An introduction to R for dynamic models in biology

Last compile: January 7, 2009

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

Preface

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 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 `www.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 a general introduction to some basics of R programming. We generally cover those in two or three 2-hour lab sessions, depending on how much previous experience students have had. Rather than lecturing, we have students work through them individually, asking for help as needed and having us or a TA check their exercise solutions. 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. However the exercises are all straightforward applications of the programming techniques being taught.

The subsequent sections are linked to our textbook in fairly obvious ways. For example, section 8 on matrix computations goes with Chapter 2 on matrix models for structured populations, and section 15 on phase-plane analysis of the Morris-Lecar model accompanies the corresponding section in Chapter 5. Some exercises in these sections are designed as 'warmups' for exercises in the textbook, such as simple

examples of agent-based (discrete-event) models as a warmup for agent-based simulation of infectious disease dynamics.

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 S-plus lags far behind in providing 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 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 .

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.

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
```

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). 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 1.2: Have R compute the values of

1. $\frac{2^7}{2^7-1}$and compare it with $\left(1-\frac{1}{2^7}\right)^{-1}$

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 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.

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. Now enter the command

```
help.search("sin")
```

and see what that does (answer: help.search pulls up all help pages that include 'sin' anywhere in their title or text. Apropos just looks at the name of the function).

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`.

```
 [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
```

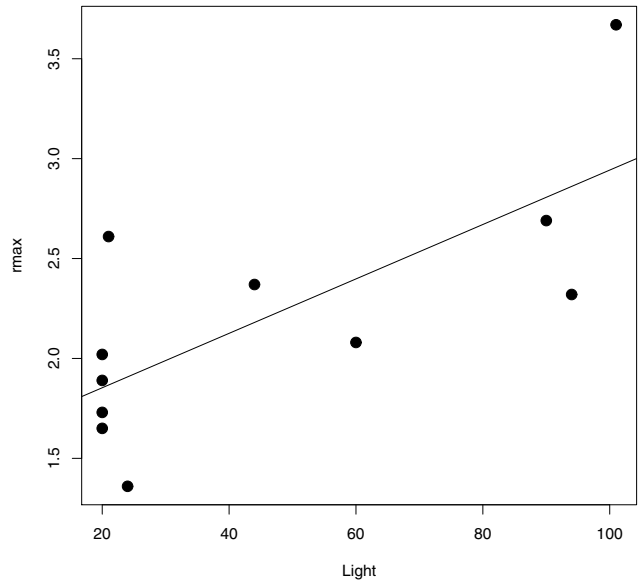

Figure 1: Graphical summary of linear regression analysis. The commands for this plot are in `Intro1.R`. Data are from the studies described in Fussmann et al. (2000). Science 290: 1358-1360.

3 Script files and data files

Modeling or complicated data analysis are often accomplished more efficiently using *scripts*, which are a series of commands stored in a text file. As of this writing, most versions of R include a built-in script editor. If your's doesn't, you will need to use an an external text editing program (e.g. Windows Notepad or PFE).

Most programs for working with models or analyzing data follow a simple pattern of program parts:

1. "Setup" statements.
2. Input some data from a file or the keyboard.
3. Carry out the calculations that you want.
4. Print the results, graph them, or save them to a file.

For example, a script file might

1. Load some libraries, or run another script file that creates some functions (more on functions later).
2. Read in from a text file the parameter values for a predator-prey model, and the numbers of predators and prey at time $t = 0$.
3. Calculate the population sizes at times $t = 1, 2, 3, \ldots, T$.
4. Graph the results, and save the graph to disk for including in your term project.

Even for relatively simple tasks, script files are useful for build up a calculation step-by-step, making sure that each part works before adding on to it.

As first examples, the files **Intro1.R** has the commands from the interactive regression analysis. **Important:** before working with a script file that you have downloaded to your computer, create a copy of it and work with the copy, not the original.

Now open **your copy of** of **Intro1.R**. In your editor, select and Copy the entire text of the file, and then Paste the text into the R console window. This has the same effect as if you entered the commands by hand into the console, and they will be executed resulting in a graph being displayed with the results. Cut-and-Paste allows you to execute script files one piece at a time (which is useful for finding and fixing errors). The `source` function allows you to run an entire script file, e.g.

```
> source("c:/temp/Intro1.R")
```

Source'ing can also be done in point-and-click fashion via the **File** menu on the console window.

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

```
X=read.table("c:\\temp\\ChlorellaGrowth.txt")
```

reads the file and puts the data values into the variable `X`. **NOTE** that we specified the path to the file; you will have to do the same, using the correct path for your computer. The double-backslash notation is one way to indicated subfolders in Windows; a single forward slash also works, as in the `source()` statement above.

The variables are then extracted from `X` with the commands

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

Think of these as shorthand for "Light = everything in column 1 of X", and "rmax = everything in column 2 of X" (we'll learn about working with matrices later). From there out it's the same as before, with some additions that set the axis labels and add a title.

Exercise 3.1 Make a copy of **Intro2.R** under a new name, and modify the copy so that it does linear regression of algal growth rate on the log of light intensity, LogLight=log(Light), and plots the data appropriately. You should end up with a graph sort of like Figure (2).

Exercise 3.2 Run **Intro2.R**, then enter the command `plot(fit)` in the console and follow the directions in the console. Figure out what just happened by entering `?plot.lm` to bring up the Help page for the function `plot.lm` that carries out a `plot` command for an object produced by `lm`. [This is one example of how R uses the fact that statistical analyses are stored as model objects. `fit` "knows" what kind of object it is (in this case an object of type `lm`), and so `plot(fit)` invokes a function that produces plots suitable for an `lm` object.] **Answer:** R produced a series of diagnostic plots exploring whether or not the fitted linear model is a suitable fit to the data. In each of the plots, the 3 most extreme points (the most likely candidates for "outliers") have been identified according to their sequence in the data set.

Exercise 3.3 The axes in plots are scaled automatically, but the outcome is not always ideal (e.g. if you want several graphs with exactly the same axes limits). You can control scaling using the `xlim` and `ylim` arguments in `plot`:

```
plot(x,y,xlim=c(x1,x2),ylim=c(y1,y2))
```

will draw the graph with the x-axis running from x1 to x2, and the y-axis running from y1 to y2.

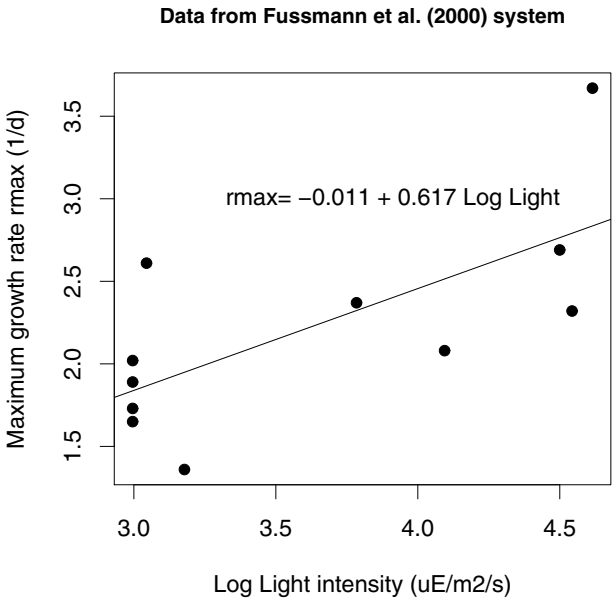

Figure 2: Graphical summary of regression analysis using log of light intensity.

Create a plot of growth rate versus Light intensity with the x axis running from 0 to 120, and the y axis running from 1 to 4.

Exercise 3.4 Several graphs can be placed within a single figure by using the `par` function (short for "parameter") to adjust the layout of the plot. For example the command

