APPENDIX: AN INTRODUCTION TO R

Introduction

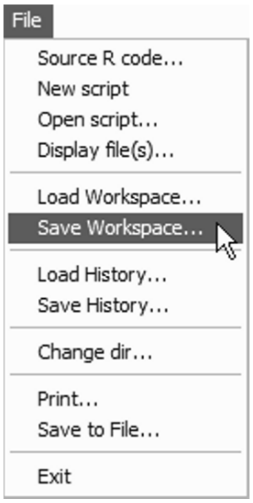
All of the plots and numerical output displayed in this book were produced with the R software, which is available at no cost from the R Project for Statistical Computing. The software is available under the terms of the Free Software Foundation's GNU General Public License in source code form. It runs on a wide variety of operating systems, including Windows, Mac OS, UNIX, and similar systems, including FreeBSD and Linux. R is a language and environment for statistical computing and graphics, provides a wide variety of statistical methods (time series analysis, linear and nonlinear modeling, classical statistical tests, and so forth) and graphical techniques, and is highly extensible. In particular, one of the authors (KSC) has produced a large number of new or enhanced R functions specifically tailored to the methods described in this book. They are available for download in an R package named TSA on the R Project Website at www.r-project.org. The TSA functions are listed on page 468.

Important references for learning much more about R are also available at the R-Project Website, including *An Introduction to R: Notes on R, a Programming Environment for Data Analysis and Graphics.* Version 2.4.1 (2006-12-18), by W. N. Venables, D. M. Smith, and the R Development Core Team, (2006), and *R: A Language and Environment for Statistical Computing Reference Index*, Version 2.4.1 (2006-12-18), by The R Development Core Team (2006a).

The R software is the GNU implementation of the famed S language. It has been under active development by the R team, with contributions from many statisticians all over the world. R has become a versatile and powerful platform for doing statistical analysis. We shall confine our discussion to the Windows version of R. To obtain the software, visit the Website at www.r-project.org. Click on CRAN on the left-side of the screen under Download. Scroll down the list of CRAN Mirror sites and click on one of them nearest to you geographically. Click on the link for Windows (or Linux or MacOS X as appropriate) and click on the link named **base**. Finally, click on the link labeled R-2.6.1-win32.exe. (This file indicates release 2.6.1, the latest available release as of this writing. Newer versions come out frequently.) Save the file somewhere convenient, for example, on your desktop. When the download finishes, double-click the program icon and proceed with installing the software. (The discussion that follows assumes that you accept all of the defaults during installation.) At the end of this appendix, on page 468, you will find a listing and brief description of all the new or enhanced functions that are contained in the TSA package.

Before you start the R software for the first time, you should create a folder or directory, say `Rwork`, to hold data files that you will use with R for this project or course. This will be the working directory whenever you use R for this particular project or course. This directory is to contain the `workspace`, a file that contains all the objects (variables and functions) created in an R session. You should create separate working directories for different projects or different courses.[†] After R is successfully installed on your computer, there will be an R shortcut icon on your desktop. If you have created your working directory, start R by clicking the R icon (shown at the right). When the software has loaded, you will have a console window similar to the one shown in Exhibit 1 with a bottom line that reads > followed by a large rectangular cursor (probably in red). This is the R prompt. You may enter commands at this prompt, and they will be carried out when you press the Enter key. Several tasks are available through the menus.

The first task is to save your workspace in the working directory you created. To do so, select the `File` menu and then click on the choice `Save workspace...` .[‡] You now may either browse to the directory `Rwork` that you created (which may take many steps) or type in the full path name; for example "C:\Documents and Settings\JoeStudent\My Documents\Course156\Rwork". If your working directory is on a USB flash drive designated as drive E, you might simply enter "E:Rwork". Click `OK`, and from this point on in this session, R will use the folder `Rwork` as its working directory.



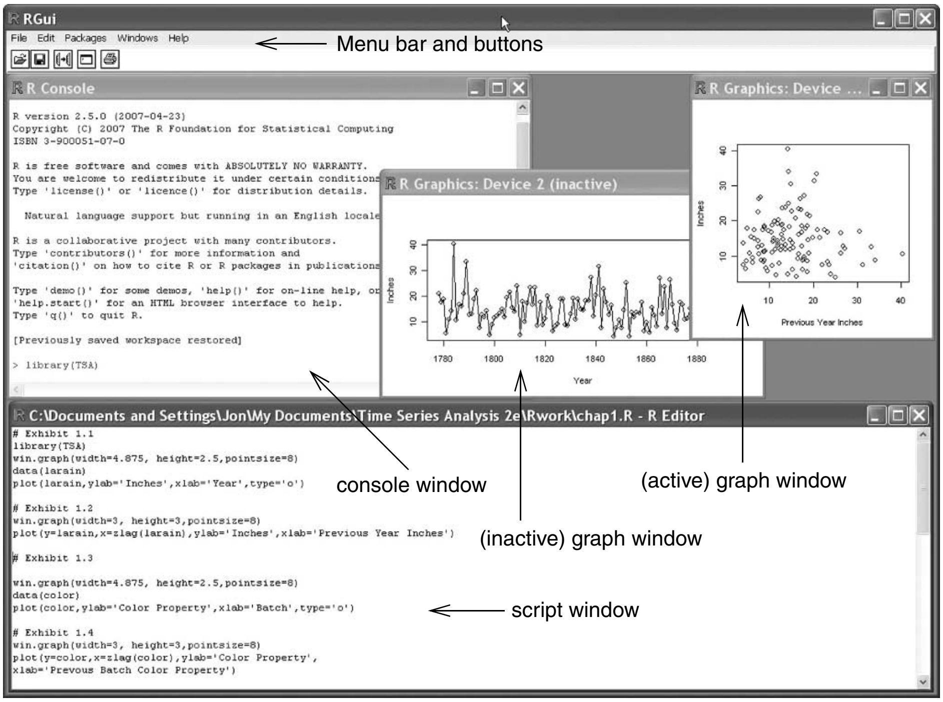
You exit R by selecting `Exit` on the `File` menu. Every time you exit R, you will receive a message as to whether or not to `Save the workspace image`. Click `Yes` to save the workspace, and it will be saved in your current working directory. The next time you want to resume work on that same project, simply navigate to that working directory and locate the R icon there attached to the file named `.RData`. If you double-click this icon, R will start with this directory already selected as the working directory and you can get right to work on that project. Furthermore, you will receive the message `[Previously saved workspace restored]`.

