An introduction to R for dynamic models in biology

Last compile: August 24, 2016

Stephen P. Ellner[1] and John Guckenheimer[2]
[1]Department of Ecology and Evolutionary Biology, and
[2]Department of Mathematics
Cornell University, Ithaca NY 14853

Contents

1 Interactive calculations **3**

2 An interactive session: fitting a linear regression model **5**

3 Vectors **7**

4 Matrices **10**

4.1 cbind and rbind 11

4.2 Matrix addressing 12

4.3 Matrix operations and matrix-vector multiplication 13

5 Iteration (“Looping”) **13**

5.1 For-loops 13

5.2 While-loops [advanced] 16

6 Branching **17**

7 Numerical Matrix Algebra **18**

7.1 Eigenvalues and eigenvectors 19

7.2 Eigenvalue sensitivities and elasticities 20

7.3 Finding the eigenvalue with largest real part [advanced] 22

8 Creating new functions **22**

9 Solving systems of differential equations **23**

9.1 Always use lsoda! 26

10 References **27**

Preface

Disclaimer: The following notes are a subset of the original laboratory manual authored by S.P Ellner and J. Guckenheimer.

These notes for computer labs accompany our textbook *Dynamic Models in Biology* (Princeton University Press 2006), but they can also be used as a standalone introduction to R as a scripting language for simulating dynamic models of biological systems. They are based in part on course materials by former TAs Colleen Webb, Jonathan Rowell and Daniel Fink (then at Cornell University), 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). We also have drawn on the documentation supplied with R (R Development Core Team 2005).

The current home for these notes is `https://people.cam.cornell.edu/~dmb/DMBsupplements.html`, a web page for the textbook that we maintain ourselves. If that fails, an up-to-date link should be in the book's listing at the publisher (`www.pupress.princeton.edu`). Parallel notes and script files for MATLAB are also available at those sites.

Sections 1-7 are an autotutorial introduction to basic R programming (we generally cover them in two or three 2-hour lab sessions, depending on how much previous experience students have had). Those sections contain many sample calculations. It is important to do them yourselves – *type them in at your keyboard and see what happens on your screen* – to get the feel of working in R . All exercises in the middle of a section should be done *immediately* when you get to them, and make sure that you have them right before moving on. Exercises at the ends of sections may be more appropriate as homework exercises.

What is R ?

R is an object-oriented scripting language that combines

- the programming language S developed by John Chambers (Chambers and Hastie 1988, Chambers 1998, Venables and Ripley 2000).
- 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 Development Core Team 2005) available free via the Web (see below). Originally a research project in statistical computing (Ihaka and Gentlemen 1996) it is now managed by a team that includes a number of 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 R provides better tools for 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, rather than the more advanced grid and lattice graphics packages.

The main sources for R are CRAN (`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

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 1: 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.

for LINUX or UNIX is more complicated and idiosyncratic (which will not bother the corresponding users), but many recent LINUX distributions include a fairly up-to-date version of R .

For Windows PCs we strongly suggest 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, and selects the most powerful version of the help system.

This document was written at a Windows PC and may sometimes refer to Windows-specific aspects of R . We will be happy to make changes as these lapses are brought to our attention.

Statistics in R

Some of the important functions and libraries (collections of functions) for data analysis and statistical modeling are summarized in Table 1. The book by Venables and Ripley (2002) gives a good practical overview, and a list of available libraries and their contents is available at CRAN (`www.cran.r-project.org`, click on `Package sources`). Maindonald (2004) and Verzani (2002) – both available online – present the basics of statistical data analysis and graphics in R . For the most part, we are not concerned here with this side of R .

1 Interactive calculations

Launching R 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!) **do not** use the underscore character (_) or blank space as a separator in variable names. Recent versions of R have been progressively removing limitations like this, but for compatibility with older versions of R and other implementations of S it is best to not use underscores and blanks. R is case sensitive: `Abc` and `abc` are **not** the same variable.

Exercise 1.1 Here are some variable names that cannot be used in R ; explain why: `cell maximum size; 4min; site#7 .`

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
```

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
```

Table 2: 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.

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

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 2).

