology and Organismal Biology
The Ohio State University
hurtado.10@mbi.osu.edu
Last compile: June 17, 2013

Contents

[1]These notes are largely copied from *An introduction to R for theoretical ecology* by Stephen P. Ellner ©2009, with slight modifications since then by Paul Hurtado. That document is based in part on course materials by Ellner's 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**. This document also draws from Ecological Models and Data in R ©Benjamin M. Bolker.

1 Introduction

These are notes for the the Cornell course BioEE 7600 *Ecological design and analysis I: programming in R* . They refer frequently to the optional text for the course, *Ecological Models and Data in R* by Benjamin Bolker ©2008. The text book itself is optional, as I've tried make these notes follow both the published text as well as the draft book chapters available online at the books' website:

http://people.biology.ufl.edu/bolker/emdbook/index.html

The notes are based in large part on course materials provided by Stephen P. Ellner, which themselves include contributions from former TAs Colleen Webb, Jonathan Rowell and Daniel Fink at Cornell, Professors Lou Gross (University of Tennessee) and Paul Fackler (NC State University), and the book *Getting Started with Matlab* by Rudra Pratap (Oxford University Press). They also draw on the documentation supplied with R . There may in places be a strong resemblance to the notes on R that accompany the textbook *Dynamic Models in Biology* by Steve Ellner and John Guckenheimer (Princeton University Press 2006).

These notes are written under the assumption that you have a computer (running R) in front of you. It is important to work through examples yourselves - **i.e. type them into R and see what happens on your screen** - to get the feel and experience of working in R . Exercises or examples in the middle of a section (here or in the text) should be done immediately when you get to them, making sure you have them right before moving on. Additional exercises may be given at the end of some sections.

What is R ? R is an object-oriented scripting language that combines

- the S programming language developed by John Chambers (Chambers and Hastie 1988, Chambers 1998).
- a user interface with a few basic menus and extensive help facilities.
- an enormous set of functions for classical and modern statistical data analysis and modeling.
- graphics functions for visualizing data and model output.

R is an open-source project (R Core Development Team 2005) available free via the Web. Originally a research project in statistical computing (Ihaka and Gentlemen 1996) it is now managed by a team that includes many well-regarded statisticians, and is widely used by statistical researchers and a growing number of theoretical biologists. The commercial implementation of the S language (called S-plus) offers a "point and click" interface that R lacks. However for our purposes this is outweighed by the fact that S-plus still lacks some essential tools for simulating dynamic models. The standard installation of R includes extensive documentation, including an introductory manual and a comprehensive reference manual. At this writing, R mostly follows version 3 of the S language, but some packages are starting to use version 4 features. *These notes refer only to version 3 of* S. We also limit ourselves to graphics functions in the base graphics package; much more is available in packages such as **lattice** and **grid**.

1.1 Getting and installing R

The main sources for R are CRAN (http://www.cran.r-project.org) and its mirrors. You can get the source code, but most users will prefer a precompiled version. To get one from CRAN, click on the link for your OS, continue to the folder corresponding to your OS version, and find the appropriate download file for your computer.

For Windows or OS X, 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), but many LINUX distributions include a fairly up-to-date version of R .

For Windows PCs it is suggested that you edit the file `Rconsole` in R 's `etc` folder and change the line `MDI=yes` to `MDI=no`, and also edit `Rprofile` to un-comment the line `options(chmhelp=TRUE)` by removing the # at the start of the line. These changes allow R 's command and graphics windows to move independently on the desktop, select the most powerful version of the help system and reduce the rate of inexplicable crashes on Windows laptops.

This document was written at a Windows PC and sometimes refers to Windows-specific aspects of R ; please let me know about these so I can correct them.

1.2 First 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 is 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.

There are some rules for giving names to variables:

- 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`.
- **Do not** use the underscore character ('_') or blank space as a separator in variable names (R will probably give you an error message if you do).
- 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
```

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**. Use the Help system to get a more complete list: ?Arithmetic for simple functions, ?log for logarithmic, ?sin for trigonometric, and ?Special for special functions.

Even though the values for `x, y` were not displayed, R "remembers" that values have been assigned to them. Type `> x; y` to display the values.

The ↑ key recalls previous commands to the prompt, so they can be corrected or modified. 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
```

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
```

It's generally best to *use parentheses to specify exactly what you want*, such as

```
> 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 1.1 : Have R compute the values of

1. $\frac{2^7}{2^7-1}$ and compare it with $\frac{1}{1-2^{-7}}$.
2. $\sin(0.25), \cos^2(0.5)$ [Note that typing `cos^2(0.5)` won't work!]
3. $\frac{2^7}{2^7-1} + 4\sin(0.25)$, using cut-and-paste to assemble parts of your past commands.

Exercise 1.2 : Do an Apropos on `sin` via the Help menu, to see what it does. Next, enter the command `help.search("sin")` and see what that does (answer: `help.search` finds *all* Help pages that include 'sin' anywhere in their title or text, most having nothing to do with trig functions. `Apropos` just finds functions that have 'sin' somewhere in their name).

Exercise 1.3 : Work through the *Sample Session* at the end of Bolker, Chapter 1. Use the help system (or your neighbor) to do a linear regression of the *frog-tadpole* data. (Tip: Make use of the online version of the text to copy/paste the frogs data. Try entering *help.search("fitting linear models")* or *?lm* in the R console.)

Reading[2] **Assignment:** Bolker, Chapter 1

- Read pages 1-4 (1-4)
- Skim pages 4-17 (4-20)
- Read 17-22 (20-26).