Exhibit 1 shows a possible screen display after you have started R, produced two different graphs, and worked with R commands in a script window using the R editor. Numerical results in R are displayed in the console window. Commands may be entered (keyed) in either the console window and executed immediately or (better) in a script window (the R editor) and then submitted to be run in R. The Menu bar and buttons will change depending on which window is currently the "focus."

[†] If you work in a shared computer lab, check with the lab supervisor for information about starting R and about where you may save your work.

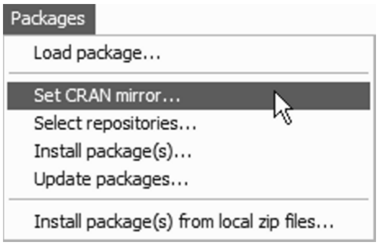
[‡] If you neglected to create a working directory *before* starting R, you may do so at this point. Navigate to a suitable place, click the `Create new folder` button, and create the folder `Rwork` now.

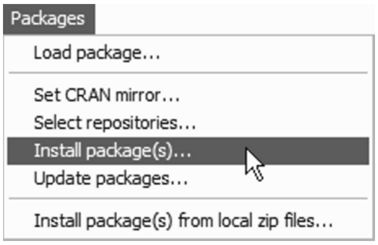
Exhibit 1 Windows Graphical User Interface for the R Software



```
R version 2.5.0 (2007-04-23)
Copyright (C) 2007 The R Foundation for Statistical Computing
ISBN 3-900051-07-0

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[Previously saved workspace restored]

> library(TSA)
```

```
# Exhibit 1.1
library(TSA)
win.graph(width=4.875, height=2.5,pointsize=8)
data(larain)
plot(larain,ylab='Inches',xlab='Year',type='o')

# Exhibit 1.2
win.graph(width=3, height=3,pointsize=8)
plot(y=larain,x=zlag(larain),ylab='Inches',xlab='Previous Year Inches')

# Exhibit 1.3

win.graph(width=4.875, height=2.5,pointsize=8)
data(color)
plot(color,ylab='Color Property',xlab='Batch',type='o')

# Exhibit 1.4
win.graph(width=3, height=3,pointsize=8)
plot(y=color,x=zlag(color),ylab='Color Property',
xlab='Prevous Batch Color Property')
```





A particularly useful feature of R is its ease of including supplementary tools in the form of libraries or packages. For example, all the datasets and the new or enhanced R functions used in this book are collected into a package called TSA that can be downloaded and installed in R. This can be done by clicking the Packages menu and then selecting `Set CRAN mirror`. Again select a mirror site that is closest to you geographically, and a window containing the names of all available packages will pop up.

In addition to our TSA package, you will need to install packages named `leaps`, `locfit`, `MASS`, `mgcv`, `tseries`, and `uroot`. Click the Packages menu once more, click `Install package(s)`, and scroll through the window. Hold down the Ctrl key and click on each of these seven package names. When you have all seven selected, click OK, and they will be installed on your system by R. You only have to install them once (but, of course, they may be updated in the future and some of them may be incorporated into the core of R and not need to be installed separately).

We will go over commands selected from the various chapters as a tutorial for R, but before delving into those, we first present an overview of R. R is an object-oriented language. The two main objects in R are data and functions. R admits many data structures. The simplest data structure is a vector that contains raw data. To create a data vector named `Dat` containing, say, 31, 4, 15, and 93, after the > prompt in the console window, enter the following command

```
Dat=c(31,4,15,93)
```

and then press the Enter key. The equal sign symbol signifies assigning the object on its right-hand side to the object on its left-hand side. The expression `c(31,4,15,93)` stands for concatenating the numbers within the parentheses to make a vector. So, the command creates an object named `Dat` that is a vector containing the numbers 31, 4, 15, and 93. R is case-sensitive, so the objects named `Dat` and `DAt` are different. To reveal the contents of an object, simply type the name of the object and press the Enter key. So, typing `Dat` in the R console window (and pressing the Enter key) will display the contents of `Dat`. If you subsequently enter `DAt` at the R prompt, it will complain by returning an error message saying that object "`DAt`" is not found. The name of an object is a string of characters that may contain letters, numerals, and the period sign, but the leading character is required to be a letter.[†] For example, `Abc123.a` is a valid name for an R object but `12a` is not. R has some useful built-in objects, for example `pi`, which contains the numerical value of π required for trigonometric operations such as computing the area of a circle.

For us, the most useful data structure is a time series. A time series is a vector with additional information on the epoch of the first datum and the number of data per a basic unit of time interval. For example, suppose we have quarterly data starting from the second quarter of 2006: 12, 31, 22, 24, 30. This time series can be created as follows:

```
> Dat2=ts(c(12,31,22,24,30), start=c(2006,2), frequency=4)
```

Its content can be verified by the command

```
> Dat2
     Qtr1 Qtr2 Qtr3 Qtr4
2006        12   31   22
2007   24   30
```

Larger datasets already in a data file (raw data separated by spaces, tabs, or line breaks) can be loaded into R by the command

```
> Dat2=ts(scan('file1'), start=c(2006,2), frequency=4)
```

where it is assumed that the data are contained in the file named `file1` in the same directory where you start up R (or the one changed into via the `change dir` command). Notice that the file name, `file1`, is surrounded by single quotes (`'`). In R, all

[†] Certain names should be avoided, as they have special meanings in R. For example, the letter `T` is short for true, `F` for false, and `c` for concatenate or combine.

character variables must be so enclosed. You may, however, use either single quotes or double quotes (") as long as you use them in pairs.