An essential R skill is *using the help system*. If you know that `plot` is a function that plots things, but you don't know exactly how to make the kind of plot you want to make just now, type `?plot` at the R command prompt. If `plot` doesn't do what you want, try `??plot)` to get a list of many other functions that have "plot" as part of their name - maybe one of them will do the trick for you. Once you get used to using it regularly, and get used to the way help pages are structured, the R Help system is in fact enormously helpful. You should also explore the items available on the Help menu, which include the manuals, FAQs, and ways to search Help files (like ??) and function names ('Apropos' on the menu).

Exercise 1.2 : 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 1.3 : Do an Apropos on `sin` via the Help menu, to see what it does. Next do

```
??sin)
```

and see what that does (answer: ??sin pulls up all help pages that include 'sin' anywhere in their title or text. Apropos just searches function names for 'sin'.)

Exercise 1.4 Use the Help system to find out what the `hist` function does – most easily, by typing `?hist` at the command prompt. Prove that you have succeeded by doing the following: use the command `y=rnorm(5000)` to generate a vector of 5000 random numbers with a Normal distribution, and then use `hist` to plot a histogram of the values in y with about 20 bins. Why did we say "about 20" rather than "exactly 20"?

2 An interactive session: fitting a linear regression model

To get a feel for working in R we'll fit a straight line model (linear regression) to some data. Below are data on the maximum per-capita 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. 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 object created by `lm`:

```
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
```

Adding the regression line to the plot of the data is similarly accomplished by a function taking `fit` as its input.

```
> abline(fit)
```

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

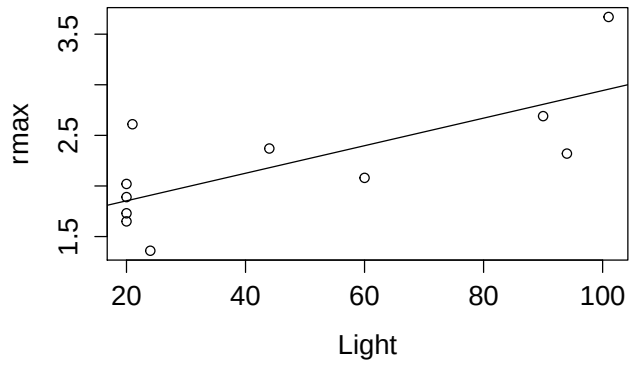

Figure 1: Graphical summary of linear regression analysis. The commands for this plot are in `Intro1.R`. Data are from the studies described in Fussmann et al. (2000). Science 290: 1358-1360.

```
 [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
```

3 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()`, for example:
`initialsize=read.table("c:\\temp\\initialdata.txt")`
(Note: if the file you're tying to load doesn't exist, this is not going to work!).

Once it has been created, a vector can 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.

Functions for vector construction

Some of the main functions for creating and working with vectors are listed in Table 3. A set of regularly spaced values can be constructed with the `seq` function, whose syntax is

`seq(from,to,by)` or `seq(from,to,length)`

The first form generates a vector `(from,from+by,from+2*by,...)` with the last entry not being larger than `to`. If a value for `by` is not specified, its value is assumed to be 1 or -1, depending on whether `from` or `to` is larger. The second generates a vector of `length` evenly-spaced values, running from `from` to `to`, for example

```
> seq(1,3,length=6)
[1] 1.0 1.4 1.8 2.2 2.6 3.0
```