```
par(mfrow=c(m,n))
```

divides the plotting area into m rows and n columns. As a series of graphs are drawn, they are placed along the top row from left to right, then along the next row, and so on. `mfcol=c(m,n)` has the same effect except that successive graphs are drawn down the first column, then down the second column, and so on.

Save **Intro2.R** with a new name and modify the program as follows. Use `mfcol=c(2,1)` to create graphs of growth rate as a function of Light, and of log(growth rate) as a function of log(Light) in the same figure. Do the same again, using `mfcol=c(1,2)`.

Exercise 3.5 Use ?par to read about other plot control parameters that can be set using `par()`. Then write a script that draws a 2×2 set of plots, each showing the line $y = 5x + 3$ with x running from 3 to 8, but with 4 different line styles and 4 different line colors. Recall that `x=3:8` will create `x` as a vector of the integers from 3 to 8 inclusive.

Exercise 3.6 Use `?savePlot` to read about the `savePlot` function, and then use it modify one of your scripts so that at the very end the plot is saved to disk. Note that the argument `filename` in savePlot can include the path to a folder, for example `filename="c:/temp/Intro2Figure"` in Windows. [1]

[1] This and the previous seem to be about plotting, but really they are exercises in using the Help system with the bonus that you learn more about `plot`. (Let's see, we know `plot` can graph data points...maybe it can also draw a line to connect the points, or just draw the line and leave out the points. That would be useful. So let's try `?plot` and see if it says anything about lines...Hey, it also says that `graphical parameters can be given as arguments to plot`, so maybe I can set line colors inside the plot command instead of using `par` all the time....). The Help system is *very* helpful once you get used to it and acquire the habit of using it often.

4 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 4.1 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 4.2 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 4.3 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 $n = 50$.

`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.

Vector addressing

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

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

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

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

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

Extracted parts of a vector don't have to be regularly spaced. For example

```
> v=q[c(1,2,5)]; v
[1] 1 3 9
```

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

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

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

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

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

Exercise 4.4 write a **one-line** command to extract a vector consisting of the second, first, and third elements of `q` **in that order**.

Exercise 4.5 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.

5 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

```
> 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 5.1 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 5.2 Use `rnorm` and `matrix` to create a 5×7 matrix of Gaussian random numbers with mean 1 and standard deviation 2.

`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.

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.

5.1 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 5.3 Verify that `rbind(A,B)` works, `cbind(A,A)` works, but `cbind(A,B)` doesn't. Why not?

5.2 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 5.4 Use **runif** to construct a 5×5 matrix **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 **B** by `2 5 8 11 14`.

5.3 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
```

6 Iteration ("Looping")

6.1 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 6.1 : 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 6.2 : 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

6.2 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

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.

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 6.3 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 6.4 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)`.

7 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 7.1 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 7.2 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,

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

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.

8 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. Many functions exist in two versions corresponding to these, with the default choice 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.*

`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.

8.1 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.[2] `eigen` is convenient for the former: it 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

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

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 "$" is used to extract components of a compound object, for example `vA$values` is the `values` part of `vA`, a vector consisting of the sorted eigenvalues:

```
vA$values
```

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

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

Note that the eigenvalues are sorted by magnitude,so `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
```

Exercise 8.1 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 8.2 Enter the command `names(vA)` and see what results. From this, infer what the `names` function does. Enter the command `names(A)` and infer the meaning of the object `NULL` in R . Check your guess by using the Help menu on the console window: select R language (standard) and type NULL into the popup window that appears.

To get the left eigenvectors of a matrix, you have to use `eigen` again. Recall that the left eigenvectors of a matrix A are the same as the eigenvectors of its transpose, `t(A)`. So

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

gets you the left eigenvectors of A.

Exercise 8.3 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 8.4 : 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.

8.2 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 8.5 Construct the transition matrix $\mathbf{A}$, and then find λ, $\mathbf{v}$, $\mathbf{w}$ for an age-structured model with the following survival and fecundity parameters.

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

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

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

Results: $\lambda = 1.0419$

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

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

8.3 Finding the eigenvalue with largest real part

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 where the real part is maximized.

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

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

Exercise 8.6 Try the above on `A=diag(3,1,3)` and see what you get for `lmax` and `vmax`. Why?

9 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 9.1 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 9.2 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 9.3 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.

10 A simulation project

This section is an optional "capstone" project putting into use the programming skills that have been covered so far. Nothing new about R *per se* is covered in this section.

The first step is to write a script file that simulates a simple model for density-independent population growth with spatial variation. The model is as follows. The *state variables* are the numbers of individuals in a series of $L = 20$ patches along a line (L stands for "length of the habitat") .

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

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

1. Geometric growth within patches:

$$M_j(t) = \lambda_j N_j(t) \quad \text{for all } j. \tag{2}$$

2. Dispersal between neighboring patches:

$$N_j(t+1) = (1-2d)M_j(t) + dM_{j-1}(t) + dM_{j+1}(t) \quad \text{for } 2 \le j \le L-1 \tag{3}$$

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

$$\begin{aligned} N_1(t+1) &= (1-d)M_1(t) + dM_2(t) \\ N_L(t+1) &= (1-d)M_L(t) + dM_{L-1}(t) \end{aligned} \tag{4}$$

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

Notes and hints:

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

2. Notice that this model is not *totally* different from **Loop1.m**, in that you start with a founding population at time 1, and use a loop to compute successive populations at times 2,3,4, and so on. The difference is that the population is described by a vector rather than a number. Therefore, to store the population state at times $t = 1, 2, \cdots, 50$ you will need a matrix `njt` with 50 rows and L columns. Then `njt(t,:)` is the population state vector at time t.

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

Exercise 10.1 Use the model (modified as necessary) to ask how the spatial arrangement of good versus bad habitat patches affects the population growth rate. For example, does it matter if all the good sites ($\lambda > 1$) are at one end or in the middle? What if they aren't all in one clump, but are spread out evenly (in some sense) across the entire habitat? **Be a theoretician**: (a) Patterns will be easiest to see if good sites and bad sites are very different from each other. (b) Patterns will be easiest to see if you come up with a nice way to compare growth rates across different spatial arrangements of patches. (c) Don't confound the experiment by also changing the proportion of good versus bad patches at the same time you're changing the spatial arrangement.

Exercise 10.2 Modify your script file for the model (or write it this way to begin with...) so that the dispersal phase (equations 3 and 4) is done by calling a function `reflecting` whose arguments are the pre-dispersal population vector $M(t)$ and the dispersal parameter d, and which returns $N(t+1)$, the population vector after dispersal has taken place.

11 Coin tossing and Markov Chains

The exercises on coin tossing and Markov chains in Chapter 3 of the textbook can be used as the basis for a computer-lab session. For convenience we also include them here. All of the R functions and programming methods required for these exercises have been covered in previous sections, but it is useful to "remember"

- how to generate sets of random uniform and Gaussian random numbers using `runif` and `rnorm`.
- how logical operators can be used to convert a vector of numbers into a vector of 1's and 0's according to whether or not a condition holds.
- how to find the places in a vector where the value changes, using logicals and `which`.

```
>> v=rnorm(100);
>> u = (v<0.3);
>> w=which(u[2:100]!=u[1:99])
```

Exercise 11.1 Experiment with sequences of coin flips produced by a random number generator:

- Generate a sequence `r` of 1000 random numbers uniformly distributed in the unit interval $[0, 1]$.
- Compute and plot a histogram for the values with 10 equal bins of length 0.1. How much variation is there in values of the histogram? Does the histogram make you suspicious that the numbers are not independent and uniformly distributed random numbers?
- Now compute sequences of 10000 and 100000 random numbers uniformly distributed in the unit interval $[0, 1]$, and a histogram for each with ten equal bins. Are your results consistent with the prediction that the range of variation between bins in the histogram is proportional to the square root of the sequence length?

Note: `q=hist(runif(1000),10)` will plot the first histogram you need, and `q$counts` will hold the number in each bin of the histogram.

The theoretical "benchmark" for this experiment, and its relation to coin tossing, goes as follows. Let X_i be 1 if the i^{th} random number falls into the first bin of the histogram, otherwise $X_i = 0$, for $i = 1, 2, 3, \cdots n$. So each X_i is a coin toss with $P(H) = 0.1$. The fraction of tosses that fall in the first bin is then $f = (X_1 + X_2 + \cdots X_n)/n$. Using the basic properties of means and variances, we can derive that $E(f) = E(X)$ and $Var(f) = Var(X)/n$. The 10 bins of the histogram that you constructed can be regarded as (approximately) 10 repetitions of this experiment, because there's nothing special about bin 1: they all have the same "probability of Heads". So if you compute

`n=10000; q=hist(runif(n),10); f1=q$counts/n; sqrt(var(f1))`

and then compute `f2` with $n = 100,000$, you should see (what?)