Datasets with several variables may be read into R by the `read.table` function. The data must be stored in a table form: The first row contains the variable names, and starting from the second line, the data are stored so that data from each case make up a row in the order of the variable names. The relevant command is

```
Dat3=read.table('file2',header=T)
```

where `file2` is the name of the file containing the data. The argument `header=T` specifies that the variable names are in the first line of the file. For example, let the contents of a file named `file2` in your working directory be as follows:

```
Y X
1 2
3 7
4 8
5 9
```

```
> Dat3=read.table('file2',header=T)
> Dat3
  Y X
1 1 2
2 3 7
3 4 8
4 5 9
```

Note that in displaying `Dat3`, R adds the row labels, defaulted to be from 1 to the number of data cases. The output of `read.table` is a `data.frame`, which is a data structure for a table of data. More discussion on `data.frame` can be found below. Presently, it suffices to remember that the variables inside a `data.frame` are not accessible. Think of `Dat3` as a closed suitcase. It has to be opened before its variables are accessible in an R session. The command to "open" a `data.frame` is to `attach` it:

```
> Y
Error: object "Y" not found
> attach(Dat3)
> Y
[1] 1 3 4 5
> X
[1] 2 7 8 9
```

R can also read in data from an Excel file saved in the `csv` (*comma-separated values*) format, with the first row containing the variable names. Suppose `file2.csv` contains a spreadsheet containing the same information as in `file2`. The commands for reading in the data from `file2.csv` are similar to the one for a text file.

```
> Dat4=read.csv('file2.csv',header=T)
> Dat4
  Y X
1 1 2
2 3 7
3 4 8
4 5 9
```

The functions `scan`, `read.table`, and `read.csv` have many other useful options. Use R Help to learn more about them. For example, run the command `?read.table`, and a window showing detailed information for the `read.table` command will open. Remember that prefacing the question mark to any function name will display the function's details in a new Help window.

Functions in R are similar to functions in the programming language C. A function is invoked by typing its name followed by a list of arguments enclosed by parentheses. For example, the concatenate function has the name "c" and its purpose is to create a vector obtained by concatenating the arguments supplied to the function.

```
> c(12,31,22,24,30)
```

Note that there can be no space between the left parenthesis and the function name. Even if the argument list is empty, the parentheses must be included in invoking a function. Try the command

```
> c
```

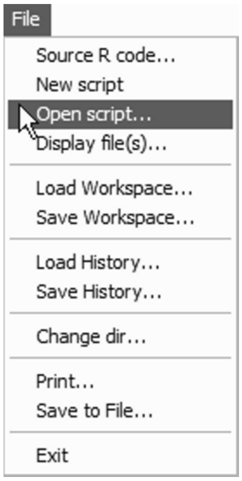
R now sees the name of an object and will simply display its contents by printing the entire set of commands making up the function in the console window. R has many useful built-in functions, including `abs`, `log`, `log10`, `exp`, `sin`, `cos`, `sqrt`, and so forth, that are useful for manipulating data. (The function `abs` computes the absolute value; `log` does the log-transformation with base *e*, while `log10` uses base 10; `exp` is the exponentiation function, `sin` and `cos` are the trigonometric functions; and `sqrt` computes the square root.) These functions are applied to a vector or a time series element by element. For example, `log(Dat2)` log-transforms each element of the time series `Dat2` and transfers the time series structure to the transformed data.

```
> Dat2=ts(c(12,31,22,24,30), start=c(2006,2), frequency=4)
> log(Dat2)
         Qtr1     Qtr2     Qtr3     Qtr4
2006          2.484907 3.433987 3.091042
2007 3.178054 3.401197
```

Furthermore, vectors and time series can be manipulated algebraically with the usual addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), or power (`^` or `**`) carried out element by element. For example, applying the transformation $y = 2x$^3 $- x + 7$ to `Dat2` and saving the transformed data to a new time series named `new.Dat2` can be easily carried out by the command

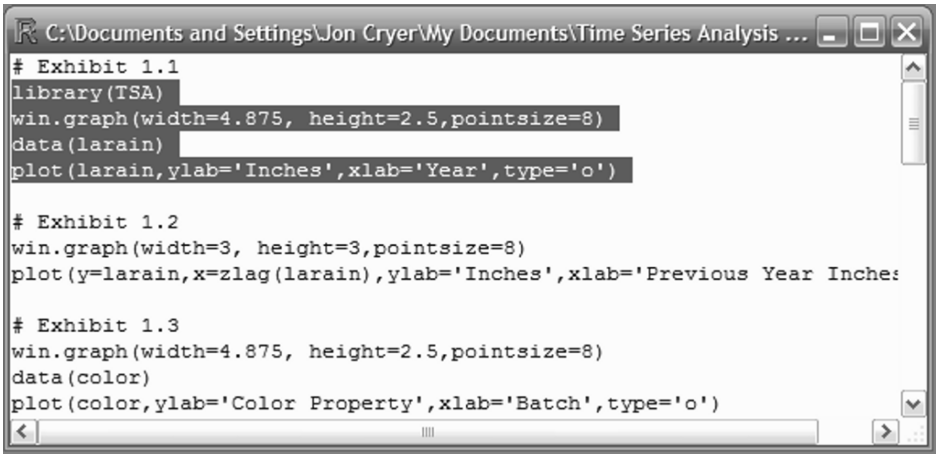
```
new.Dat2= 2*Dat2^3-Dat2+7
```

Chapter 1 R Commands



Now, we are ready to check out selected R commands used in Chapter 1 of the book. Script files of the commands used in each of the fifteen chapters are available for download at www.stat.uiowa.edu/~kchan/TSA.htm. The script files contain the R commands needed to carry out the analyses shown in the chapters. They also contain a limited amount of additional explanation. Download the scripts and save them in your working directory. You may then open them within R in an R editor (script) window and you will save much typing! Once they are downloaded, script files may be opened by either clicking the open file button or by using the file menu shown at the left.

Exhibit 2 A Script Window with Chapter 1 Scripts Displayed


C:\Documents and Settings\Jon Cryer\My Documents\Time Series Analysis ...

```
# Exhibit 1.1
library(TSA)
win.graph(width=4.875, height=2.5,pointsize=8)
data(larain)
plot(larain,ylab='Inches',xlab='Year',type='o')

# Exhibit 1.2
win.graph(width=3, height=3,pointsize=8)
plot(y=larain,x=zlag(larain),ylab='Inches',xlab='Previous Year Inche:

# Exhibit 1.3
win.graph(width=4.875, height=2.5,pointsize=8)
data(color)
plot(color,ylab='Color Property',xlab='Batch',type='o')
```

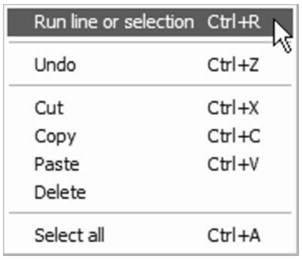


Exhibit 2 shows a portion of the script file for Chapter 1 in a script window. The first four commands have been highlighted by dragging the mouse pointer across them. They can now all be executed by either pressing Control-R (Ctrl-R) or by right-clicking the highlighted group and choosing Run from the choices displayed, as shown at the left. If the cursor is in a single command line with no highlighting, that one command may be executed similarly.