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
```

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` 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.

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, `x=rnorm(100)` generates 100 random numbers with a Normal (Gaussian) distribution having mean=0, standard deviation=1. But `rnorm(100,2,5)` yields 100 random numbers from a Gaussian distribution with mean=2, standard deviation=5.

Here, and in the R documentation and help pages, the existence of default values for some arguments of a function is indicated by writing (for example) **rnorm(n, mean=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 vectors of random numbers are listed in Table 4. Functions to evaluate the corresponding distribution functions are also available. For a listing use the Help system: ?Normal, ?Uniform, ?Lognormal, etc. will give lists of the available functions for each distribution family.

Exercise 3.1 Create a vector `v=(1 5 9 13)` using `seq`. Create a vector going from 1 to 5 in increments of 0.2 first by using `seq`, and then by using a command of the form `v=1+b*c(i:j)`.

Exercise 3.2 Generate a vector of 5000 random numbers from a Gaussian distribution with mean=3, standard deviation=2. Use the functions `mean, sd` to compute the sample mean and standard deviation of the values in the vector, and `hist` to visualize the distribution.

Exercise 3.3 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 $[r^0, r^1, r^2, \ldots, r^n]$ and computes the sum of all its elements. Compare the sum of this vector to the limiting value $1/(1 - r)$. Repeat this for $n = 50$.

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

`seq(from,to,by=1)`	Vector of evenly spaced values with specified increment (default = 1)
`seq(from,to,length)`	Vector of evenly spaced values with specified length
`c(u,v,...)`	Combine a set of numbers and/or vectors into a single vector
`rep(a,b)`	Create vector by repeating elements of `a` by amounts in `b`
`hist(v)`	Histogram plot of value in `v`
`mean(v),var(v),sd(v)`	Population mean, variance, standard deviation estimated from 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. Note that statistical functions such as `var` regard the values as samples from a population (rather than a list of value for the entire population) and compute an estimate of the population statistic; for example `sd(1:3)=1`.

`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 `size`=#trials N, `prob`=probability of success p
`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 n random numbers. To create random matrices, these vectors can be reshaped using the `matrix()` function, for example: `matrix(rnorm(50*20),50,20)` generates a 50×20 matrix of Gaussian(0,1) random numbers.

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

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

Exercise 3.4 write a **one-line** command to extract a vector consisting of the second, first, and third elements of `q` **in that order**.

Exercise 3.5 Write a script file that computes values of $z = \frac{(x-1)}{(x+1)}$ and $w = \frac{sin(x^2)}{x^2}$ for $x = 1, 2, 3, \cdots, 12$ and plots both of these as a function of x with the points connected by a line.

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. The fact that R wants you to succeed is both good and bad – good when it saves you needless worry about details, bad when it masks an error that you would rather know about.

4 Matrices

A matrix is a two-dimensional array of numbers. Like vectors, matrices can be created by reading in values from a data file, using the `read.table` function. 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 `matrix` function. 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

`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
`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
`apply(A,MARGIN,FUN)`	apply the function `FUN` to each row of A (if MARGIN=1) or each column of A (if MARGIN=2). See ?apply for details and examples.

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.

```
> 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 4.1 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 4.2 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, one can use the `data.entry` function. This function can only edit existing matrices, but for example

```
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.

4.1 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:

```
> A=cbind(1:3,4:6,7:9); A
```

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

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,

```
>  B=rbind(1:3,4:6); B
     [,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 4.3 Verify that `rbind(A,B)` works, `cbind(A,A)` works, but `cbind(A,B)` doesn't. Why not?

4.2 Matrix addressing

Matrix addressing works 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 8, which is the (2^{nd} row, 3^{rd} column) entry of the matrix **A** , and

```
> A[2,2:3];
[1] 5 8
> B=A[2:3,1:2]; B
     [,1] [,2]
[1,]    2    5
[2,]    3    6
```

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

```
> first.row=A[1,]; first.row
[1] 1 4 7
> second.column=A[,2]; second.column;
[1] 4 5 6
```

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

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

```
> A[1,]=runif(3); A
        [,1]     [,2]       [,3]
[1,] 0.985304 0.743916 0.00378729
```

```
[2,] 2.000000 5.000000 8.00000000
[3,] 3.000000 6.000000 9.00000000
```

Exercise 4.4 Use **runif** to construct a 5×5 matrix $\mathbf{B}$ of random numbers with a uniform distribution between 0 and 1. (a) 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). (b) Use `seq` to replace the values in the first row of $\mathbf{B}$ by `2 5 8 11 14`.