Exercise 11.2 (a) Recall that coin tossing is modeled by the binomial distribution: the probability of k heads in a sequence of n tosses, with $p = 0.6$ being the probability of heads, is given by

$$c_k(0.6)^k(0.4)^{1000-k} \quad \text{where } c_k = \binom{1000}{k} = \frac{1000!}{k!(1000-k)!}.$$

In R , the binomial coefficient $\binom{n}{k}$ is computed using `choose(n,k)`, e.g. `choose(1000,5)` gives $\binom{1000}{5}$.

Calculate the probability of k heads for values of k between 500 and 700 in a sequence of 1000 independent tosses. Plot your results with k on the x-axis and the probability of k heads on the y-axis. Comment on the shape of the plot.

(b) Generate a sequence of 1000 uniformly distributed random numbers `r`, as in the last exercise, convert them into a sequence of coin tosses in which the probability of heads is 0.6 and the probability of tails is 0.4, and compute the total number of heads in the 1000 coin tosses. Write a script that does this 1000 times (i.e., 1000 repetitions of tossing 1000 coins) and for each repetition stores (in a vector) the total number of heads. Now test the binomial distribution by plotting a histogram of the number of heads obtained in each repetition, and compare the results with the predictions of the binomial distribution.

(c) Modify the last experiment by doing 10000 repetitions of 100 coin tosses. Comment on the differences you observe between this histogram and the histogram for 1000 repetitions of tossing 1000 coins.

Tossing multi-sided coins Uniform random numbers can also be used to simulate "coin-toss" experiments where the coin has 3 or more sides. Conceptually this is easy. If there are n sides with probabilities $p_1, p_2, \cdots, p_n$, you generate `r=runif(1)` and declare that

Side 1 occurs if $r \leq p_1$.

Side 2 occurs if $p_1 < r \leq p_1 + p_2$

Side 3 occurs if $p_1 + p_2 < r \leq p_1 + p_2 + p_3$

...

Side n occurs if $p_1 + p_2 + \cdots + p_{n-1} < r$.

To code this compactly in R , note that the outcome depends on the cumulative sums

$$c_1 = p_1, \quad c_2 = p_1 + p_2, \quad c_3 = p_1 + p_2 + p_3, \quad \cdots, c_n = 1.$$

If r is larger than exactly k of these, then the outcome is Side $k+1$. Cumulative sums of a vector are computed using the function `cumsum`.

As an example, the following code tosses a 5-sided coin with probabilities given by the vector `p` defined in the first line:

```
p=c(.3,.1,.3,.1,.2); b=cumsum(p);
side=sum(runif(1)>b)+1
```

To toss the coin repeatedly you can use the `sapply` function, which applies an arbitrary function to all elements in a vector.

```
side=sapply(runif(1000),FUN=function(x) sum(x>b)+1)
```

In the code above the function to be applied is defined within the call to `sapply`. More complicated functions can be defined separately, prior to the call, as in the following example:

```
side.choose=function(x) {sum(x>b) +1}
side=sapply(runif(1000),FUN=side.choose)
```

Exercise 11.3 Generate 10000 tosses of a 3-sided coin (with your choice of probabilities), and plot a histogram of the results to verify that your coin is behaving the way it should.

11.1 Markov chains and residence times

The purpose of the following exercises is to generate synthetic data for single channel recordings from finite state Markov chains, and to explore patterns in the "data". Single channel recordings give the times that a Markov chain makes a transition from a closed to an open state or vice versa. The histogram of expected residence times for each state in a Markov chain is exponential, with different mean residence time for different states. To observe this in the simplest case, we again consider coin tossing. The two outcomes, heads or tails, are the different states in this case. Therefore the histogram of residence times for heads and tails should each be exponential. The following steps are taken to compute the residence times:

- Generate sequences of independent coin tosses based on given probabilities.
- Look at the number of *transitions* that occur in each of the sequences (a *transition* is when two successive tosses give different outcomes).
- Calculate the residence times by counting the number of tosses between each transition.

Exercise 11.4 Find the script `cointoss.R`. This program calculates the residence times of coin tosses by the above methodology. Are the residence times consistent with the prediction that their histogram decreases exponentially? Produce a plot that compares the predicted results with the simulated residence times stored by `cointoss.R` in the vectors `hhist` and `thist`. (Suggestion: use `par(ylog=TRUE)` to use a logarithmic scale for the plotted values). Note: a run of exactly k H's in a row occurs whenever a T is followed by k H's in a row and then another T. How many of these should occur in the output of `cointoss.R`?

Models for stochastic switching among conformational states of membrane channels are somewhat more complicated than the coin tosses we considered above. There are usually more than 2 states, and the transition probabilities are state dependent. Moreover, in measurements some states cannot be distinguished from others. We can observe transitions from an open state to a closed state and vice versa, but transitions between open states (or between closed states) are "invisible". Here we shall simulate data from a Markov chain with 3 states, collapse that data to remove the distinction between 2 of the states and then analyze the data to see that it cannot be readily modeled by a Markov chain with just two states. We can then use the distributions of residence times for the observations to determine how many states we actually have.

We will consider a membrane that has three states: two closed states C_1 and C_2, and one open state O. We assume that direct transitions between C_1 and O are impossible, but that the intermediate state C_2 has a shorter residence time than C_1 and O. Here is the transition matrix of a Markov chain we will use to simulate these conditions:

$$\begin{array}{ccc} C_1 & C_2 & O \\ \begin{bmatrix} .98 & .1 & 0 \\ .02 & .7 & .05 \\ 0 & .2 & .95 \end{bmatrix} & \begin{array}{c} C_1 \\ C_2 \\ O \end{array} \end{array}$$

You can see from the matrix that the probability 0.7 of staying in state C_2 is much smaller than the probability 0.98 of staying in state C_1 or the probability 0.95 of remaining in state O.

Exercise 11.5 Generate a set of 100000 samples from the Markov chain with these transition probabilities. We will label the state C_1 by 1, the state C_2 by 2 and the state O by 3. This can be done by a modification of the method that we used to toss coins with 3 or more sides. The modification is that the probabilities of each side depend on the current state of the membrane:

```
nt = 100000;
A = matrix(c(0.98, 0.10, 0, 0.02, 0.7, 0.05, 0, 0.2, 0.95),3,3,byrow=T);
B = apply(A,2,cumsum); #cumulative sums of each column
A; B;
states=numeric(nt+1); rd=runif(nt);
states[1] = 3; # Start in open state
for(i in 1:nt) {
   b=B[,states[i]]; #cumulative probabilities for current state
   states[i+1]=sum(rd[i]>b)+1 # do the ‘‘coin toss’’ based on current state
}
plot(states[1:1000],type="s");
```

Notice the use of `apply` to compute the cumulative sum of each column of the transition matrix, and `type="s"` to get a "stairstep" plot of the state transitions (see `?plot`).

Exercise 11.6 Compute the eigenvalues and eigenvectors of the matrix A. Compute the total time that your "data" in the vector `states` spends in each state (use vector operations to do this!) and compare the results with predictions coming from the dominant right eigenvector of A.

Exercise 11.7 Produce a new vector `rstates` by "reducing" the data in the vector `states` so that states 1 and 2 are indistinguishable. The states of `rstates` will be called "closed" and "open".

Exercise 11.8 Plot histograms of the residence times of the open and closed states in `rstates` by applying the methods used in the script `cointoss.R`. Comment on the shapes of the distributions in each case. Using your knowledge of the transition matrix A, make a prediction about what the residence time distributions of the open states should be. Compare this prediction with the data. Show that the residence time distribution of the closed states is not fit well by an exponential distribution.

12 The Hodgkin-Huxley model

The purpose of this section is to develop an understanding of the components of the Hodgkin-Huxley model for the membrane potential of a space-clamped squid giant axon. It goes with the latter part of Chapter 3 in the textbook, and with the **Recommended reading:** Hille, Ion Channels of Excitable Membranes, Chapter 2.