[2] As a convention, book chapters and page numbers will be given for the printed version of the text, followed by the corresponding chapters and page numbers for online version in parentheses.

1.3 More R basics: functions and their documentation

As seen above, R has a nice help system that can be queried in multiple ways. From the command line, these include:

```
> help.search("search string")
> ?function.name
> apropos("search string")
```

To best utilize the help pages for specific functions, let us take a closer look at what functions are and how they work.

At the command line, type the name of the function *read.csv*, no parentheses. You should get something like

```
> read.csv
function (file, header = TRUE, sep = ",", quote = "\"", dec = ".",
    fill = TRUE, comment.char = "", ...)
read.table(file = file, header = header, sep = sep, quote = quote,
    dec = dec, fill = fill, comment.char = comment.char, ...)
<environment: namespace:utils>
```

In R , typing the function name alone returns the function itself[3]. If you go to the help by typing *?read.csv* you'll notice similarities between the first line above and the *read.csv* entry in the help page.

Each function takes a set of arguments, which may or may not have a default value, and returns some object. In this case, *read.csv* is a function that reads in data from a comma separated file and stores it as a variable in R. In this case, *file* is the name of the file to open, *header* is a logical (true/false) variable indicating whether or not the first row is data or column names, and so on. Here *file* has no default file name, whereas *header=TRUE* means that if omitted, the function assumes the first row gives the column names.

If you had a CSV file **without labeled columns** in the following directory[4], you could load it into a variable *my.data* and display the contents to the screeb by typing

```
> my.data = read.csv(file="C:/MyDirectory/Data/frog-tadpole.csv");
> my.data
```

Notice anything funny? Try it again but this time type

```
> my.data = read.csv(file="C:/MyDirectory/Data/frog-tadpole.csv", header=F);
> my.data
```

In the first case, the default *header=T* told read.csv to take the first row as the column names. Since variables can't start with a number, they recieved the column names *X1.1* and *X2.036981755* instead. In the second case, they recieved the default *V1* and *V2*. We can change these by

```
> names(my.data) = c("frogs", "tadpoles");
> names(my.data)
> my.data
```

[3]Compare this to typing *read.csv()* and *"read.csv"*

[4]First, type getwd() at the command line to see what your current "working directory" is. If you need to change it, use setwd(). Next, go to the course website and download the frog-tadpole.csv file to that directory. Edit the directory name in the example code accordingly.

aov, anova	Analysis of variance or deviance
lm, glm	Linear and generalized linear models
gam, gamm	Generalized additive models and mixed models (in **mgcv** package)
nls	Fit nonlinear models by least-squares (in **nls** package)
lme,nlme	Linear and nonlinear mixed-effects models (in **nlme** package)
nonparametric regression	Various functions in numerous libraries including **stats** (smoothing splines, loess, kernel), **mgcv, fields, KernSmooth, logspline, sm**
Commmander	Point-and-click GUI for basic statistics and model fitting, can import data from SPSS, Minitab or STATA data files (in package **Rcmdr**)
boot	Package: functions for bootstrap estimates of precision and significance
multiv	Package: multivariate analysis
survival	Package: survival analysis
tree	Package: tree-based regression

Table 2: A small selection of the functions and add-on packages in **R** for statistical modeling and data analysis. There are **many** more, but you will have to learn about them somewhere else. A number of free manuals are available at CRAN `www.cran.r-project.org`.

Exercise 1.4 : The best way to get a feel for how functions work is to create your own. Here we'll create a function *square* that returns the square of it's argument, or if no argument is given the default value is a random integer from 0 and 10. This defines the function:

```
> square = function(x = round(runif(1,min=0,max=10)) )   {
>   return(x^2);
> }
```

What happens when you type the following?

```
> square(12)
> square()
> square
```

Exercise 1.5 : Plot our data using *plot(my.data)*. Read the help for the plot command, and plot it with a new title and axis labels of your choosing.

1.4 Statistics in R

Some of the important functions and packages for statistical data analysis are summarized in Table 2. The book *Modern Applied Statistics with S* by Venables and Ripley gives a complete practical overview. For the basics Maindonald (2004) and Verzani (2002) are available free from CRAN (`www.cran.r-project.org`), and you can also find there a complete list of libraries and their contents (click on `Package sources`). There is also a large collection of R packages for the analysis of genomic data distributed at `www.bioconductor.org`. For the most part, we will not go too deeply into this side of R - at least until the last few weeks of the course.

2 Common R Objects: Vectors, Matrices, and Data Frames

Understanding how to use vectors and matricies is fundamental to capitalizing on R 's computational abilities and for doing scientific computing in general. That said, the data frame is also common to work with in R . In this next section, we get familiar with all three.

2.1 Vectors

Vectors and matrices (1- and 2-dimensional rectangular arrays of numbers) are pre-defined data types in R . 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.csv()`:
`my.data=read.csv("c:\\temp\\frog-tadpole.csv", header=F)`
(**Remember:** this only works if the file that you're trying to read actually exists!)

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 that operations and functions act on 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. In MATLAB, matrix operations are the default and element-by-element operations require special notation.

2.1.1 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 `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 2.1 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 .

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

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
```

`rep(values,lengths)` can also be used with a vector of values and their associated lengths, for example

```
> rep( c(3,4),c(2,5) )
[1] 3 3 4 4 4 4 4
```

The value 3 was repeated 2 times, followed by the value 4 repeated 5 times.

Some of the main functions for creating and working with vectors are listed in Table 3. R also has many 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, 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.

Some of the functions for creating length-n vectors of random numbers are listed in Table 4. We will see soon some examples of using these functions to simulate models with random components to their dynamics.

Exercise 2.2 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.

2.1.2 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
```

`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()`. 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).