4.3 Matrix operations and matrix-vector multiplication

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)`. Matrix × matrix and matrix × vector multiplication (when they are of compatible dimensions) is indicated by the special notation `%*%`. Remember, **element-by-element is the default in R**. This requires some attention, because R 's eagerness to make things work can sometimes let errors get by without warning. So for example

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

A is a 2×2 matrix, and v is a vector of size 2, so the matrix-vector product Av is legitimate. However, Av should be a vector, not a matrix. Since you (incorrectly) "asked" for element-by-element multiplication, that's what R did, "recycling" through the elements of v when it ran out of entries in v before it ran out of entries in A. What you should have done is

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

5 Iteration ("Looping")

5.1 For-loops

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.
- Updating the state of every neuron in a model network based on the inputs it received in the last time interval.
- Simulating a biochemical reaction network 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. These are written as

```
for (var in seq) {
      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") 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 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 get help**. Loops are essential from here on out.

Exercise 5.1 : 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, j days after your return to Ithaca your parasite load is $n(j) = 400(1.1)^j$ and the number in your assistant is $m(j) = 120(1.2)^j$.

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 30 days).

Exercise 5.2 : 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?

$$\begin{matrix} 0 & 1 & 2 & 3 & 4 \\ 0.1 & 0 & 0 & 0 & 0 \\ 0 & 0.2 & 0 & 0 & 0 \\ 0 & 0 & 0.3 & 0 & 0 \\ 0 & 0 & 0 & 0.4 & 0 \end{matrix}$$

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.

5.2 While-loops [advanced]

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 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.

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 5.3 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.

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:
`>> a=c(1,2,3,4); b=c(1,1,5,5); (a<b)&(a>3); (a<b)|(a>3);`

An alternative to `(x==y)` is the `identical` function. `identical(x,y)` returns TRUE if x and y are exactly the same, else FALSE. The difference between these is that if (for example) x and y are vectors (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 use ?Logical to read more about logical operators.

Exercise 5.4 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` works on logical variables, so `rep(TRUE,5)` returns the vector `(TRUE,TRUE,TRUE,TRUE,TRUE)`.

6 Branching

Logical conditions also allow the rules for "what happens next" in a model to 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 6.1 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.

More complicated decisions can be built up by nesting one `if` block within another, i.e. the "other commands" under `else` can include a second `if` block. **Branch2.R** uses this method 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 6.2 Modify **Parasite1.m** so that there is random variation in parasite success, depending on whether or not conditions on a given day are stressful. 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,

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

Do this by using `runif(1)` and an `if` statement to "toss a coin" each day: if the random value produced by `unif` for that day is < 0.5 it's a good day, and otherwise it's bad. Make sure that your script does a new "coin toss" for each day, but that the same toss applies to both you and your assistant.

7 Numerical Matrix Algebra

R has functions for matrix-algebra calculations that use "industry standard" numerical libraries. At this writing R is completing a transition from older (LINPACK, EISPACK) to newer (BLAS, LAPACK) libraries. Some

`iden(n)`	$n \times n$ identity matrix (in `boot` library)
`zero(n,m)`	$n \times m$ matrix of zeros (in `boot` library)
`outer(v,w)`	outer product of vectors v and w; very useful for avoiding loops
`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 as a vector. See ?apply for details and examples. Very useful for avoiding loops
`sapply(A,FUN)`	apply a function `FUN` to each element in matrix A, returning a *vector* of values.

Table 7: Some important functions for creating and working with matrices in R . Many of these have additional optional arguments; use the Help system for full details.

functions exist in two versions corresponding to these, with the default generally being the newer libraries. Some of these functions are listed in Table 7.

Some of R 's matrix functions only work on square matrices and will return an error if **A** is not square, in particular functions for computing eigenvalues and eivenvectors. *For the remainder of this section we only consider square matrices*.

7.1 Eigenvalues and eigenvectors

Because eigenvalues are so important for studying dynamic models, we will now study `eigen` in some detail. 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.

We are often most interested in the dominant eigenvalue, which depending on context (discrete versus continuous time models) means either the one with the largest absolute value, or the one with the largest real part.[1] In R , the `abs` function computes the absolute value of a real or complex number (and acts element-by-element on vectors and matrices).

`eigen` returns eigenvalues sorted by absolute value, with the largest first. For example,

```
A=matrix(1:9,3,3); vA=eigen(A); vA;
$values
[1]  1.611684e+01 -1.116844e+00 -4.054215e-16