The Hodgkin-Huxley model is the system of differential equations

$$\begin{aligned} C\frac{dv}{dt} &= i - \left[g_{Na}m^3h\,(v-v_{Na}) + g_K n^4\,(v-v_K) + g_L\,(v-v_L)\right] \\ \frac{dm}{dt} &= 3^{\frac{T-6.3}{10}}\left[(1-m)\Psi\left(\frac{-v-35}{10}\right) - 4m\exp\left(\frac{-v-60}{18}\right)\right] \\ \frac{dn}{dt} &= 3^{\frac{T-6.3}{10}}\left[0.1\,(1-n)\,\Psi\left(\frac{-v-50}{10}\right) - 0.125n\exp\left(\frac{-v-60}{80}\right)\right] \\ \frac{dh}{dt} &= 3^{\frac{T-6.3}{10}}\left[0.07(1-h)\exp\left(\frac{-v-60}{20}\right) - \frac{h}{1+\exp(-0.1(v+30))}\right] \end{aligned} \tag{5}$$

where

$$\Psi(x) = \frac{x}{\exp(x)-1}.$$

The state variables of the model are the membrane potential v and the ion channel gating variables m, n, and h, with time t measured in msec. Parameters are the membrane capacitance C, temperature T, conductances g_{Na}, g_K, g_L, and reversal potentials v_{Na}, v_K, v_L. The gating variables represent channel

opening probabilities and depend upon the membrane potential. The parameter values used by Hodgkin and Huxley are:

g_{Na}	g_K	g_L	v_{Na}	v_K	V_L	T	C
120	36	0.3	55	-72	-49.4011	6.3	1

Most of the data used to derive the equations and the parameters comes from voltage clamp experiments of the membrane, e.g Figure 2.7 of Hille.

In this set of exercises, we want to see that the model reproduces the voltage clamp data well, and examine some of the approximations and limitations of the parameter estimation. Note that because $T = 6.3$ in the parameter set we are using, the prefactor $3^{\frac{T-6.3}{10}}$ in the equations for m, n and h equals 1, and it can be omitted in all of the exercises. Also, the exercises all consider voltage clamp experiments in which the membrane potential $v(t)$ is externally imposed, and is constant except for instantaneous jumps from one value to another. So for the situation we are considering here, the model (5) reduces to:

$$\begin{aligned}
\frac{dm}{dt} &= (1-m)\Psi\left(\frac{-v-35}{10}\right) - 4m\exp\left(\frac{-v-60}{18}\right) \\
\frac{dn}{dt} &= 0.1\,(1-n)\,\Psi\left(\frac{-v-50}{10}\right) - 0.125n\exp\left(\frac{-v-60}{80}\right) \\
\frac{dh}{dt} &= 0.07(1-h)\exp\left(\frac{-v-60}{20}\right) - \frac{h}{1+\exp(-0.1(v+30))}
\end{aligned} \tag{6}$$

When the membrane potential v is constant, the equations for the gating variables m, n, h are first order linear differential equations that can be rewritten in the form

$$\tau_x \frac{dx}{dt} = -(x - x_\infty)$$

where x is m, n or h.

Exercise 12.1 Re-write the differential equations for m, n, and h in the form above, thereby obtaining expressions for τ_m, τ_n, τ_h and $m_{\text{inf}}, n_{\text{inf}}, h_{\text{inf}}$ as functions of v.

Exercise 12.2 Write an R script that computes and plots τ_m, τ_n, τ_h and $m_{\text{inf}}, n_{\text{inf}}, h_{\text{inf}}$ as functions of v for v varying from -100mV to 75mV. You should obtain graphs that look like Figure 2.17 of Hille.

In voltage clamp, $\frac{dv}{dt} = 0$ so we obtain the following formula for the current from the Hodgkin-Huxley model:

$$i = g_{Na}m^3h(v - v_{Na}) + g_K n^4(v - v_K) + g_L(v - v_L)$$

The solution of the first order equation

$$\tau_x \frac{dx}{dt} = -(x - x_\infty)$$

is

$$x(t) = x_\infty + (x(0) - x_\infty)\exp(\frac{-t}{\tau_x})$$

Exercise 12.3 Write an R script to compute and plot as a function of time the current $i(t)$ obtained from voltage clamp experiments in which the membrane is held at a potential of -60mV and then stepped to a higher potential v_s for 6msec. (When the membrane is at its holding potential -60mV, the values of m, n, h approach $m_\infty(-60), n_\infty(-60), h_\infty(-60)$. Use these approximations as starting values.) As in Figure 2.7 of Hille, use $v_s = -30, -10, 10, 30, 50, 70, 90$ and plot each of the curves of current on the same graph.

Exercise 12.4 Separate the currents obtained from the voltage clamp experiments by plotting on separate graphs each of the sodium, potassium and leak currents.

Exercise 12.5 Hodgkin and Huxley's 1952 papers explain their choice of the complicated functions in their model, but they had no computers available to analyze their data. In this exercise and the next, we examine procedures for estimating from experimental data the sodium current parameters $m_\infty, h_\infty, \tau_m, \tau_h$. However, the data that we will use will be generated by the model itself.

As in Exercises 2 and 3, compute the Hodgkin-Huxley sodium current generated by a voltage clamp experiment with a holding potential of -90mV and steps to $v_s = -80, -70, -60, -50, -40, -30, -20, -10, 0$. This is your "data".[3] Then using the expression $g_{Na}m^3h(v - v_{Na})$ for the sodium current, estimate m_∞, τ_m, h_∞ and τ_h as functions of voltage from this simulated data. The most commonly used methods assume that τ_m is much smaller than τ_h, so that the activation variable m reaches its steady state before h changes much. So for times just after the step occurs, you can assume that m is changing but h is holding constant at its initial value. For times long after the step (many multiples of τ_m), you can assume that h is changing but m is holding constant at its steady state value.

Explain the procedures that you used. Some of the parameters are difficult to determine, especially over certain ranges of membrane potential. Why? How do your estimates compare with the values computed in Exercise 1?

Challenge: For the parameters that you had difficulty estimating in the previous exercise, design and simulate voltage clamp protocols that help you estimate these parameters better. (See Hille, pp. 44-45.) Describe your protocols and how you estimate the parameters. Plot the currents produced by the model for your new experiments, and give the parameter estimates that you obtain using the additional "data" from your experiments. Further investigation of these procedures is a good topic for a course project.

12.1 Getting started

We offer here some suggestions for completing the exercises in this section.

Complicated expressions are often built by composing simpler expressions. In any programming language, it helps to introduce intermediate variables. Here, let's look at the gating variable h first. We have

$$\frac{dh}{dt} = 0.07 \exp\left(\frac{-v-60}{20}\right)(1-h) - \frac{h}{1+\exp(-0.1(v+30)}$$

Introduce the intermediate expressions

$$a_h = 0.07 \exp\left(\frac{-v-60}{20}\right)$$

and

$$b_h = \frac{1}{1+\exp(-0.1(v+30)}$$

Then

$$\frac{dh}{dt} = a_h(1-h) - b_h h = a_h - (a_h + b_h)h$$

We can then divide this equation by $(a_h + b_h)$ to obtain the desired form

$$\tau_h \frac{dh}{dt} = -(h - h_\infty)$$

[3] Analyzing data that come from a model in order to estimate parameters that you already know may sounds a bit crazy at first, but statisticians do this all the time and many of them are not crazy. Before working with real data, it's a good idea to generate artificial "data" from the model with known parameters, and make sure that your approach to fitting the model allows you to correctly recover the parameters that generated the "data".

as

$$\frac{1}{a_h + b_h}\frac{dh}{dt} = \frac{a_h}{a_h + b_h} - h.$$

Comparing these two expressions we have

$$\tau_h = \frac{1}{a_h + b_h}, \qquad h_\infty = \frac{a_h}{a_h + b_h}.$$

Implementing this in R to compute the values of $h_\infty(-45)$ and $\tau_h(-45)$, we write

```
v = -45;
ah = 0.07*exp((-v-60)/20);
bh = 1/(1+exp(-0.1*(v+30)));
tauh = 1/(ah+bh);
hinf = ah/(ah+bh);
```

Evaluation of this script gives tauh = 4.6406 and hinf = 0.1534.