At the beginning of each session with R, you need to load the TSA library. The following command will accomplish this (but you may wish to investigate the `.First` function that can automate some startup tasks).

```
library(TSA)
```

The TSA package contains all datasets and functions needed for repeating the analyses and doing the exercises.

```
# Exhibit 1.1 on page 2.
win.graph(width=4.875,height=2.5,pointsize=8)
```

Comments may be interspersed in the R codes to improve their readability. The # sign in a R command signifies that what follows the sign are comments, and hence ignored by R. The first R command opening with the # sign is therefore a comment. The second R command opens a window for graphics that is 4.875 inches wide and 2.5 inches tall with characters printed with point size 8. The chosen setting and similar settings produce time sequence plots that are appropriate for inclusion in the book. Other settings will be appropriate for other purposes. For example, quantile-quantile plots are best viewed with a 1:1 aspect ratio (height = width). For exploratory data analysis, you will want larger graphics windows to use the full resolution of your computer screen to see more detail. The command `win.graph` can be safely omitted altogether. If there is currently no open graphics window, R will open a graphics window whenever a graphics command is issued. You can resize this window in the usual ways by dragging edges or corners.

```
data(larain)
```

This loads the time series `larain` into the R session and makes it available for further analysis such as

```
plot(larain,ylab='Inches',xlab='Year',type='o')
```

`Plot` is a function. It draws the time sequence plot for `larain`. The argument `ylab='Inches'` specifies “Inches” as the label for the *y*-axis. Similarly, the label for the *x*-axis is “Year.” The argument `type` indicates how the data are displayed in the plot. For `type='o'`, the individual data points are *overplotted* on the curve; `type='b'` (for *both*) is another option that superimposes the data points on the curve, but with the curve broken around the data points. For `type='l'`, only the *line* segments connecting the points are shown. (Note: This character (`l`) is an “el,” not a one.) To show only the data *points*, supply the argument `type='p'`. To learn more about the `plot` function and the full options for the `type` argument, run the command

```
?plot
```

A Help window on the `plot` function will then pop up for your browsing. Try it now. What will be plotted if the option `type='h'` is used instead of `type='o'`? All graphs may be saved (File > Save as > …) in any of several graphics formats: jpeg, pdf, etc. Saved graphs may then be imported into most word-processing programs to create high-quality reports.

```
# Exhibit 1.2 on page 2.
win.graph(width=3,height=3,pointsize=8)
plot(y=larain,x=zlag(larain),ylab='Inches',
   xlab='Previous Year Inches')
```

The plot function is a multipurpose function. It can do many different kinds of plots, depending on the set of arguments passed to it and their attributes. Here, it draws the scatter diagram of larain against its lag 1 values through the arguments y=larain (that is, larain on the *y*-axis) and x=zlag(larain) (that is, the lag 1 of larain is on the *x*-axis). Note that zlag is a function in the TSA package. Run the command ?zlag to learn what you can do with it.

```
# Exhibit 1.3 on page 3.
data(color)
plot(color,ylab='Color Property',xlab='Batch',type='o')
```

Here we have supplied four arguments to the plot function to draw the time sequence plot of the time series color. The first argument is simply color, but the other supplied arguments are of the form name of the argument = argument value so the first supplied argument is an unnamed argument, while the other arguments are named arguments. You may wonder how an unnamed argument is interpreted by R. To understand this, use the ?plot command to check that the argument list of the plot function is x, y, and … . You may guess that the x argument represents the *x*-variable, and the y argument for the *y*-variable in a plot. The ellipsis (…) argument stands for all other allowable arguments, which must, however, be specified with the name of the argument. (Again, consult the pages of the plot function to figure out which other arguments besides x and y may be passed to plot.) Any unnamed argument is interpreted to be the value for the argument whose order matches that of the unnamed argument supplied to the function. For example, color appears as the first argument supplied to the plot function, so R interprets it as the value for the x argument. Now there is no value supplied to the y argument. In this case, plot will examine the nature of the *x*-variable to determine what actions to be taken. Since color is a time series, plot draws a time sequence plot of color. To reinforce understanding, now try the following command in which color appears twice in the argument list, as the first and second arguments.

```
plot(color, color, ylab='Color Property',
   xlab='Batch',type='o')
```

Guess what will be drawn by R? Now, color is interpreted as the *x*-variable and also the *y*-variable; hence a 45 degree line is drawn. However, the line seems to be of nonuniform thickness. (Can you see this?) Why? It is because seeing that the variables are time series, plot draws the line by connecting data points in the order they are recorded, with the order of the data points marked in the plot. This feature can be useful in some analyses but in this case this feature is distracting. A remedy is to strip the time series attribute from the *x*-variables before plotting. (Plot takes the clue of how to do the plot from the attribute of the *x*-variable.) To temporarily turn color into a raw data vector, use the command

```
as.vector(color)
```

Now, try the command

```
plot(as.vector(color), color, ylab='Color Property',
   xlab='Batch',type='o')
```

```
# Exhibit 1.4 on page 4.
plot(y=color,x=zlag(color),ylab='Color Property',
   xlab='Previous Batch Color Property')
```

The `zlag` function outputs an ordinary vector; that is, `zlag(color)` is the lag 1 of `color`, but with its time series attribute stripped.

```
# Exhibit 1.9 on page 7.
plot(oilfilters,type='l',ylab='Sales')
```

`Plot` is a high-level graphics function and, as such, it will replace what is currently in the graphics window or create a new graphics window if none exists. Recall that the argument `type='l'` instructs `plot` to just draw the line segments connecting the individual time series points.

```
Month=c('J','A','S','O','N','D','J','F','M','A','M','J')
```

creates a vector named `Month` that contains 12 elements that represent the 12 months of the year beginning with July.

```
points(oilfilters,pch=Month)
```

`Points` is a low-level graphics function that draws on top of an existing graph. Since `oilfilters` is a time series, `points` plots `oilfilters` against time order, but the argument `pch=Month` instructs the `points` function to plot the data points using the successive values of the `Month` vector as plotting symbols. So, the first point plotted is plotted as a J, the second as an A, and so forth. When the values of `Month` are used up, they are recycled; think of `Month` being replicated as `Month`, `Month`, `Month`,…, to make up any deficiency. So, the 13th data point is plotted as a J and the 14th as an A. What letter is used for the 30th data point?