$vectors
           [,1]       [,2]       [,3]
[1,] -0.4645473 -0.8829060  0.4082483
[2,] -0.5707955 -0.2395204 -0.8164966
[3,] -0.6770438  0.4038651  0.4082483
```

As with the output from `lm` in our first interactive session (you remember `lm ...`) vA is a compound *object* composed of two *components*, whose names are `values` and `vectors`. The "$" is used to extract components

[1] The general definition of absolute value, which covers both real and complex numbers, is $|a + ib| = \sqrt{a^2 + b^2}$, where $i = \sqrt{-1}$.

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 -4.054215e-16
```

As noted above, `va$values[1]` is the eigenvalue with the largest absolute value.

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
```

This calculation verifies the definition of eigenvalues and eigenvectors: $Av_1 = \lambda_1 v_1$, so $A\vec{v}_1 - \lambda_1\vec{v}_1 = \vec{0}$. The [2,] and [3,] values above are really zeros, but because eigenvalues and eigenvectorsare computed numerically, there is a small amount of numerical error.

Exercise 7.1 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 7.2 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, we use the fact that the left eigenvectors of a matrix A are the eigenvectors of its transpose, `t(A)`. So

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

gets you the left eigenvectors of A.

Exercise 7.3 Compute `vLA[,j]%*%A-vA$values[j]*vLA[,j]` to verify the last claim about left eigenvectors, for j=1 to 3.

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 7.4 : Write a script file which applies 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.

7.2 Eigenvalue sensitivities and elasticities

For an $n \times n$ matrix **A** with entries a_{ij}, the sensitivities $s_{ij}$and elasticities e_{ij}can be computed as

$$s_{ij} = \frac{\partial \lambda}{\partial a_{ij}} = \frac{v_i w_j}{\langle v, w \rangle} \qquad e_{ij} = \frac{a_{ij}}{\lambda} s_{ij} \tag{1}$$

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` 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.

Exercise 7.5 Construct the transition matrix **A**, and then find λ, **v**, **w** for an age-structured model with the following survival and fecundity parameters. Then use those to construct the elasticity matrix for **A**.

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: $\lambda = 1.0419$

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

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

7.3 Finding the eigenvalue with largest real part [advanced]

For the Jacobian matrix of a differential equation model, the dominant eigenvalue is the one with the largest real part. To find this, and the associated eigenvector, we need to extract the real parts of the eigenvalues and locate the largest one.

Use the help system – (`?complex`) – to see the R functions for working with complex numbers. The one we need now 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 which eigenvalue is dominant (i.e., which has the largest real part). Run the code below, and follow what it's doing; see `?which` and `?max` if you need to.

```
vA=eigen(A)$values; # find the eigenvalues
rmax=max(Re(vA));   # find the largest Real part of any eigenvalue
j=which(Re(vA)==rmax); # which eigenvalue is dominant?
lmax=vA[j]; # dominant eigenvalue
vmax=eigen(A)$vectors[,j];  # dominant eigenvector
```

Exercise 7.6 Try the above on `A=diag(c(1,2,3))` and see what you get for `j,r,lmax` and `vmax`. Try it again for `A=diag(c(1,2,sqrt(as.complex(-9))))` and `A=diag(c(3,1,3))`, and make sure that you understand what happens in each case.

8 Creating new functions

Functions (often called subroutines in other computing languages) allow you to break a program into subunits. This makes complex problems easier to program, and helps us (and others) to understand the logical flow of programs. Each function is an independent little program, performing one task or a few related tasks, and returning the results. A program can then be written to call on various functions to perform different tasks. Each function can be written and tested independently, which leads naturally to the generally recommended modular style of program construction.

The basic syntax for creating a function is as follows. Suppose [for the sake of an easy 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 for doing that is as follows:

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

This code effectively adds `mysquare` as a new 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`.

Exercise 8.1 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,

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

Functions can be placed anywhere in a script file. Once the code for a function has been executed within a session, the new function can be treated like any other R command.

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
```

Exercise 8.2 Write a script that defines a function `DominantRealPart`, which takes as input a single matrix, and returns the largest real part of any of its eigenvalues.