To do the second exercise, for t_h and h_∞, we want to evaluate the script for values of v that vary from -100 to 75. We will do this at integer values of v with a loop. We first make vectors to hold the data, and then store each value as it is computed:

```
tauh = rep(0,176); # to start with v = -100, end with v = 75
hinf = rep(0,176);

for(j in 1:176){
    v = -101+j; # so v goes from -100 to 75
    ah = 0.07*exp((-v-60)/20);
    bh = 1/(1+exp(-0.1*(v+30)));
    tauh(j) = 1/(ah+bh);
    hinf(j) = ah/(ah+bh);
}
```

The same strategy can be used to compute $m_\infty, \tau_m, n_\infty, \tau_n$, but there is one slight twist: the function Ψ. This function defined by

$$\Psi(x) = \frac{x}{\exp(x) - 1}$$

gives the *indeterminate* value $0/0$ when $x = 0$, so R cannot evaluate it there. Nonetheless, using *l'Hopital's rule* from calculus, we can define $\Psi(0) = 1$. When computing the values in R , either avoid $x = 0$ or use an `if` statement to check whether $x = 0$. It is helpful in writing R scripts to compute the terms involving Ψ to introduce intermediate variables for its argument:

```
...
amv = -(v+35.0)/10.0;
am = amv/(exp(amv) - 1);
...
```

You will need some of the data from the second exercise in completing the third and fourth, and you should extend the range of v to include $v = 90$ for these exercises. It is helpful to define the parameters, a vector t of the time values that you want to use in computing the currents, values of m, n, h at the holding potential $v = -60$, values of $m_\infty, n_\infty, h_\infty$ and τ_m, τ_n, τ_h at the potentials of the steps, arrays that

will hold all of the data for each of the gating variables m, n, h, etc. before you compute the currents. Use code like `m[s,j] = m1[s] + (m0 - m1[s])*exp(-t[j]/mt1[s])` to compute the gating variables, with `m0` the value of m at the holding potential, `m1[s]` the value of m_∞ during the step and `mt1[s]` the value of τ_m during the step. Once these arrays have been computed, use

$$i = g_{Na} m^3 h(v - v_{Na}) + g_K n^4 (v - v_K) + g_L (v - v_L)$$

to compute the total current, with $g_{Na} m^3 h(v - v_{Na})$ and $g_K n^4 (v - v_K)$ giving the sodium and potassium currents.

The fifth exercise requires much more ingenuity than the previous ones. To get started, repeat computations like those of Exercise 3 to generate the sodium current data used in the exercise. Currents must be converted to conductances by dividing by $(v - v_{Na})$. After this is done, strategies must be developed to estimate $m_\infty, h_\infty, \tau_m, \tau_h$. Frequently used procedures assume that

1. the experiment starts at a potential sufficiently hyperpolarized that there is no inactivation (i.e. $h = 1$) and
2. activation is so fast relative to inactivation that m reaches its steady state before h has changed significantly.

This separation of time scales – m changing much faster than h because $\tau_m \ll \tau_h$ – means that early, rapid changes in conductance are caused by changes in m, while later, slower changes in conductance are caused by changes in h. One can then estimate m_∞ and τ_m from the increasing portion of the conductance traces, assuming that $h = 1$ during this entire period of time, so that the conductance is $g_{Na} m^3$. Then to estimate τ_h we assume that the decreasing "tail" of the conductance curve is given by $g_{Na} m_\infty^3 h(t)$, because $m(t)$ has already converged to its steady state.

It is easier to estimate h_∞ from a different set of voltage traces, having different values of the holding potential v_0, followed by a step to another potential v_1 that is the same for each trace. In this protocol we start with h partially inactivated (i.e. $h < 1$), so the peak conductance during the subsequent trace is proportional to the initial value of h, which is $h_\infty(v_0)$. A plot of peak conductance as a function of v_0 is then *proportional to* a plot of h_∞ as a function of v_0. The constant of proportionality can be inferred from the fact that the maximum value of $h_\infty(v)$ is 1. Consult Hille for further descriptions of these protocols.

13 Solving systems of differential equations

R built-in 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

$$\begin{aligned} \frac{du}{dt} &= -u + \frac{\alpha_u}{1+v^\beta} \\ \frac{dv}{dt} &= -v + \frac{\alpha_v}{1+u^\gamma} \end{aligned} \tag{7}$$

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 (7) 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 13.1 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 (7) 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 **odesolve** package. Download this (if necessary) from CRAN, and then use the command `library(odesolve)` to load it into your R session.

Exercise 13.2 After you load the odesolve 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 13.3 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, and that these constants have different values. 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 13.4 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 13.5 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 13.6 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 13.7 Change the value of α from 3 to 1.5. How does the phase portrait change? Plot the nullclines to help answer this question.

13.1 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 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. 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 13.8 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 13.9 Write a script using `lsoda` to solve the constant population size SIR model with births,

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

For parameter values $\mu = 1/60, \gamma = 25$ (corresponding to a mean lifetime of 60 years, and disease duration of 1/25 of a year) and population size $S(0) + I(0) + R(0) = 1000000$, explore how the dynamics of the disease prevalence $I(t)$ changes as you increase the value of β from 0.

13.2 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, suppose that the number of infectives becomes negative in a standard SIR-type infectious disease model, the transmission rate βSI becomes negative. So contacts between susceptibles and infectives still occur, but their effect in the model is to make the sick one become healthy, pushing I even more negative. Once a model goes down the rabbit hole, it may 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 much easier than it sounds, because of the fact that for any state variable $x(t)$,

$$\frac{d(\log x(t))}{dt} = \frac{1}{x(t)}\frac{dx}{dt}. \tag{8}$$

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 computes the toggle-switch model vector field on log scale. The input argument `logy` is the log-transformed state vector $(\log u, \log v)$.

```
Toggle.log=function(t,logy,parms) {
y=exp(logy);
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/y));
}
```

There are only two differences from the original `Toggle`: the first line that back-transforms from `logy` to `y`, and the last line that uses equation (8) to compute the vector field of the log-transformed state vector. And there are just two minor differences in the call to lsoda: log-transforming the initial conditions, and back-transforming the output.

```
x0=log(c(.2,.1)); times=seq(0,50,by=1); parms=c(3,2,2);
out=lsoda(x0,times,Toggle.log,parms);
out[,-1]=exp(out[,-1]) #column 1 is time, which was not transformed
matplot(out[,1],out[,2:3],type="l",ylim=c(0,3),xlab="time t",ylab="u,v");
```

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 13.10 Write a script to solve the Lotka-Volterra model (in the previous subsection) using the log trick. Note that for this model you can do (8) in your head and avoid the `dY/y` calculation.

14 Equilibrium points and linearization

This section continues our study of differential equations with R . We will investigate the computation of *equilibrium points* and their *linearization*. Recall that an equilibrium point of the system $\frac{dx}{dt} = f(x)$ is a vector x_0 in the phase space where $f(x_0) = 0$. If the model's state vector has dimension n, the equilibrium condition $f(x_0) = 0$ is a system of n equations in n variables that may have multiple solutions, or no solutions at all.

Solving nonlinear equations is a difficult task for which there are no sure-fire algorithms. *Newton's method* is a simple *iterative* algorithm that is usually very fast when it works, but it doesn't always work. Newton's method takes as its input a starting value of x_0, ideally one that is close to the solution of $f(x_0) = 0$ that we seek. It evaluates $y_0 = f(x_0)$ and terminates if the magnitude of y_0 is smaller than a desired tolerance. If y_0 is larger than the desired tolerance, then a new value x_1 is computed from the solution of the linear or tangent approximation to f at x_0:

$$f(x) \approx L(x) = f(x_0) + Df(x_0)(x - x_0)$$

Here $Df(x_0)$ is the $n \times n$ Jacobian matrix of the partial derivatives of f evaluated at x_0 – the j^{th} column of Df is the derivative of f with respect to the j^{th} state variable. If $Df(x_0)$ is an invertible matrix, then we can solve the linear system $L(x) = 0$ for x, yielding the new value of x:

$$x_1 = x_0 - [Df(x_0)]^{-1} f(x_0)$$

So we replace x_0 by x_1 and start over again by evaluating $f(x_1)$. If its magnitude is small enough, we stop. Otherwise, we take x_1 as our new "starting value" and repeat the process to find a point x_2 that is (hopefully) closer to being a solution of $f(x) = 0$.

Close to a solution of $f(x) = 0$ at which $Df(x)$ is an invertible matrix, Newton's method converges "quadratically", meaning that the error after one additional iteration is proportional to the current error squared – which is a very good thing one the error has become small.