`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. to extract the 2^{nd} through 5^{th} elements of `q`

```
v=q[2:5]; v
[1] 3 5 7 9
```

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

Exercise 2.4 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 2.5 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 $n = 50$.

Vector orientation. 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.

2.2 Matrices

2.2.1 Creating matrices

A matrix is a two-dimensional rectangular array of numbers. Matrices can be created by reading in values from a data file using `read.table`, or from a properly formatted .csv spreadsheet file using `read.csv`. 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
```

By default the values in the data vector are put into the matrix column-wise. 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 2.6 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 2.7 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)` creates a square matrix with data vector v on its diagonal – try

```
diag(1:3)
```

and see what happens. `diag(v,n)` creates an $n \times n$ matrix with v on the diagonal, and if necessary it "recycles" through v to find n values. To see what that means, try

```
diag(1:3,5);   diag(1,6)
```

to see what they create (and make sure you understand why!)

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

`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)
`t(x)`	transpose the matrix x
`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.

2.2.2 cbind and rbind

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

`cbind` binds together columns of two objects. One thing it can do is put vectors together to form a matrix:

```
> C=cbind(1:3,4:6,5:7); C
     [,1] [,2] [,3]
[1,]    1    4    5
[2,]    2    5    6
[3,]    3    6    7
```

Remember that R interprets vectors as row or column vectors according to what you're doing with them. Here it treats them as column vectors so that columns exist to be bound together. On the other hand,

```
>  D=rbind(1:3,4:6); D
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

treats them as rows. Now we have two matrices that can be combined.

Exercise 2.8 Verify that `rbind(C,D)` works, `cbind(C,C)` works, but `cbind(C,D)` doesn't. Why not?

2.2.3 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 2.9 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`.

2.3 Data Frames (and more R tricks)

A common data type in R is the *data frame*, which has many of the same properties as lists, vectors and matrices. Most of what we'll do later in the course draws from our intuition about vectors and matrices. Still, data frames are ubiquitous and knowing how to use them will come in handy later on. Noteably, this is the data type returned when we load data into R , so lets pull up the frog-tadpoles data.

```
> my.data = read.csv(file=file.choose(), header=F);
> names(my.data) = c("Frogs","Tadpoles");
```

We can querry what type of object `my.data` is using the commands `mode`, `typeof`, which tell us it's of the generic type `list`. We can get more specific using `class`, or any of the many `is.`(data type) commands:

```
> class(my.data);
> is.matrix(my.data);
> is.data.frame(my.data);
```

To see other such function, use the autocomplete capabilities in R . Type the start of a function name then hit tab twice to show a list of possibilities:

```
> is.<TAB><TAB>
is.array           is.atomic          is.call            is.character       ...
is.double          is.element         is.empty.model     is.environment     ...
...
```

In short, data frames are most useful because they can have columns of different data types:

```
> data.frame(chars=letters[1:10],ints=90:99,logicals=rep(c(TRUE,FALSE),5));
```

2.3.1 Data frame addressing

Data frames can be treated just like matrices, addressing by row and column:

```
> my.data;
> my.data[2:4,]
  Frogs Tadpoles
2   1.3 2.876231
3   1.7 3.062528
4   1.8 3.707180
> my.data[2:4,2]
[1] 2.876231 3.062528 3.707180
```

Data frames can also have rows and columns indicated by their names, using either square brackets or the special character `$`:

```
 > names(my.data)
[1] "Frogs"    "Tadpoles"
> my.data$Frogs;
 [1] 1.1 1.3 1.7 1.8 1.9 2.1 2.3 ...
> my.data$Tadpoles;
 [1] 2.036982 2.876231 3.062528 3.707180 ...
```

`data.frame(u,v,...)`	combines columns u, v into a data frame
`names(x)`	access the column names in the data frame x
`as.data.frame(x)`	convert an object of some other type to a data frame, if possible
`attach(x)`	include columns of x in the workspace
`head(x)`	view the first few rows of x
`tail(x)`	view the last few rows of x

Table 6: More functions for creating and working with data frames. Also see the functions for matrices like cbind. See the help for details.

```
> my.data["Frogs"];
   Frogs
1    1.1
2    1.3
...
> my.data[3:5,"Tadpoles"];
```

Exercise 2.10 Use the `data.frame()` and `names()` functions to make the following data frame

```
  sex weight
1   M    135
2   F    106
3   M    164
4   F    127
```

What happens if you use `as.matrix()` to convert it to a different data type? What about with `as.vector()`?

2.3.2 More on vector, matrix and data frame addressing

One somewhat advanced (and R specific) way of pulling out specific elements of a vector (and rows or columns of a matrix or data frame) is using a vector of logicals (`TRUE`, `FALSE` statements). Indexing with a vector of `TRUE/FALSE` values, only the elements with a `TRUE` are returned.

```
> set.seed(101)
> x=runif(9); x
[1] 0.372 0.0438 0.709 0.657 0.249 0.300 0.584 0.333 0.622
> x[rep(c(T,F,F),3)]
[1] 0.372 0.657 0.584
```

Thus the equivalent to returning "all x less than 0.6" is accomplished by

```
> x<0.6
[1]  TRUE  TRUE FALSE FALSE  TRUE  TRUE  TRUE  TRUE FALSE
> x[x<0.6]
[1] 0.37219838 0.04382482 0.24985572 0.30005483 0.58486663 0.33346714
```

Exercise 2.11 Use the above and the frog-tadpole data to determine which in rows there are more than twice as many tadpoles as frogs (i.e. `2*Frogs < Tadpoles`). Can you create a third column (e.g. `my.data$Over`) indicating this outcome for each row?