Alternatively, the exhibit can be reproduced by the following commands

```
plot(oilfilters,type='l',ylab='Sales')
points(y=oilfilters,x=time(oilfilters),
   pch=as.vector(season(oilfilters)))
```

The `time` function outputs the epochs when the time series values were collected. The `season` function returns the month of the data in `oilfilters`; `season` is a smart function, as it returns the quarter of the data for quarterly data and so forth. The `pch` argument expects a vector as its value, but the output of the `season` function has been designed to be a `factor` object; hence the application of the `as.vector` function to `season(oilfilters)` strips its factor attribute. (See more about `factor` objects on page 435.)

A good way to appreciate the natural variation in a stochastic process is draw realizations from the process and plot them in a time sequence plot. For example, the independent and identically normally distributed process is often used as a data generating mechanism for completely random data; that is, data with no temporal structure. In other words, such data constitute a random sample from a normal distribution that are drawn sequentially over time. Simulating data from such a process and viewing their time sequence plots is a valuable exercise that can train our eyes to differentiate whether a time series is random or dependent over time, c.f. Exercise 1.3. The R command for simulating and storing in a variable named y a random sample of size, say $n = 48$, from a standard normal distribution is

```
y=rnorm(48)
```

The data can then be plotted using the command

```
plot(y, type='p', ylab='IID Normal Data')
```

Try the `type='o'` option in the above command. Which plotting option do you find better to see the randomness in the data? Notice that executing the command `y=rnorm(48)` again will yield a different time series realization of the random process. The `set.seed` command discussed below addresses the issue of how to make simulations in R "reproducible."

Data can be simulated from other distributions. For example, the command `rt(n=48,df=5)` simulates 48 independent observations from a *t*-distribution with 5 degrees of freedom. Similarly, `rchisq(n=48,df=2)` simulates a realization of size 48 from the chi-square distribution with 2 degrees of freedom.

Chapter 2 R Commands

We show some R code to simulate your own random walk with, say, 60 independent standard normal errors.

```
# Exhibit 2.1 on page 14.
n=60
```

This assigns the value of 60 to the object named `n`.

```
set.seed(12345)
```

This initializes the random number generator so that the simulation is reproducible if needed.

```
sim.random.walk=ts(cumsum(rnorm(n)),freq=1,start=1)
```

The expression `rnorm(n)` generates `n` independent values from the standard normal distribution. The function `cumsum` then computes the vector of cumulative sums of the normally distributed sample, resulting in a random walk realization. The random walk realization is then given the attribute of a time series and saved into the object named `sim.random.walk`.

```
plot(sim.random.walk,type='o',ylab='Another Random Walk')
```

plots the simulated random walk.

Chapter 3 R Commands

We now move to discuss some of the R commands appearing in Chapter 3.

```
# Exhibit 3.1 on page 31.
data(rwalk)
```

This command loads the time series `rwalk`, which is a random walk realization.

```
model1=lm(rwalk~time(rwalk))
```

The function `lm` fits a *linear model* (a regression model) with its first argument being a formula. A formula is an expression including a tilde sign (~), the left-hand side of which is the response variable and the right-hand side are the covariates or explanatory variables (separated by plus signs if there are two or more covariates). By default, the intercept term is included in the model. The intercept can be removed by including the term “−1” on the right-hand side of the tilde sign. Recall that `time(rwalk)` yields a time series of the time epochs at which the random walk was sampled. So the command `lm(rwalk~time(rwalk))` fits a time trend regression model to the `rwalk` series. The model fit is saved as the object named `model1`.

```
summary(model1)
```

The function `summary` prints out a summary of the fitted model passed to it. Hence the command above prints out the fitted time trend regression model for `rwalk`.

```
# Exhibit 3.2 on page 31.
plot(rwalk,type='o',ylab='y')
abline(model1)
```

The function `abline` is a low-level graphics function. If a fitted simple regression model is passed to it, it adds the fitted straight line to an existing graph. Any straight line of the form $y = \beta_0 + \beta_1 x$ can be superimposed on the graph by running the command

```
abline(a=beta0,b=beta1)
```

For example, the following command adds a 45 degree line on the current graph.

```
abline(a=0,b=1)
```

Recall the `lm` function can fit multiple regression models, with the covariates or explanatory variables specified one by one, on the right side of the tilde sign (~) in the formula. The covariates must be separated with a plus sign (+). Suppose we want to fit a quadratic time trend model to the rwalk series. We need to create a new covariate that contains the square of the time indices. The quadratic variable may be created before invoking the `lm` function. Or it may be created on the fly when invoking the `lm` function. The latter approach is illustrated here.

```
model1a=lm(rwalk~time(rwalk)+I(time(rwalk)^2))
```

Notice that the expression `time(rwalk)^2` is enclosed within the `I` function which instructs R to create a new variable by executing the command passed into the `I` function. The fitted quadratic trend model can be inspected with the summary function.

```
> summary(model1a)
Call:
lm(formula = rwalk ~ time(rwalk) + I(time(rwalk)^2))
Residuals:
      Min        1Q    Median        3Q       Max
-2.696232 -0.768018  0.008256  0.853365  2.344685
Coefficients:
                   Estimate Std. Error t value Pr(>|t|)
(Intercept)      -1.4272911  0.4534893  -3.147  0.00262 **
time(rwalk)       0.1746746  0.0343028   5.092 4.16e-06 ***
I(time(rwalk)^2) -0.0006654  0.0005451  -1.221  0.22721
---
```

```
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
Residual standard error: 1.132 on 57 degrees of freedom
Multiple R-Squared: 0.8167, Adjusted R-squared: 0.8102
F-statistic:   127 on 2 and 57 DF,  p-value: < 2.2e-16
```

The `summary` function repeats the function call to the `lm` function. It then prints out the five-number numerical summary of the residuals, followed by a table of the parameter estimates with their standard errors, *t*-values and *p*-values. All significant covariates are marked with asterisks (*); more asterisks means higher significance, that is, smaller *p*-value, as explained in the line labeled as Signif. codes. Finally, it outputs the residual standard error, that is, the noise standard deviation estimate, and the multiple R-squared of the fitted model. Clearly, the quadratic term is not significant so that it is not needed, as is also obvious from the time plot of the series.