The function `newton` in `DMBpplane.R` implements Newton's method functions set up in the format used for solving differential equations with `rk4` or `lsoda`. So we can apply Newton's method to our toggle switch model (7) using the commands

```
newton(Toggle,x0=c(2.5,0.4),parms=c(3,2,2))
```

The function newton has three optional arguments for which default values are provided: the maximum number of iterations `niter`, the convergence tolerance `tol`, and the increment `inc` used to compute Jacobian matrices Df by finite-difference approximation to the derivatives.

Exercise 14.1 Download and run `DMBpplane.R`, and then run the command above. Note that the values of the state variables are displayed at each iteration. Recall that for these parameter values there are three equilibrium points. Choose different values of x_0 to find the other two equilibrium points.

The file `repress.R` implements the six dimensional repressilator model of Elowitz and Leibler:

```
repress=function(t,y,p){
   dy = rep(0,6);
   dy[1] = -y[1] + p[1]/(1.+y[6]^p[4])+ p[2];
   dy[2] = -y[2] + p[1]/(1.+y[4]^p[4])+ p[2];
   dy[3] = -y[3] + p[1]/(1.+y[5]^p[4])+ p[2];
   dy[4] = -p[3]*(y[4]-y[1]);
   dy[5] = -p[3]*(y[5]-y[2]);
   dy[6] = -p[3]*(y[6]-y[3]);
   return(list(dy))
}
```

Note that to save typing `parms` is replaced here by `p` – the name doesn't matter so long as it's in the right place in the list of arguments.

Exercise 14.2 Use `lsoda` and `repress` to reproduce the textbook figure that shows oscillations in this model by computing and graphing a trajectory for this model with parameters `parms = c(50,0,0.2,2)`. Almost any initial conditions should work – try `x0 = 2*runif(6)`

Exercise 14.3 Use Newton's method to compute an equilibrium point of the repressilator for the same values of the parameters.

We can use eigenvalues and eigenvectors as tools to study solutions of a vector field near an equilibrium point x_0, as discussed in the textbook. The basic idea is that we approximate the vector field by the *linear* system

$$\frac{dw}{dt} = Aw$$

where A is the $n \times n$ matrix $Df(x_0)$ that `newton` computes for us and $w = x - x_0$. In many circumstances the phase portrait of this linear system will look similar to the phase portrait of $\frac{dx}{dt} = f(x)$. Now, if v is an eigenvector of A with eigenvalue λ, the curve

$$w(t) = \exp(t\lambda)v$$

is a solution of $\frac{dw}{dt} = Aw$ because $Av = \lambda v$.

If the eigenvalue λ is negative, then the exponential $\exp(t\lambda) \to 0$ as $t \to \infty$. Complex eigenvalues give solutions that have trigonometric terms: $\exp(it) = \cos(t) + i\sin(t)$. Whenever the real parts of all the eigenvalues are negative, the equilibrium point is *linearly stable*. Otherwise it is unstable.

Exercise 14.4 Compute the eigenvalues of the Jacobian at the equilibrium point that you found for the repressilator model (recall that the Jacobian at the equilibrium is one of the components of the object returned by `newton`). Now change the parameters to `parms = c(50,1,0.2,2)` and recompute the equilibrium point and its eigenvalues.

Exercise 14.5 Compute the eigenvalues of the three equilibrium points for the toggle switch model with `parms = c(3,2,2)` – by plotting the nullclines you will be able to pick starting conditions for `newton` near each of these. You should find that the two equilibria off the diagonal are stable. The equilibrium point on the diagonal has one positive and one negative eigenvalue, making it a *saddle*.

Exercise 14.6 Continuing with the toggle switch model: choose initial points $x = x_0 \pm \epsilon w_1$ where $0 < \epsilon \ll 1$ and w_1 is the eigenvector of the Jacobian corresponding to the positive eigenvalue. Compute and plot (on the same graph as the nullclines) solution trajectories starting at these points. These trajectories are an approximation to the *unstable manifold* of the saddle.

Exercise 14.7 Next do the same for the eigenvector w_2 corresponding to the negative eigenvalue, but integrate *backward* in time; i.e., choose a negative final time for your integration. These trajectories approximate the *stable manifold* of the saddle.

Parameter	*Set 1*	*Set 2*
g_{Ca}	4.4	5.5
g_K	8	8
g_L	2	2
v_{Ca}	120	120
v_K	-84	-84
v_L	-60	-60
C	20	20
ϕ	0.04	0.22
i	90	90
v_1	-1.2	-1.2
v_2	18	18
v_3	2	2
v_4	30	30

Table 8: Parameter sets for the Morris-Lecar model.

15 Phase-plane analysis and the Morris-Lecar model

Recommended reading: Rinzel and Ermentrout, Analysis of Neural Excitability and Oscillations. In: Koch and Segev, Methods in Neuronal Modeling: From Synapses to Networks (2nd edition). MIT Press, Cambridge, MA (1998).

In this section we continue to study phase portraits of two-dimensional vector fields, using the Morris-Lecar model for the membrane potential of barnacle muscle fiber. The differential equations for the Morris-Lecar model are

$$\begin{aligned} C\frac{dv}{dt} &= i - g_{Ca}m_\infty(v)(v - v_{Ca}) - g_K w(v - v_K) - g_L(v - v_L) \\ \tau_w(v)\frac{dw}{dt} &= \phi(w_\infty(v) - w) \\ m_\infty(v) &= 0.5\left(1 + \tanh(\frac{v - v_1}{v_2})\right) \\ w_\infty(v) &= 0.5\left(1 + \tanh(\frac{v - v_3}{v_4})\right) \\ \tau_w(v) &= \frac{1}{\cosh\left(\frac{v - v_3}{2v_4}\right)} \end{aligned} \tag{9}$$

The parameters used in the textbook are listed in table 8.

For phase-plane analysis we recommend stepping outside R and using a specialized tool. If your computer lab has MATLAB installed you can use John Polking's `pplane`, which runs inside MATLAB but does not require any knowledge of the MATLAB language. Another fine option is Bard Ermentrout's open-source program XPP (Ermentrout 2002). Finally, we provide an R utility called `Rpplane` that includes the essential tools for the phase-plane exercises here and in the textbook, but not much else.

Using pplane At this writing, `pplane` for various versions of Matlab can be obtained at `http://math.rice.edu/~dfield`. Download the appropriate versions `pplane?.m` and `dfield?.m` into any convenient folder. Then start Matlab, and use File/open on the Matlab menu bar to load `pplane?.m`

into the Matlab script editor. In the editor, use Debug/run to start `pplane`.[4]

To load the Morris-Lecar model into `pplane`, download the file `ML.pps` from the textbook webpage. In the Pplane7 setup window, use File/load a system... and select `ML.pps`. The "default" system in the setup window will then be replaced by the Morris-Lecar model, with two parameters free to be varied, g_{Ca} and ϕ. Click on the Proceed bar at the bottom right of the setup window; this will open a graph window in which you can work with the phase portrait. Everything you need is in the menus, but it will take a bit of looking around or a guided tour by your instructor.

Using XPP The home page for XPP is `www.math.pitt.edu/~bard/xpp/xpp.html`; you can download the source code and binaries for various operating systems, including (at this writing) OS X. For Windows PCs, the best option at this writing appears to be the one described at `www.math.pitt.edu/~bard/xpp/ezwin.html`, with links to all required files. Also download the file `MLdmb.ode` from the textbook web page, and save it into any convenient folder. If you follow Bard's directions for installing XPP, you'll be able to launch the program by first starting the X server that you installed, and then drag-and-drop a model-definition file, such as `MLdmb.ode`, onto the `xpp.bat` icon on your desktop. There's some online documentation and a tutorial for XPP at its home page, and full details in Ermentrout (2002).