Exercise 8.3 [advanced] Write a script that defines a function `domeig` which takes as input a single matrix, and returns a list with components `value` and `vector`, which are respectively the eigenvalue with largest absolute value, and the corresponding eigenvector scaled so that the absolute value of its entries sum to 1. (If `v` is a vector, then `v/sum(abs(v))` will have the property that the absolute values of its entries sum to 1).

Exercise 8.4 [advanced] Modify your function from the last exercise so that it has a third, optional argument j with default value 1. Call the function `oneeig`. If no value of j is specified, then `oneeig` should act just like `domeig`. If a value of j is specified, it should return the j^{th} largest eigenvalue (in absolute value), and the corresponding eigenvector scaled as in the last exercise.

9 Solving systems of differential equations

Built-in R functions make it relatively easy to do some complicated things. One important example is finding numerical solutions for a system of differential equations

$$\frac{dx}{dt} = f(t, x).$$

Here x is a vector assembled from quantities that change with time, and the *vector field* f gives their rates of change. The Hodgkin-Huxley model from the last section is one example. Here we start with the simple model

of a gene regulation network from Gardner et al. (2000) that is described in the textbook. The model is

$$\begin{aligned} \frac{du}{dt} &= -u + \frac{\alpha_u}{1+v^\beta} \\ \frac{dv}{dt} &= -v + \frac{\alpha_v}{1+u^\gamma} \end{aligned} \tag{2}$$

The variables u, v in this system are functions of time. They represent the concentrations of two repressor proteins P_u, P_v in bacteria that have been infected with a plasmid containing genes that code for P_u and P_v. The plasmid also contains promoters, with P_u a repressor of the promoter of the gene coding for P_v and vice-versa.

The equations are a simple compartment model describing the rates at which u and v change with time. P_u degrades at rate 1 (so the loss rate is $-1 \times u$) and it is produced at a rate $\frac{\alpha_u}{1+v^\beta}$, which is a decreasing function of v. The exponent β models the *cooperativity* in the repression of P_u synthesis by P_v. The processes of degradation and synthesis combine to give the equation for $\frac{du}{dt}$, and the equation for $\frac{dv}{dt}$ is similar.

There are no explicit formulas to solve this pair of equations, but we can interpret what the equations mean geometrically. At each point of the (u, v) plane, we regard $(\frac{du}{dt}, \frac{dv}{dt})$ as a **vector** that gives the direction and magnitude for how fast (u, v) jointly change as a function of t. Solutions to the equations give rise to parametric curves $(u(t), v(t))$ whose tangent vectors $(\frac{du}{dt}, \frac{dv}{dt})$ are those specified by the equations.

To plot the vector field, first run the script DMBpplane.R (this script is based on pplane.R by Daniel Kaplan, Department of Mathematics, Macalester College, and has been modified and used here with his permission). This script includes a function that computes the vector field for model (2) with the assumption that $\alpha_u = \alpha_v$:

```
toggle=function(u,v,parms) {
    du= -u + parms[1]/(1+v^parms[2]);
    dv= -v + parms[1]/(1+u^parms[3]);
    return(c(du,dv));
}
```

Note how this function is set up:

- Its input arguments are the two state variables `u,v` and a parameter vector `parms`. Even if `parms` is not used by the function, it must be included in the list of arguments (you can set `parms=0`).
- The function returns the vector field *as a vector*.
- The computations are set up so they will go through if `u` and `v` are matrices of equal size. This eliminates the needs for for-loops in many of the computations that we will use to study vector fields.

Use this setup for any vector field that you want to study using the functions in DMBpplane.R.

The function `phasearrows` (also in DMBpplane.R) can then be used to plot the vector field, as in this example:

```
phasearrows(fun=toggle,xlim=c(0,3),ylim=c(0,3),resol=25,parms=c(3,2,2))
```

In this statement, `fun` is the name of the function that calculates the vector field, `xlim` and `ylim` define the range of values for the plot, and setting `resol=25` means that arrows showing the vector field will be drawn at a 25×25 grid of points within the plotting region. The parameter vector `parms` is passed to the vector field function – this argument can be omitted if the vector field function does not use `parms`.