The reader may wonder why the `I` function is needed. This is because without the `I` function, R interprets the term `time(rwalk)+time(rwalk)^2` using the formula convention (run `?formula` to learn more about the formula convention), which results in fitting the linear trend model! Refit the quadratic trend model but now omit the `I` function in the R command, and compare the model fit with those of the linear and quadratic trend models.

```
# Exhibit 3.3 on page 32.
data(tempdub)
```

This loads the `tempdub` series. You can learn more about the dataset `tempdub` by running the command `?tempdub`.

```
month.=season(tempdub)
```

The expression `season(tempdub)` outputs the monthly index of `tempdub` as a `factor`, and saves it into the object `month.`. The first period sign (.) is part of the name (`month.`) and is included to make the printout from later commands more clear.

We now digress to explain what a factor is. A `factor` is a kind of data structure for handling qualitative (nominal) data that do not have a natural ordering like numbers do. However, for purposes of summary and graphics, the user may supply the `levels` argument to indicate an ordering among the factor values. For example, the following command creates a factor containing the qualitative variable `sex`, with the default ordering using the dictionary order.

```
> sex=factor(c('M','F','M','M','F'))
> sex
[1] M F M M F
Levels: F M
```

We can change the ordering as follows:

```
> sex=factor(c('M','F','M','M','F'),levels=c('M','F'))
> sex
[1] M F M M F
Levels: M F
```

Note the swap of F and M in the levels. The function `table` counts the frequencies of the two sexes.

```
> table(sex)
sex M F
    3 2
```

The printout lists the frequencies of the values according to the order supplied in the level argument. Now, we return to the R scripts in Chapter 3.

```
model2=lm(tempdub~month.-1)
```

Recall that `month` is a factor containing the month of the data. When a formula contains a factor covariate, the function `lm` replaces the factor variable by a set of indicator variables corresponding to each distinct level (value) of the factor. Here, `month.` has 12 distinct levels: Jan, Feb,..., and so forth. So, in place of `month.`, `lm` creates 12 monthly indicator variables and replaces `month.` by the 12 indicator variables. Because these 12 indicator variables are linearly dependent (they add up to a vector of all ones), the intercept term has to be removed to avoid multicollinearity. The expression "`-1`" in the formula takes care of this. The fitted model corresponds to fitting a mean separately for each month. If the expression "`-1`" is omitted, `lm` deals with the multicollinearity by omitting the first indicator variable; that is, the indicator variable for January will be deleted. In such a fitted model, the intercept represents the overall January mean and the coefficients for other months are the deviations of their means from the January mean.

```
summary(model2)
```

A summary of the fitted regression model is printed out with this command. Many variables derived from the fitted model can also be easily obtained. For example, the fitted values can be printed as

```
fitted(model2)
```

whereas residuals are obtained by using

```
residuals(model2)
```

```
# Exhibit 3.4 on page 33.
model3=lm(tempdub~month.) # intercept is automatically
   included so one month (January) is dropped
summary(model3)
```

```
# Exhibit 3.5 on page 35.
har.=harmonic(tempdub,1)
```

The first pair of harmonic functions (sine and cosine pairs) can be constructed by the `harmonic` function, which takes a time series as its first argument and the number of harmonic pairs as its second argument. Run `?harmonic` to learn more about this function. The output of the harmonic function is a matrix that is saved into an object named `har.`. Again, the first period is part of the name and included to make the later printouts clearer.

```
model4=lm(tempdub~har.)
summary(model4)
```

We now briefly discuss the use of matrices in R. A matrix is a rectangular array of numbers. It can be created by the `matrix` function. Here is an example:

```
> M=matrix(1:6,ncol=2)
> M
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

The `matrix` function expects a vector as its first argument, and it uses the values in the supplied vector to fill up a matrix column by column. The column dimension of a matrix is specified by the `ncol` argument and the row dimension by the `nrow` argument. The expression `1:6` stands for the vector containing the integers from 1 to 6. So the matrix function creates a matrix consisting of two columns using the six numbers 1, 2, 3, 4, 5, and 6. Since the row dimension is missing, R assumes that the matrix has six elements and hence the missing row dimension is set to 2. The dimensions of a matrix can be extracted using the `dim` function.

```
> dim(M)
[1] 3 2
```

This displays the row and column dimensions of M as a vector. The function `apply` can process a matrix column by column, with each column operated by a supplied function. For example, the column means of M can be computed as follows:

```
> apply(M,2,mean)
[1] 2 5
```

The first argument of the `apply` function is the matrix on which it processes, and the second argument is `MARGIN`, which should be set to 1 for row processing or 2 for column processing. The third argument is `FUN`, which takes the user-specified function. The example above instructs R to process M column by column and apply the mean function to each column. How would you modify the preceding R command to compute the row sums of M?

```
# Exhibit 3.6 on page 35.
plot(ts(fitted(model4),freq=12,start=c(1964,1)),
   ylab='Temperature',type='l',
   ylim=range(c(fitted(model4),tempdub)))
points(tempdub)
```

The `ylim` option ensures that the *y*-axis has a range that includes both the raw data and the fitted values.

```
# Exhibit 3.8 on page 43.
plot(y=rstudent(model3),x=as.vector(time(tempdub)),
   xlab='Time', ylab='Standardized Residuals',type='o')
```

The expression `rstudent(model3)` returns the (externally) Studentized residuals from the fitted model. To compute the (internally) standardized residuals, use the command `rstandard(model3)`.

```
# Exhibit 3.11 on page 45.
hist(rstudent(model3),xlab='Standardized Residuals')
```

The function `hist` draws a histogram of the data passed to it as the first argument. Note that the default heading of the histogram says that the plot is a histogram of `rstudent(model3)`. While the default main label correctly depicts what is plotted, it is often desirable to have a less technical but more descriptive label; for example, setting the option `main='Histogram of the Standardized Residuals'`.

```
# Exhibit 3.12 on page 45.
qqnorm(rstudent(model3))
```

The expression `rstudent(model3)` extracts the standardized residuals of `model3`. The `qqnorm` function then plots the Q-Q normal scores plot of the residuals. A reference straight line can be superimposed on the Q-Q normal score plot by running the command `qqline(rstudent(model3))`.

```
# Exhibit 3.13 on page 47.
acf(rstudent(model3))
```

The `acf` function computes the sample autocorrelation function of the time series supplied to the function. The maximum number of lags is determined automatically based on the sample size. It can, however, be changed to, say, 30 by setting the option `max.lag=30` when calling the function.

The Shapiro-Wilk test and the runs test on the residuals can be carried out respectively by the following commands.