Using Rpplane The function `Rpplane` is included in `DMBpplane.R`. The syntax is

```
Rpplane(fun,xlim,ylim,parms=NULL,add=F,ngrid=5,maxtime=100)
```

- `fun` is the function defining the vector field, which needs to be in the format of the `toggle` function of the last section.
- `xlim,ylim` define the plotting region.
- `parms` is the vector of any parameter values that are used by `fun`.
- `add` indicates whether the results should be displayed on the currently active graphics device.
- `ngrid` is the number of grid points used for plotting a grid of trajectories
- `maxtime` is the time interval over which solution trajectories are computed

So `Rpplane(ml,c(-60,40),c(-0,1),parms=c(5.5,0.22),maxtime=250);` will launch Rpplane with parameters $g_{Ca} = 5.5, \phi = 0.22$ using the `ml` function that defines the vector field in `DMBpplane.R`, drawing trajectories out to time $t = 250$, and default values for the other arguments. A high value of `maxtime` is needed in this system for drawing the stable/unstable manifolds of saddle points.

Always start by drawing the nullclines or phase arrows. You can then add to those plots by selecting from the menu. To start a trajectory, find a fixed point or get the local stable/unstable manifold for a saddle, you need to click on the current graph to select the starting point. Fixed points with complex eigenvalues are plotted as circles (closed = stable, open = unstable), those with real eigenvalues are plotted as triangles. The fixed points and their eigenvalues are printed to the console window. At saddles, the unstable manifold is plotted in red and the stable manifold in blue.

Exercise 15.1 Compute phase portraits for the Morris-Lecar model at the two different tabulated sets of parameter values. Label

- Each of the equilibrium points by type,
- The stable and unstable manifolds of any saddle points

[4]Note that "?" here is a pplane version number, not literally a question mark. At this writing pplane7 is the latest version, so you would download `pplane7.m` and `dfield7.m`, and load `pplane7.m` to get it started.

- The stability of the periodic orbits.

It's OK to print the graph out and do the labeling by hand, or you can use the `text` function in the Console window to add text to an existing plot.

Bifurcations of a system occur at parameters where the number of equilibria or periodic orbits change. The typical bifurcations encountered while varying a single parameter at a time in a system with at most a single saddle point are

1. **Saddle-node bifurcation:** The Jacobian at an equilibrium points has a zero eigenvalue.
2. **Hopf bifurcation:** The Jacobian at an equilibrium point has a pair of pure imaginary eigenvalues.
3. **Homoclinic bifurcation:** There is a trajectory in both the stable and unstable manifold of a saddle.
4. **Saddle-node of limit cycle bifurcation:** A periodic orbit has double eigenvalue 1.

The changes in dynamics that occur at each kind of bifurcation are discussed in Chapter 5 of the textbook.

Exercise 15.2 At saddle-node bifurcations, two equilibria appear or disappear. Figure 5.14 of the textbook shows that as g_{Ca} is varied, saddle-node bifurcations occur near $g_{Ca} = 5.32$ and $g_{Ca} = 5.64$. Compute phase portraits for values of g_{Ca} near these bifurcations, describing in words how the phase portraits change.

Exercise 15.3 Now set $g_{Ca} = 5.5$ and vary ϕ in the range from $(0.04, 0.22)$. Show that both Hopf and homoclinic bifurcations occur in this range. What are the approximate parameter values at which bifurcations occur? Draw labeled phase portraits on both sides of the bifurcations, indicating the changes that occur.

Exercise 15.4 Hopf bifurcations are *supercritical* if stable periodic orbits emerge from the equilibrium and *subcritical* if unstable periodic orbits emerge from the equilibrium. Is the Hopf bifurcation you located in the previous exercise subcritical or supercritical? Explain how you know.

Exercise 15.5 With g_{Ca} set to 4.4, show that the two periodic orbits you computed in the first exercise approach each other and coalesce as ϕ is increased. This is a saddle-node of limit cycles (*snlc*) bifurcation. Draw phase portraits on the two sides of the bifurcations.

Exercise 15.6 For parameter values $\phi = 0.33$ and g_{Ca} varying near the saddle-node value approximately 5.64, the saddle-node is a *snip*. Explain what this is using phase portraits as an illustration.

16 Simulating Discrete-Event Models

This section is an introduction to simulating models that track discrete agents (organisms, molecules, neurons) as they change in state, as an alternative to compartment models that assume large numbers of agents. It can can be regarded as a warmup for simulating finite-population disease models (Chapter 6 in the textbook), or as some simple examples of agent-based models (Chapter 8).

Figure 3 shows a compartment model for biased movement of particles between two compartments. The corresponding system of differential equations is

$$\begin{array}{l} \frac{dx_1}{dt} = Lx_2 - Rx_1 \\ \frac{dx_2}{dt} = Rx_1 - Lx_2 \end{array} \tag{10}$$

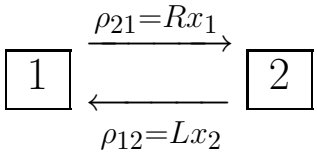

Figure 3: Compartment diagram for biased movements between 2 compartments.

Even for molecules – but much more so for individuals catching a disease – changes in state are discrete events in which individuals move one by one from one compartment to another. In some cases, such as molecular diffusion, transitions can really occur at any instant. But for modeling purposes we can have transitions occurring at closely-spaced times $dt, 2dt, 3dt, \cdots$ for some *short* time step dt, and allow each individual to follow a Markov chain with transition matrix

$$A = \begin{bmatrix} 1-(R\times dt) & L\times dt \\ R\times dt & 1-(L\times dt) \end{bmatrix}$$

In `TwoState.R` the movement decisions for many particles are made by using `runif` to toss many coins at once. If there are N particles in compartment 1, each with probability Rdt of moving to compartment 2, then `sum(runif(N)<R*dt)` simulates the combined outcome (number moving) from all of the "coin tosses". Note in `TwoState.R` that at each time step, first *all* coins are tossed – for all particles in all compartments – and only then are particles moved to update the state variables. Note also the use of **ifelse** to avoid trying to toss zero coins (and see `?ifelse`).

Each simulation of the model will have a different outcome, but some properties will be more or less constant. In particular

1. Once dt is small enough to approximate a continuous-time process, further decreases in dt have essentially no effect on the behavior of simulations. Roughly, dt is small enough to model continuous time if there would be practically no chance of an individual in continuous time doing 2 or more things in a time interval of length dt. For this model, that means that we must have $(Rdt)\times(Ldt) \ll 1$, i.e. $dt \ll 1/\sqrt{RL}$.
2. A compartment's typical range of departures from solutions of the differential equation are of order $1/\sqrt{n}$ where n is the expected number of particles *in the compartment*.

If there are many particles, coin-tossing with **runif** becomes slow. Instead, we can recall that the binomial random variable $\mathbf{B}(N,p)$ is the number of heads in N coin-tosses with probability p of heads on each toss. In R , binomial-distributed random numbers are generated by the `rbinom` function. The syntax is:

```
>> rbinom(n,N,p)
```

This generates a vector of `n` random numbers from a $\mathbf{B}(N, p)$ distribution. The script file `TwoState2.R` uses **rbinom** instead of **runif** to do the coin tosses.

Exercise 16.1 The pure death process in discrete time tracks a population of initial size N in which the only events are the occasional death of individuals. Between times t and $t+1$, each individual alive at time t has probability p of death, and probability $1-p$ of surviving to time $t+1$, independent of what happens to any other individual in the population. Eventually everybody is dead, so the main summary measure of a simulation run is the time it takes until the population is extinct.

Write an R script to simulate the pure death process by using coin-tossing as in the two-compartment example above. That is, if it is now time t and there are $N(t)$ individuals still alive, we do $N(t)$ coin-tosses with probability p of Heads(=death) and subtract those from the current population. This continues until everyone is dead. The first line of your m-file should specify the values of N and p, e.g.

```
N=250; p=0.05;
```

Exercise 16.2 One run of a stochastic model is not the whole story, because the next run will be different. One needs to do multiple runs, and look at the distribution of possible outcomes. Extend your script file from the last exercise so that it does 100 runs of the pure death process (all having the same values of N and p), stores the time at which the population goes extinct in each run, computes the mean and standard deviation of extinction times across runs, and plots a histogram of the extinction times.

Exercise 16.3 Let's give the two-compartment example a totally different interpretation. We have n potential sites for local populations in a landscape, which can either be empty (state=1) or occupied (state=2). Then R is the chance that an empty site becomes occupied, and L is the chance that an occupied site becomes extinct. It's plausible that L is constant over time, but the rate at which empty sites are occupied should depend on the how many sites are occupied. So modify `TwoState2.R` so that in each time step, R is proportional to the number of occupied (state=2) sites. Note that you should choose parameters so that R is never bigger than 1 – that is, even if only one site is empty, that site has some probability of remaining empty.