Additional Exercies: Vectors, Matrices, Data Frames

Exercise 2.12 Before typing in the commands, write down the matrices *that you think* will be returned by `matrix(1:4)`, `matrix(1:4,2,2)`, and `matrix(1:4,2,2,byrow=F)`. After confirming your answers with R , see the help for `matrix()` to describe any discrepencies or confirm your correct reasoning.

Exercise 2.13 Use `diag()` and then `data.entry()` to create the following matrix `A`.

```
> A
     var1 var2 var3 var4
[1,]    1    0    5    7
[2,]    0    2    6    8
[3,]    0    0    3    0
[4,]    0    0    0    4
```

Replace the four non-zero values in the upper right corner of the matrix

```
> A[1:2 , 3:4]
     var3 var4
[1,]    5    7
[2,]    6    8
```

with the 2x2 matrix given by `matrix(1:4,2,2,byrow=TRUE)`.

Exercise 2.14 Frequently, a column or two of a large data frame is used to define a new object. Depending on the data type we want for this new object (e.g. a vector, matrix or data frame), we can extract the columns (or rows) in different ways. Using the frog-tadpole data, and recalling that `my.data` is a data frame, we can extract the first column of data using either: `my.frogs = my.data[,1]`, `my.frogs = my.data$Frogs`, `my.frogs = my.data["Frogs"]`, or `my.frogs = my.data[[1]]`. Try each, and describe how these differ (or are the same) using the functions like `mode`, `class`, `is.matrix`, `is.data.frame`, etc.

3 Iteration ("Looping")

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

- Making population forecasts 1 year ahead, then 2 years ahead, then 3, etc.
- Simulating a model multiple times, with different values for one of the parameters.

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

3.1 For-loops

The **for** loops are written as

```
for (var in seq) {
      commands
      }
```

The dummy variable `var` is sequentially assigned the values in the list `seq`, each time executing the code between braces `{ }`. Here's an example (file **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") for printing 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 the **C** language) so that each (n,x) pair is put on a separate line.

Several `for` loops can be nested within each other, which is needed for working with matrices 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 or copy this into a script file 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 - this is an easy way to create a matrix of whatever size you need, which can then be filled in with meaningful values as your program runs. Then two nested loops are used to fill in the entries. 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 also let us automate the process of 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 does a "population growth" simulation Line 8 then closes the loop over initial sizes

The result when you run **Loop2.R** is that the "population growth" calculation 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 the formatting was done.

If this discussion of looping doesn't make sense to you, **stop now and ask for some help**! Loops are essential from here on out.

Exercise 3.1 : Imagine 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 of them in you, and only 120 in your assistant. However, your field-hardened immune system is more effective. In your body the number of parasites grows by 10 percent each day, while in your assistant's it increases by 20 percent each day. That is, after your return to Ithaca your parasite load $n(j)$ grows according to the rule `n[j+1]=1.1*n[j]` where `j` is time in days, and your assistant's parasite load grows according to the rule `m[j+1]=1.2*m[j]`.

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 draws a single plot of both on log-scale (i.e. $\log(n(j))$

and $\log(m(j))$ versus time for j=1 to 30 days). **Save the script!** You'll be using it later, as the starting point for some other exercises.

Exercise 3.2 : Write a script file that uses for-loops to create the following 5×5 matrix A. Think first: do you want to use nested loops, or sequential?

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 3.3 Write a `for` loop that fills vector X with the first 100 Fibonacci numbers starting at 0.

Exercise 3.4 Write a script file that uses a for-loop to calculate solutions of the difference equation model

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

up to $j = 13$, where $r(j) = 2e^{-0.2j}$. Remember that in R , e^x is computed using the `exp` function. Write your script so that all values of $r(j)$ are computed and stored in a vector before the start of the for-loop, and have your script plot the solution as a graph of $N(j)$ versus j with the values plotted as points and connected by lines.

Exercise 3.5 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 can use the R script editor or any other text editor to create r.txt and put into it any *positive* numbers that you like. This would be a good occasion to use `?scan` and `?read.table` to remind yourself about these functions for reading data into R .

Exercise 3.6 Use a `for` loop and the `Sys.sleep()` function to make an animated particle diffusion cartoon described by the *pseudocode* below:

```
N=500 particles
x,y = coordinate vectors
      initialize from normal distribution, mean 0.5, variance (ahem) 0.0025
for 100 time steps  {
   update the coordinates by adding Normal(0,0.01) noise to x,y
   plot the points using pch=19, xlim=c(0,1), and ylim=c(0,1)
   pause for 0.05sec
}
```

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

Table 7: Some comparison operators in R . Use ?Comparison to learn more.

3.2 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 similar to the for-loop example; source it to get a graph of population sizes over time. A few things to notice about the program:

1. Although the condition in the while loop said `while(popsize<1000)` the last population value was > 1000. That's because the loop condition is checked **before** the commands in the loop are executed. When the population size was 640 in generation 6, the condition was satisfied so the commands were executed again. After that the population size is 1280, so the loop is finished and the program moves on to statements following the loop.
2. Since we don't know in advance how many iterations are needed, we couldn't create in advance a vector to hold all the results. Instead, a vector of results was constructed by starting with the initial population size and appending each new value as it is calculated. .
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.