```
shapiro.test(rstudent(model3))
runs(rstudent(model3))
```

These commands compute the test statistics as well as their corresponding *p*-values.

Chapter 4 R Commands

```
# Exhibit 4.2 on page 59.
data(ma1.2.s)
plot(ma1.2.s,ylab=expression(Y[t]),type='o')
```

The software R can display mathematical symbols in a graph. The option `ylab=expression(Y[t])` specifies that the *y* label is *Y* with *t* as its subscript, all in math font. Typesetting a formula does require some additional work. Read the help pages for `legend` (`?legend`) and run the command `demo(mathplot)` to learn more about this topic.

An MA(1) series with MA coefficient equal to $\theta_1 = -0.9$ and of length $n = 100$ can be simulated by the following commands.

```
set.seed(12345)
```

This command initializes the seed of the random number generator so that a simulation can be reproduced if needed. Without this command, the random generator will initialize “randomly,” and there is no way to reproduce the simulation. The argument `12345` can be replaced by other numbers to obtain different random numbers.

```
y=arima.sim(model=list(ma=-c(-0.9)),n=100)
```

The `arima.sim` function simulates a time series from a given ARIMA model passed into the function as a list that contains the AR and MA parameters as vectors. The simulated model above is an MA(1) model, so there is no AR part in the model list. The software R uses a plus convention in parameterizing the MA part, so we have to add a minus sign before the vector of MA values to agree with our parameterization. The sample size is determined by the value of the argument *n*. So, the command above instructs R to simulate a realization of size 100 from an MA(1) model with $\theta_1 = -0.9$.

We now digress to explain some pertinent facts about `list`. A list is the most flexible data structure in R. You may think of a list as a cabinet with many drawers (elements or components), each of which contains data with possibly different data structures. For example, an element of a list can be another list! The elements of a list are ordered according to the order they are entered. Also, elements can be named to facilitate their easy retrieval. A list can be created by the `list` function with elements supplied as its arguments. The elements may be passed into the `list` function in the form of `name = value`, delimited by commas. Below is an example of a list containing three elements named `a`, `b`, and `c`, where `a` is a three-dimensional vector, `b` is a number, and `c` is a time series.

```
> list1=list(a=c(1,2,3),b=4,c=ts(c(5,6,7,8),
   start=c(2006,2),frequency=4))
> list1
$a
[1] 1 2 3
$b
[1] 4
$c   Qtr1 Qtr2 Qtr3 Qtr4
2006         5    6    7
2007    8
```

To retrieve an element of a list, run the command `listname$elementname`, for example

```
> list1$c
     Qtr1 Qtr2 Qtr3 Qtr4
2006         5    6    7
2007    8
```

Data of irregular structure can be stored as a list. The output of a function is often a list. Simply entering the name of a list may result in dazzling output if the printed list is large. An alternative is to first explore the structure of a list by the function `str` (`str` stands for *structure*). An example follows.

```
> str(list1)
List of 3
 $ a: num [1:3] 1 2 3
 $ b: num 4
 $ c: Time-Series [1:4] from 2006 to 2007: 5 6 7 8
```

This shows that `list1` has three elements and describes these elements briefly.

Chapter 5 R Commands

```
# Exhibit 5.4 on page 91.
plot(diff(log(oil.price)),ylab='Change in Log(Price)',
   type='l')
```

The function `diff` outputs the first difference of the supplied time series. Higher-order differences can be computed by supplying the `differences` argument. For example, the second difference of `log(oil.price)` can be computed by the command

```
diff(log(oil.price), differences=2)
```

A useful convention of R is that the name of an argument in a function can be abbreviated if it does not result in ambiguity. For example, the previous command can be shortened to

```
diff(log(oil.price),diff=2)
```

Note that the second argument of the `diff` function is the lag argument. By default, `lag=1` and the `diff` function computes regular differences—first or higher differences. Later, when we deal with seasonal time series data, it will sometimes be desirable to consider seasonal differences. For example, we may want to subtract this month's number from the number of the same month one year ago; that is, the differences are computed with a lag of 12 months. This can be done by specifying `lag=12`. As an illustration, computing the seasonal differences of period 12 can be done by issuing the command `diff(tempdub,lag=12)`. What will be computed by the command `diff(log(oil.price),2)`? One of the authors (KSC) committed a serious error, more than once, when he tried to compute the second regular differences of some time series by running a similar command with unnamed arguments. Instead of the second regular differences, the first seasonal differences of lag 2 were actually computed by the command with unnamed arguments! Imagine his frustrations of many anxious hours, all because the data analysis from the flawed computations seriously conflicted with expectations based on theory! The moral is that passing unnamed arguments to a function is risky unless you know the positions of the relevant arguments very well. It is well to remember that unnamed arguments, if present, should appear together in the beginning part of the argument list, and there should be no unnamed argument after a named one. Indeed, mixed arguments (some named and some unnamed in a haphazard order) may result in erroneous interpretation by R. The order of the arguments in a function can be quickly checked by running the command `args(function.name)` or `?function.name`, where `function.name` should be replaced by the name of the function you are checking.

```
# Exhibit 5.11 on page 102.
library(MASS)
```

This loads the library MASS. Run the command `library(help=MASS)` to see the content of this library.

```
boxcox(lm(electricity~1))
```

The function `boxcox` computes the maximum likelihood estimate of the power transformation on the response variable to make a linear regression model appropriate for the data. The first argument is a fitted model by the `lm` function. By default, the `boxcox` function produces a plot of the log-likelihood function of the power parameter. The MLE of the power parameter is the value that maximizes the plotted likelihood curve. Here the model is that some power transform of electricity is given by the model of a constant mean plus normally distributed white noise. But we already know that `elec-`

`tricity` is serially correlated, so this method is not entirely correct, as the autocorrelation in the series is not accounted for.

For time series analysis, a more appropriate model is that some power transform of the time series variable follows an AR model. The function `BoxCox.ar` implements this approach. It has two drawbacks in that it is much more computer-intensive and that other covariates cannot be included in the model in the current version of the function. The first argument of `BoxCox.ar` is the name of the time series variable. The AR order may be supplied by the user through the `order` argument. If the AR order is missing, the function estimates the AR order by minimizing the AIC for the log-transformed data. Both `boxcox` and `BoxCox.ar` require the response variable to be positive.