Exercise 16.4 Modify your script from the last exercise so that it runs until all sites are in state 1 or until $t = 100$, whichever comes first, and prints out the extinction time (the population as a whole goes extinct when all sites are in state 1). Find parameters such that extinction at or before $t = 100$ is likely but not 100% certain.

17 Simulating dynamics in systems with spatial patterns

Recommended reading: Winfree (1991).

The purpose of this section is to investigate the formation of spiral waves by a reaction-diffusion mechanism in the simplest possible manner. The system of equations that we study has as its reaction mechanism the *Fitzhugh-Nagumo* model that is often used as a caricature for the Hodgkin-Huxley equations:

$$\begin{aligned} \frac{\partial u}{\partial t} &= \frac{1}{e}(u - \frac{u^3}{3} - v) + D\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) \\ \frac{\partial v}{\partial t} &= e(u + b - 0.5v) \end{aligned} \tag{11}$$

In this form of the model the substance v does not diffuse - the model is the extreme limit of the differing diffusion constants that are required for pattern formation by the Turing mechanism. In an electrophysiological context, v represents the gating variable of a channel (which does not move), while u represents the membrane potential which changes due to diffusion of ions in the tissue as well as by transmembrane currents. The tissue could be the surface of the heart, or (with one space dimension) a nerve axon.

To solve equation (11) we want to discretize both space and time, replacing the derivatives in the equations by finite differences. For the time derivatives, we estimate

$$\frac{\partial u}{\partial t}(x, y, t) \approx \frac{u(x, y, t + h) - u(x, y, t)}{h}$$

and

$$\frac{\partial v}{\partial t}(x, y, t) \approx \frac{v(x, y, t + h) - v(x, y, t)}{h}$$

h being the *time step* of the method. For the spatial derivatives, we estimate

$$\frac{\partial^2 u}{\partial x^2}(x, y, t) \approx \frac{\frac{\partial u}{\partial x}(x, y, t) - \frac{\partial u}{\partial x}(x - k, y)}{k} \approx \frac{u(x + k, y, t) - u(x, y, t) - (u(x, y, t) - u(x - k, y, t))}{k^2}$$

and

$$\frac{\partial^2 u}{\partial y^2}(x, y, t) \approx \frac{\frac{\partial u}{\partial y}(x, y, t) - \frac{\partial u}{\partial y}(x, y - k, t)}{k} \approx \frac{u(x, y + k, t) - u(x, y, t) - (u(x, y, t) - u(x, y - k, t))}{k^2}$$

The values of the function u "in the lower-left corner" of the lattice is given by

$\vdots$	$\vdots$	$\vdots$	$\vdots$
$u(k, 3k, t)$	$u(2k, 3k, t)$	$u(3k, 3k, t)$	$\cdots$
$u(k, 2k, t)$	$u(2k, 2k, t)$	$u(3k, 2k, t)$	$\cdots$
$u(k, k, t)$	$u(k, 2k, t)$	$u(3k, k, t)$	$\cdots$

We will work with a rectangular domain and impose *no flux* boundary conditions. This means that none of the u material should flow out of the domain due to the diffusion. Each of the terms of the form $u(x, y) - u(x - k, y)$ in the discretized Laplacian represents the net material flowing between two sites. Therefore, if we surround the domain by an additional "ring" of sites that take the same values as those at the adjacent site in the interior of the domain, then we can apply the discrete approximation of the Laplacian throughout the domain, including the sites at the domain boundary. Using this trick, the following R function calculates the right hand side of our discretized operator for the arrays u and v:

```
sfn=function(u,v) {
```

```
    ny = dim(u)[1]; nx = dim(u)[2];
    uer = cbind(u[,1],u,u[,nx]); # u with extra rows
    uec = rbind(u[1,],u,u[ny,]); # u with extra columns
    ul = uec[3:(ny+2),]+uec[1:ny,]+uer[,1:nx]+uer[,3:(nx+2)]-4*u;
    uf = (u-0.3333*u*u*u-v)/e + dx2*ul;
    vf = e*(u + b-0.5*v);
    return(list(uf=uf,vf=vf));
}
```

An important practical consideration here is that there are *no loops*. Everything is written as matrix operations. The program is already quite slow to run - loops would make it intolerable.

We use the simple Euler method to update the points, and plot the results:

```
sfn.run=function(nsteps,init) {
   u=init$u; v=init$v;
   for(i in 1:nsteps){
     out=sfn(u,v);
     u = u+h*out$uf; v = v+h*out$vf;
     if(i%%5==1) image(1:dim(u)[2],1:dim(u)[1],t(u),col=rainbow(n=100,start=0,end=0.7))
   }
    return(list(u=u,v=v));
}
```

Because each time step depends on the results from the previous step, we *do* need a for-loop to iterate the model over time. The value returned by `sfn.run` is the final state of the model, suitable for use as a new "initial condition" to continue the run.

The file sfn.R includes everything needed to produce and plot simulations of spiral patterns. First, select and run (using the R script editor or a text editor) the block of code with the functions `sfn` and `sfn.run`. Then select and run the code for the first parameter set/initial condition at the bottom of the script. The command

```
nsteps=1000; # or however long you want to run it
out=sfn.run(nsteps,init1)
```

will then run the model with the first parameters and initial conditions for 1000 iterations. To continue the model run, you can use `sfn.run` again, but starting from the model state at the last time step:

```
out=sfn.run(nsteps,out)
```

Here are some more things things that you can do with these files:

- Restart the model using the second set of parameters and initial conditions, and see what happens when spiral waves collide with with one another.
- Figure 13 from Winfree (1991) shows a (b, e) bifurcation diagram for the *rotor* patterns that he studied. Can you reproduce some of these patterns?
- Experiment with changing the spatial discretization parameter k. What effect do you expect to see on the spatial pattern?

- Experiment with *gradually* increasing the time step h. You should discover that solutions quickly go from reasonable to nonsensical, or the computation crashing. It is characteristic of explicit solution methods, such as those in `sfn.run`, that very small values of h are needed to keep numerical solutions from becoming *unstable* and developing errors that grow exponentially fast.

17.1 General method of lines

The solution method in `sfn.R` consists of three steps:

1. Space is discretized so that the model only "lives" at a discrete grid of points $\{(x_i, y_j)\}$.
2. Values of state variables at the grid points are used to approximate the derivatives with respect to x and y that appear in the dynamic equations (11).
3. We numerically solve the system of coupled ODEs that result from steps 1 and 2.

This general approach is called the *method of lines*. This name comes from imagining a set of lines running forward in time, one based at each grid point, along which we solve the ODEs. In `sfn.R` we used a regular grid for step 1, a finite difference approximation for step 2, and Euler's method for step 3. All of those are choices favor simplicity over efficiency. More accurate results can be gotten with a bit more work. As one simple example, `sfnLines.R` uses order-4 Runga-Kutta for step 3, by re-writing the sfn function so it can be used with `rk4`. Because `rk4` is also an explicit method (like Euler's method) it still requires short time steps, but it gives more accurate solution of the ODE system. Only implicit solution methods, such as the widely used Crank-Nicholson finite-difference method, remain numerically stable for long time steps.

The accuracy of step 2 can be greatly improved by using *spectral methods* (Trefethen 2000). These interpolate the state variables "globally", fitting a single smooth function that matches the values at the grid points. Spatial derivates of the interpolating function are then used as the approximate spatial derivates in step 2. The best choice of interpolating function depends on the spatial domain and boundary conditions. Spectral methods with suitable interpolating functions can be very efficient, because the entire (interpolate+differentiate) process can be expressed as a single matrix multiplication, with a matrix that only has to be computed once. Details of these methods are beyond our scope here; see Trefethen (2000) for an introduction and pointers to more advanced treatments.

According to Trefethen (2000), spectral methods are the method of choice for high-accuracy solutions on simple spatial domains, but explicit methods for solving the ODE system often require very short time steps so it is advantageous to use implicit or semi-implicit methods. For complex domains, such as often arise in engineering applications, finite element methods are generally preferred. In general, apart from simple PDEs like those considered here that can be solved using relatively simple of methods, it is better to step outside R and use specialized software for numerical solution of PDEs.

18 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. 2005. 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.