Within a while-loop it is often helpful to have a **counter** variable that keeps track of how many times the loop has been executed. In the following code, the counter variable is `n`:

```
n=1;
while(condition) {
    commands
    n=n+1;
}
```

The result is that `n=1` is true while the `commands` (whatever they are) are being executed for the first time. Afterward `n` is set to 2, and this remains true during the second time that the commands are executed, and so on. One use of counters is to store a series of results in a vector or matrix: on the n^{th} time through the commands, put the results in the n^{th} entry of the vector, n^{th} row of the matrix, etc.

The conditions controlling a **while** loop are built up from operators that compare two variables (Table 7). 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.

Warning! `x=y` and `x==y` are two completely different things. The first is an *assignment*, in which `x` is assigned a new value. The second is a *comparison*.

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

```
> x=1:5; b=(x<=3); b
[1]  TRUE  TRUE  TRUE FALSE FALSE
```

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.

Exercise 3.7 Write a script file **Parasite2.R** that uses a while-loop to compute the number of parasites in your body and your assistant's 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. Use a copy of **Parasite1.R** as your starting point.

Exercise 3.8 Rewrite the `for` loop in the following code using a `while` loop and a counter variable j:

```
# Simulate something close to Brownian Motion
set.seed(3);
Year  = seq(1980,2009,by=0.01);
Value = cumsum(rnorm(length(Year)));
Value = Value - min(0,min(Value))+1;

# Plot all the data
par(ask = TRUE);
plot(Year, Value, type="l", main=paste("Years",1980,"to",2008));

# Plot each half-decade
for(j in seq(1980,2000,by=5)) {
   plot(Year, Value, type="l", lwd=3, xlim=c(j,j+5), main=paste("Years",j,"to",j+5));
}
```

3.2.1 Logical operators

Sometimes we need to make decisions based on several different criteria: is $x > 3$ and also $y > 5$?. For these, **logical operators** are used to make and combine multiple 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:
`>> a=c(1,2,3,4); b=c(1,1,5,5); (a<b)&(a>3); (a<b)|(a>3);`
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.

An alternative to `(x==y)` is the `identical` function. `identical(x,y)` returns TRUE if `x` and `y` are exactly the same, else FALSE. For example, if x and y are vectors, then `x==y` will return a vector of values for element-by-element comparisons, while `identical(x,y)` returns a single value: TRUE if each entry in `x` equals the corresponding entry in `y`, otherwise FALSE. You can alsu use `all(x==y)`, which takes the element-by-element vector returned by `x==y` and returns `TRUE` if all are `TRUE`, `FALSE` otherwise. You can use ?Logical to read more about logical operators.

Exercise 3.9 Use the `identical` function to construct a one-line command that returns TRUE if each entry in a vector `rnorm(5)` is positive, and otherwise returns FALSE. **Hint:** `rep` also works on logical variables, so `rep(TRUE,5)` returns the vector `(TRUE,TRUE,TRUE,TRUE,TRUE)`.

Exercise 3.10 Construct a *different* one-line command that returns TRUE if each entry in a vector `rnorm(5)` is positive, and otherwise returns FALSE. This time use the fact that R does arithmetic on logical values. (Note: the point of this and the previous exercise is to emphasize ways of *vectorizing* operations – avoiding the use of loops, which are slow, when you could express the same computation as a single operation on a vector or matrix.)

Exercise 3.11 R has two lists `letters` and `LETTERS` that contain lower case and upper case letters, respectively. The function `sample(x,n)` takes a list x and return n elements chosen randomly from that list (with or without replacement). Write a `while` loop that samples 3 `LETTERS` with replacement, and does so until this random `word` equals $c("E","E","B")$. Use a counter variable to track the number of loops, and don't forget to initialize the `word`. **Warning!** This should only take a few seconds to run! If the while loop doesn't want to stop, the `ESC` key will stop the current calculation.

Exercise 3.12 Write a `for` loop that repeates the previous `while` loop 100 times and stores the number of guesses needed to match the word. Plot a histogram of the number of guesses, and discuss what distribution these numbers should follow.

3.3 Iteration alternatives in R : flavors of `apply()`

Our goal as programmers in R is to try write code that takes advantage of what R has under the hood. It's faster, takes less code (less room for error!) and makes your code easy for others to read and understand. One way we do this is to use vectors and matrices instead of an excess of `for` and `while` loops. An additional set of R specific tools to avoid unnecessary element-by-element code, are the set of functions which include `apply()`, `lapply()`, `sapply()`, and `tapply()`. They each take at least two arguments: an object (usually a matrix or data frame) `X`, and a function which is then *applied* to the rows, colums or individual elements of X. The result is then returned as a list, matrix or data frame. See the help for these functions for details.

To illustrate their use, here are a couple of examples.

Example 1: `sapply()` vs. `for()`

```
squareplusone = function(x) { return(1 + x^2); }
X = seq(0,10,by=2);
Y = rep(NA,length(X));

# This for loop...
for (j in 1:length(X) ) {
   Y[j] = squareplusone(X[j]);
};
Y

# ... does the same thing as this sapply() command.
Y = sapply(X,squareplusone);
Y
```

Example 2: Computing column means for a matrix

```
X = matrix(rnorm(60,c(10,20,30)),20,3,byrow=T); X
Y = apply(X,2,mean); # 2nd argument: take the mean of each 1=row, 2=column.
Y
```

Exercise 3.13 Modify the `apply()` command in the Example 2 above so that it returns the row sums in a vector.