Exercise 9.1 Run DMBpplane.R to create the `toggle` function, and then use the command above to plot the vector field.

We can think of solutions to (2) as curves in the plane that "follow the arrows". Given a starting point (u_0, v_0), the mathematical theory discussed in the textbook proves that there is a unique solution $(u(t), v(t))$ with $(u(0), v(0)) = (u_0, v_0)$. The process of finding the solution numerically is called *numerical integration*. Methods for numerical integration build up an approximate solution by adding short segments of an approximate solution curve, one after another.

R 's numerical ODE solvers are in the **deSolve** package. Download this (if necessary) from CRAN, and then use the command `library(deSolve)` to load it into your R session.

Exercise 9.2 After you load the deSolve library, use `?rk4` to look at the syntax for this function.

To use R 's ODE solvers, the first step is to write a function to evaluate the vector field *in the specific format required by the solvers*. Here's what that looks like for the toggle switch model:

```
Toggle=function(t,y,parms) {
    u=y[1]; v=y[2];
    du= -u + parms[1]/(1+v^parms[2]);
    dv= -v + parms[1]/(1+u^parms[3]);
    dY=c(du,dv);
    return(list(dY));
}
```

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

1. The function must have input arguments `(t,y,parms)`, even if only the state vector `y` is used to compute the vector field. Here `t` is time, `y` is the state vector, and
2. `parms` is again a vector of parameter values for the function. This allows you to see how solutions change as parameters are varied, without having to re-write and re-run the function. It also simplifies the process of fitting differential equations to data.
3. The vector field value has to be returned as a list, which is why we need both *toggle* and *Toggle*. 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 "basic" ODE solver in R is **rk4**, which implements the 4th-order Runge Kutta method with a fixed time step. The format is

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

Here `times` is a vector of the times at which you want solution values, `x0` is the value of the state vector at the initial time (i.e. `times[1]`), `func` func is the function specifying the model (such as `Toggle`), and `parms` is the vector of parameter values that will be passed to func.

Here is an example using our `Toggle` function:

```
x0=c(.2,.1); times=seq(0,50,by=0.2); parms=c(3,2,2);
out=rk4(x0,times,Toggle,parms)
matplot(out[,1],out[,2:3],type="l",ylim=c(0,3),xlab="time t",ylab="u,v");
```

Exercise 9.3 Write a script toggle.R with the commands above, and run them to see some solution trajectories as a function of time. Which plotted curve is u, and which is v? How do you know? Look at the matrix `out` to see how it is set up: the first column is a list of times, and the other columns are the computed (approximate) values of the state variables at each time.

The state trajectories both seem to be approaching constants. A second way of looking at the trajectories (**Exercise:** do this yourself right now) is:

```
plot(out[,2],out[,3],type="l",xlab="u",ylab="v")
```

This kind of plot is called a *phase portrait*. It shows the path in the (u, v) *phase plane* taken by the trajectory, but we lose track of the times at which the trajectory passes through each point on this path.

Exercise 9.4 Use `rk4` again with initial conditions $(0.2, 0.3)$ to produce a new output matrix `out2` and plot phase portraits of both solutions. Do this first for time interval $[0, 50]$ and again for $[0, 200]$.

The trajectories appear to end at the same places, indicating that they didn't go anywhere after $t = 50$. We can explain this by observing that the differential equations vanish at these endpoints. The curves where $\frac{du}{dt} = 0$ and $\frac{dv}{dt} = 0$ are called *nullclines* for the vector field. They intersect at *equilibrium points*, where both $\frac{du}{dt} = 0$ and $\frac{dv}{dt} = 0$. The solution with initial point an equilibrium is constant. Here, the equilibrium points are *(asymptotically) stable*, meaning that trajectories close to the equilibria approach them as t increases.

Nullclines for a general vector field can be plotted using the function `nullclines` in DMBpplane.R. The syntax is the same as `phasearrows`, for example:

```
nullclines(fun=toggle,xlim=c(0,3),ylim=c(0,3),resol=250,parms=c(3,2,2))
```

It is a good idea to use a large value of `resol` so that the nullclines are found accurately.