Exercise 3.14 Write a for loop that yields the same result as `Y = apply(X,1,sd);`

Exercise 3.15 Define a function `lessmean()` that takes an vector z and returns the mean of z after chopping off the decimal part of each number in z (hint: `floor()`). Modify the `apply()` command in Example 2 above so that it computes the `lessmean()` values of each column of X.

4 Branching: using `if` and `else`

Returning now to the comparison operators, we often use "rules" for model simulation or data processing that depend on the values of certain R objects. The **if/else** statements lets us do this; the basic format is

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

If `condition` is `TRUE` the first block of code is run, otherwise the second block of code gets called.

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 always just leave it out.

Exercise 4.1 Look at and source **Branch1.R** to see an `if` statement which makes the population growth rate depend on the current population size.

Exercise 4.2 Modify a copy of the **Parasite1.R** script so that there is random variation in parasite success. 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%; for your assistant the variation is ±20%. That is,

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

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. Be sure that your script does one new coin toss for each new day.

4.1 Nested if statements

More complicated decisions can be built up by nesting one `if` block within another. **Branch1.R** can be modified to use nested `if` and `else` statements 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 to 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. Then R jumps to line (11) 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.

Alternatively, nested `else-if` statements can also be written as

```
if(condition1) {
    do something
} else if(condition2) {
    do something different
} else if(condition3) {
    do something completely different
} else {
    do something else
}
```

Exercise 4.3 Modify **Branch1.R** to include the nested if statements (1)-(10) above. Once that works, then rewrite it using the alternative format using `... } else if() {...` described above.

Exercise 4.4 Use nested if's to write a script that draws one 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 $\geq 2/3$.

Exercise 4.5 Modify the script from the previous exercise and define a function `Magic8()` that takes a question (a string) as an argument and returns a random answer of "yes", "no" or "maybe."

4.2 Functions `ifelse()` and `switch()`

Sometimes you may wish to do something as simple as define one variable in a way that depends on the value of another variable. For example,

```
if(x==1) {
   y="a";
} else {
   y="b";
}
```

This can often be done using either `ifelse()` or `switch()`. The first of these two functions evalues an expression (it's first argument) and returns one of two values depending on whether or not that expression is true or false.

```
x = round(runif(1)); # x = a 0,1 coin toss
y = ifelse(x==1, "x is 1", "sorry, x wasn't 1"); y;
```

When more than a simple `TRUE/FALSE` decision determines the value to return, the `switch()` function works in a similar way (see the help for important details).

```
x = sample(c("a","b","c","d"),1); # pick a random letter
y = ifelse(x==1, "x is 1", "sorry, x wasn't 1"); y;
```

The `switch()` function will evaluate an expression followed by n arguments. It returns the argument corresponding to the number returned by the expression:

```
x = sample(1:5,1);
y = switch(x, "first","second","third","fourth","fifth"); x; y;
```

The `switch()` function also can take character values and return a corresponding value specified in the other arguments

```
x = sample(c("a","b","c","d"),1); # pick a random letter
y = switch(x, a="A", b="B", c="C", d="D"); x; y;
```

Exercise 4.6 Using a random row of the data in `FakeDiseaseData.csv` (see `?sample`) use the `switch()` function to return "Male", "Female" or "Unknown" based on the value of `Name` in that row.

Exercise 4.7 Using the data from the previous exercise, write a `for` loop that adds a third colum `Sex` to the data frame. Do this first using nested `if-else` statements, then create a second script that uses the `switch()` function.

5 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. 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 5.1 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 5.2 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 5.3 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

$$R = \frac{\sum_i P_i g_i(t_i)}{T + \sum_i P_i t_i}$$

where P_i is the fraction of type-i patches. Save this script as **TwoPatchRate.R**, we'll be using it later.

Exercise 5.4 This exercise is about using a function in a script to simulate the random population growth model

$$n(t+1) = \lambda(t)n(t), n(0) = 1. \tag{1}$$

(a) Write a function `new.pop(nt)` whose input argument `nt` is a number representing population size n in year t, and which returns the number $\lambda \times$`nt` where $\lambda = e^Z$ and Z is a random number having a normal distribution with mean=0.05, sd=0.5. Z should be generated by `rnorm` within the function, so that a new value of Z is generated each time the function is called.
(b) Write a script that uses the `new.pop` function to simulate the model (1) from time $t = 0$ to $t = 50$, and then plots $n(t)$ over time.

Exercise 5.5 (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 returned value 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 (hint: remember what `u*v` is when `u` and `v` are equal-size vectors), and that `new.pop` can use the `length` function to find out how long `nt` is).
(b) Having done this, you can now modify your script so that it simulates many populations that each follow the random growth model (1), i.e.

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

using a single loop on time t – which is much faster than a nested loop over time and over populations. Use `matplot` to display the results for $t = 0, 1, \cdots 50$ and $d = 20$ replicate populations. [Hint: set up `nt` 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: `nt[2,]=new.pop(nt[1,])`. And so on, as often as needed).