Exercise 9.5 Without erasing the nullclines, add to that plot (using `points`) the two solution trajectories with different initial conditions that you have computed (`out` and `out2` from previous exercises). You should see that each trajectory converges to an equilibrium point where the nullclines intersect.

Exercise 9.6 There is a third equilibrium point where the two nullclines intersect, in addition to the two at the ends of the trajectories that you have computed. Experiment with different initial conditions, to see if you can find any trajectories that converge onto this third equilibrium (Hint: what happens in this model if $u(0) = v(0)$ for these parameter values?)

Exercise 9.7 Change the value of α from 3 to 1.5. How does the phase portrait change? Plot the nullclines to help answer this question.

9.1 Always use lsoda!

The "industrial strength" solver in R 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 in the `deSolve` package is

```
lsoda(y, times, func, parms, rtol = 1e-6, atol = 1e-6,
  jacfunc = NULL, jactype = "fullint", verbose = FALSE,
  tcrit = NULL, hmin = 0, hmax = NULL, hini = 0, ynames = TRUE,
  maxordn = 12, maxords = 5, bandup = NULL, banddown = NULL,
  maxsteps = 5000, dllname = NULL, initfunc = dllname,
  initpar = parms, rpar = NULL, ipar = NULL, nout = 0,
  outnames = NULL, forcings = NULL, initforc = NULL,
  fcontrol = NULL, events = NULL, lags = NULL,...)
```

Don't panic. Sensible defaults are provided for everything but the arguments required by `rk4`, so `lsoda` can be called just like `rk4`:

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

Usually you can get away with doing this, and here we always will. The options `rtol` and `atol` can be used to control the accuracy that the numerical integration tries to achieve, by using smaller time steps. `lsoda` automatically adjusts step sizes to achieve the desired error tolerance (based on error estimates that it calculates), whereas `rk4` always goes directly from one value in `times` to the next.

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 are harder to solve than non-stiff systems and require special techniques. 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 when `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?

Exercise 9.8 Write a script using `lsoda` to solve the Lotka-Volterra model

$$dx_1/dt = x_1(r_1 - x_1 - ax_2)$$
$$dx_2/dt = x_2(r_2 - x_2 - bx_1)$$

in which the parameters r_1, r_2, a, b are all passed as parameters via the argument `parms`. Generate solutions for the same parameter values with `rk4` and `lsoda`, and compare the results.

Exercise 9.9 Write a script using `lsoda` to solve the constant population size SIR model with births,

$$dS/dt = \mu N - \beta SI/N - \mu S$$
$$dI/dt = \beta SI/N - (\gamma + \mu)I$$
$$dR/dt = \gamma I - \mu R$$

Set parameter values as $\mu = 1/70$, $\gamma = 0.75$ with initial conditions $S(0) = 999900$, $I(0) = 100$, and $R(0) = 0$ (such that $N = 10^6$). Obtain numerical solutions, with time values t such that $t \in [0, 25]$, while varying the value of β: $\beta = 0.50$, $\beta = 2.50$.

10 References

Chambers, J.M. and T.J. Hastie. 1992. Statistical Models in S. Chapman and Hall, London.

Chambers, J.M. 1998. Programming with Data. Springer, New York.

Ermentrout, B. 2002. Simulating, Analyzing, and Animating Dynamical Systems: A Guide to XPPAUT for Researchers and Students. SIAM (Society for Industrial and Applied Mathematics), Philadelphia.

Fussmann, G., 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.

Gardner, T.S., C.R. Cantor and J.J. Collins. 2000. Construction of a genetic toggle switch in *Escherichia coli*. Nature 403: 339-342.

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.

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

R Development Core Team. 2010. 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`.

Trefethen, L. 2000. Spectral Methods in MATLAB. SIAM (Society for Industrial and Applied Mathematics), Philadelphia.

Venables, W.N. and B.D. Ripley. 2000. S Programming. Springer, New York.

Venables, W.N. and B.D. Ripley. 2002. Modern Applied Statistics with S (4th edition). Springer, New York.

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

Winfree, A. 1991. Varieties of spiral wave behavior: an experimentalist's approach to the theory of excitable media. Chaos 1: 303-334.