6 Solving differential equations

In addition to basic programming, R includes many functions that make it relatively simple to do some complicated things, relative to the effort required in a compiled language such C or fortran. A prime example is finding numerical solutions for a system of differential equations

$$dx/dt = f(t, x) \tag{2}$$

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 (2). For example, here is an example of the Lotka-Volterra competition equations (see the example file **SolveLotka.R**):

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

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 gives 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** package, 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),(0:200)/10,lotka,c(0.5,0.5))
```

In the line above, the values of the input arguments are specified within the call to `rk4`. When doing a function call this way, the arguments have to occur in the order that the function expects. For example, try switching the order of the 2nd and 3rd arguments:

```
out=rk4(c(2,1),lotka,(0:200)/10,c(0.5,0.5))
```

The result is an error message, because `rk4` is expecting a numeric vector as its second argument, and it gets a function name instead. If you supply the names of arguments, then they can go in any order, for example:

```
out=rk4(y=c(2,1),func=lotka,times=(0:200)/10,parms=c(0.5,0.5))
```

6.1 Always use lsoda!

The "industrial strength" solver is **lsoda**. This 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. Defaults are provided for everythying but the case-specific arguments required by **rk4**, so lsoda can be called just like 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.

One key reason for using `lsoda` rather than `rk4` is *stiffness*. Differential equations are called stiff if they have some variables or combinations of variables changing much faster than others. Stiff systems require special techniques and are harder to solve than non-stiff systems. The `lsoda` solver monitors the system

that it is solving for signs of stiffness, and automatically switches to a stiff-system solver when necessary. Many biological models are at least mildly stiff, so for real work you should *always* use `lsoda` rather than `rk4`. The only time to try `rk4` is that if `lsoda` fails on your problem (returning an error message rather than a solution matrix), you may get a clue as to the reason by trying `rk4` with a very small time step and seeing how the solutions behave (e.g., does a state variable blow up to infinity in finite time?).

Update: The latest package (i.e., the one you should use to call `lsoda`) is `deSolve`. Instead of calling `lsoda` directly, use the `ode()` function, which is a wrapper around `lsoda` and a few other methods. For more information, see `?ode` and "Solving Differential Equations in R" by Soetaert, Petzoldt and Setzer.

Exercise 6.1 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)

$$\begin{aligned} dn_1/dt &= r_1 n_1(1 - n_1/K_1) - n_2 \frac{a_1 n_1}{B + n_1} \\ dn_2/dt &= n_2 \frac{a_2 n_1}{B + n_1} - r_2 n_2 \end{aligned} \tag{4}$$

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 6.2 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:

$$\begin{aligned} dn_1/dt &= r_1 n_1(1 - n_1/K_1) - n_2 \frac{a_1 n_1^q}{B^q + n_1^q} \\ dn_2/dt &= n_2 \frac{a_2 n_1^q}{B^q + n_1^q} - r_2 n_2 \end{aligned} \tag{5}$$

(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?

6.2 The logs trick

In many biological models the state variables always must be non-negative, but a numerical ODE solver doesn't know this. If a variable decreases rapidly to near-zero values in the exact solution, a numerical approximate solution might overshoot to a negative value, leading to nonsense. For example, if the number of infectives becomes negative in an SIR-type infectious disease model, the transmission rate βSI becomes negative (here β is a positive constant, S is the number of individuals susceptible to catching the disease, and I is the number of infective individuals). Disease transmission then goes in the wrong direction: contact between susceptibles and infectives makes infected individuals become healthy, pushing I even further negative. Once a model goes down the rabbit hole, it might never come back.

This problem can often be fixed by a simple trick that is often used but rarely written down. The trick is to transform the model onto natural log scale, solve the transformed model, and back-transform the

output. This is easier than it sounds, because of the fact that for any state variable $x(t)$,

$$\frac{d(\log x(t))}{dt} = \frac{1}{x(t)}\frac{dx}{dt}. \tag{6}$$

This means that you can compute the untransformed vector field dx/dt, and then get the tranformed vector field with a single element-by-element division. For example, here is the function that you would use to solve the Lotka-Volterra model on log scale:

```
Loglotka=function(t,logx,parms) {
    x=exp(logx);
    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/x));
}
```

There are only two differences from the original `lotka`: the first line that back-transforms from `logx` to `x`, and the last line that uses equation (6) to compute the vector field of the log-transformed state vector.

Or, with a bit more thinking, you can work out what $\frac{1}{x}\frac{dx}{dt}$ is, and write that as your function, as in:

```
Loglotka2=function(t,logx,parms) {
    a=parms[1]; b=parms[2]; x=exp(logx);
    xdot=rep(0,2);
    xdot[1]=(2-x[1]-a*x[2]);
    xdot[2]=(3-x[2]-b*x[1]);
    return(list(xdot));
}
```

The logs trick is not a panacea. It can even *create* problems that weren't there originally. If a positive state variable converges to 0 in non-transformed numerical solutions, then on log scale it is diverging to $-\infty$, potentially leading to numerical overflow errors.

Exercise 6.3 Write a script file that solves the Rosenzweig-MacArthur model (4) using the logs trick with lsoda, and use it to replicate your results from that exercise.

6.3 Additional arguments in lsoda

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

rtol, 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 $rtol\,|x_i| + atol$. Small values of rtol and atol give more accurate solutions, but it takes longer to get them.

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.

7 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=100*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 "restaurant" 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 using only values of the objective function. As a safeguard, we re-optimize starting from the supposed minimum identified on the first call to optim, and check to see that we converge back to the same parameter estimates. In this case the re-optimization makes no significant adjustments to parameters, so we're done and can plot the results (Figure 1) – see **FitThetaLogist.R** for the complete code.

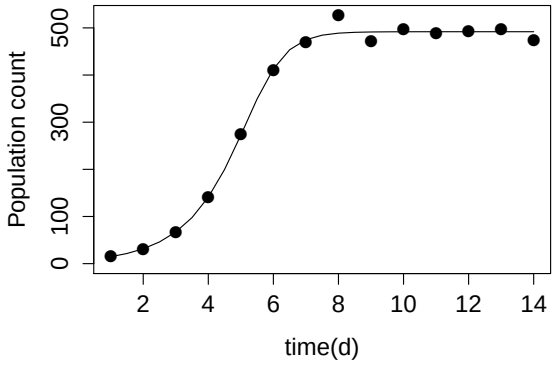

Figure 1: Results from least-squares fitting of the θ-logistic model. Points are the data (from Veilleux 1976,1979), curve is the model solution with best-fitting parameters

If simple optimization doesn't seem to be working – as evidenced by the finding that repeated calls to `optim` keep on giving different results – it may help to vary some of the default parameters in `optim`.

Exercise 7.1 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 7.2 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 7.3 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 (1). 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.

7.1 What to optimize?

This is not an easy question, because it depends on what you believe about the real system. In the 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 and that all differences between model output and the data are due to measurement errors. That is, we assume 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.

The R function `mle`, in the `stats4` package, can be used to fit models by maximum likelihood. The script **FitThetaLogist.R** includes an example. The objective function for `mle` must be the negative log likelihood of the data under the assumed model. The fitted model then includes estimated standard errors for the parameter estimates, based on large-sample theory for maximum likelihood estimates. Unless you are sure that your data set is *really large*, the estimated standard errors should be viewed as rough approximations.

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 a topic in statistics so we will mention only one important case. If the errors have a Poisson distribution, 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, for fitting with `optim` we would change the objective function `TLfit` in **FitThetaLogist.R** by redefining

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

For fitting with `mle` the `minuslogl` function is modified by setting

```
e2=sqrt(out[,2])-sqrt(xvals)
```

Exercise 7.4 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

$$X_{t+1} = F(X_t) + \sigma Z_t \tag{7}$$

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 (7) is a nonlinear regression model, which can be fitted by any modern statistics package using nonlinear least squares (in R this is function **nls**). 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 7.5 Here again is the Rosenzweig-MacArthur model

$$\begin{aligned} dn_1/dt &= r_1 n_1(1 - n_1/K_1) - n_2 \frac{a_1 n_1}{B + n_1} \\ dn_2/dt &= n_2 \frac{a_2 n_1}{B + n_1} - r_2 n_2 \end{aligned} \tag{8}$$

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**).

7.2 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=4,parscale=p0));
```

Here we have:

- changed the optimization method to BFGS,
- set the maximum number of search steps to 1000,
- set the level of tracing to 4 (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),
- told `optim` to scale parameters relative to the initial guess.

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. It is often helpful to use the `parscale` option. Internally `optim` works with parameters scaled relative to the corresponding value in `parscale`, which by default is a vector of 1's. If you are using units are such that the elements of the optimal p are very different in magnitude – such as r and K in the last exercise – you can eliminate this problem by setting `parscale` to be your current best guess for the optimal p.

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 7.6 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 function:

```
g1=function(t1) {2*t1^0.5}
g2=function(t2) {t2^0.7}
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.

7.3 How to optimize

This discussion is based on Kelley (1999), the optimization chapter in Numerical Recipes (available at `www.library.cornell.edu/nr` and `www.nr.com`), and much painful experience fitting models to data.

The basic decision in optimization is how you want to get from your initial guess to a final value: a few clever steps in parameter space, or a lot of simple steps. Being "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 identify a good direction in which to search for a local minimum. For example, the first and second derivatives at the current evaluation point define a quadratic approximation to the objective function. But rather than jumping directly to the minimum of the quadratic approximation, the algorithm does a one-dimensional search along the line connecting the current evaluation point and the minimum of the quadratic approximation. The point on that line where the function value is smallest (as approximated by a one-dimensional minimization algorithm) is then used as the next evaluation point. Methods of this type 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, and build up an approximate second derivative as the function and its derivative are repeatedly evaluated.

Cleverness is often counterproductive, because 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 is often more than outweighed by the need to run many repetitions.

Less clever method often fare better against multiple local minima. They work by gradually narrowing in on a minimum, so initially they don't pay much attention to the objective function's fine-scale structure (Kelley 1999). The most popular method of this type is 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, by searching along the line between the worst point and the average of all the others. If that fails, it may shrink the simplex down around the best point on the current simplex.

However there really is no cure for the problem of multiple local minima. To be confident of your results you need to run the optimization from multiple starting points and make sure that you repeatedly find the same "best" minimum, and always re-start the optimization from the supposed minimum to make sure that you converge back to it.

8 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: 299 314.

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

Maindonald, J.H. 2004. Using R for Data Analysis and Graphics: Introduction, Code, and Commentary. URL `http://www.cran.R-project.org`.

Press, W.H., S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. Numerical Recipes: the Art of Scientific Computing, 2nd edition. Cambridge University Press (complete editions for C, f77, and f90 are available online at `www.nr.com`).

R Development Core Team. 2004. R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL `http://www.R-project.org`.

Verzani, J. 2002. simpleR – using R for Introductory Statistics. URL `http://www.cran.R-project.org`.

Veilleux, B. G. 1976. The analysis of a predatory interaction between *Didinium* and *Paramecium*. Master's thesis, University of Alberta.

Veilleux, B. G. 1979. An analysis of the predatory interaction between *Paramecium* and *Didinium* Journal of Animal Ecology 4: 787-803.