```
BoxCox.ar(electricity)
```

This plots the log-likelihood function of the power parameter for the model that accounts for autocorrelation in the data.

Chapter 6 R Commands

```
# Exhibit 6.9 on page 120.
acf(ma2.s,ci.type='ma',xaxp=c(0,20,10))
```

The argument `ci.type='ma'` instructs R to plot the sample ACF with the confidence band for the kth lag ACF computed based on the assumption of an MA($k - 1$) model. See Equation (6.1.11) on page 112 for details.

```
# Exhibit 6.11 on page 121.
pacf(ar1.s,xaxp=c(0,20,10))
```

This calculates and plots the sample PACF function. Run the command `?par` to learn more about the `xaxp` argument.

```
# Exhibit 6.17 on page 124.
eacf(arma11.s)
```

This computes the sample `EACF` function (extended autocorrelation function) of the data `arma11.s`. The maximum AR and MA orders can be set via the `ar.max` and `ma.max` arguments. Their default values are seven and thirteen, respectively. For example, `eacf(arma11.s,ar.max=10,ma.max=10)` computes the EACF with maximum AR and MA orders of 10. The `EACF` function prints a table of symbols with X standing for a significant value and O a nonsignificant value.

```
library(uroot)
```

This loads the `uroot` library and the following commands illustrate the computation of the Dickey-Fuller unit-root test.

```
ar(diff(rwalk))
```

This command finds the AR order for the differenced series, which is order 8, by the minimum AIC criterion.

```
ADF.test(rwalk,selectlags=list(mode=c(1,2,3,4,5,6,7,8),
   Pmax=8),itsd=c(1,0,0))
```

This computes the ADF test for the data `rwalk`. The `selectlags` argument takes a list as its value. The `mode` argument specifies which lags must be included, and if it is absent, then the `Pmax` argument sets the maximum lag and the `ADF.test` function determines which lags to include in the test using several methods by setting the mode to `signf`, `aic`, or `bic`. The option `signf` is the default value for `mode`, which estimates a subset AR model by retaining only significant lags. The argument `itsd` expects a vector; the first two elements are binary, indicating whether to include a constant term (if the first element is 1) or a linear time trend (if the second element is 1); and the third element zero if there are no more covariates to include in the model. See the help pages for the `ADF.test` function to learn more about it. Hence, the R command instructs `ADF.test` to carry out the test with the null hypothesis that the model has a unit root and an intercept term. The alternative is that the model is stationary, so a small *p*-value implies stationarity!

```
ADF.test(rwalk,selectlags=list(Pmax=0),itsd=c(1,0,0))
```

In comparison, the preceding command carries out the ADF test with the null hypothesis being that the model has a unit root, an intercept but no other lags, whereas the alternative specifies that the model is a stationary AR(1) model with an intercept. If `itsd=c(0,0,0)`, then the alternative model is a centered stationary AR(1) model, that is, with zero mean. Such a hypothesis is not relevant unless the data are already mean-corrected.

```
# Exhibit 6.22 on page 132.
set.seed(92397)
test=arima.sim(model=list(ar=c(rep(0,11),.8),
   ma=c(rep(0,11),0.7)),n=120)
```

This simulates a subset ARMA model. Here `rep(0,11)` stands for a sequence of 11 zeros.

```
res=armasubsets(y=test,nar=14,nma=14,y.name='test',
   ar.method='ols')
```

The `armasubsets` function computes various subset ARMA models, with the maximum AR and MA orders specified by the `nar` and `nma` arguments, both set as 14 in the example above. The associated AR models are estimated by the default method of `ols` (ordinary least squares).

```
plot(res)
```

The `plot` function is a smart function. Seeing that `res` is the output from the `armasubsets` function, it draws a table indicating several of the best subset ARMA models.

Chapter 7 R Commands

Below is a function that computes the method-of-moments estimator of the MA(1) coefficient of an MA(1) model. It is a simple example of an R function. Simply copy and paste it into the R console. Press the enter key to compile the code, and the function `estimate.ma1.mom` will be created and then be available for use in your workspace. This function only exists in the particular workspace where it was created.

```
estimate.ma1.mom=function(x){r=acf(x,plot=F)$acf[1];
   if (abs(r)<0.5) return((-1+sqrt(1-4*r^2))/(2*r))
   else return(NA)}
```

Readers uninterested in the specifics of R programming may skip down to the material on Exhibit 7.1. The syntax of an R function takes the form

```
function.name = function(argument list){function body}
```

where `function body` is a set of R statements (commands). Normally, complete R commands are separated by line breaks. Alternatively, they may be separated by the semicolon symbol (`;`). If an R command is incomplete, R will assume that it is to be continued on the next line and so forth until R reads a complete command. So the function above has a single argument called `x` and contains two commands. The first one is

```
r=acf(x,plot=F)$acf[1]
```

which instructs R to compute the acf of `x` without plotting the values, extract the first element of the computed sample acf function (that is, the lag 1 autocorrelation) and then save it in an object called `r`. The object `r` is a local object; it only exists within the `estimate.ma1.mom` function environment. The second command is

```
if (abs(r)<0.5)
   return((-1+sqrt(1-4*r^2))/(2*r)) else return(NA)
```

Note the line break after the `if` clause and the second half of the command. Since the `if` clause alone is incomplete, R assumes that it is to be continued on the next line. With the second line, R finds a complete R command and so concludes the two lines of commands together as a complete command. In other words, R sees the next command as equivalent to the following one line:

```
if (abs(r)<0.5) return((-1+sqrt(1-4*r^2))/(2*r)) else return(NA)
```

The function `abs` computes the absolute value of the argument passed to it, whereas `sqrt` is the function that computes the square root of its argument. Now, we are ready to interpret the second command: if the absolute value of `r`, the lag 1 autocorrelation of `x`, is less than 0.5 in magnitude, the function returns the number

$$(-1 + \mathrm{sqrt}(1 - 4{*}r\text{^}2))/(2{*}r)$$

which is the method-of-moments estimator of the MA(1) coefficient θ_1; otherwise the function returns `NA` (see Equation (7.1.4) on page 150). The symbol `NA` is the code standing for a missing value in R. (`NA` stands for *not available*.) In this example, R is specifically instructed what value to return to the user. However, the default procedure is that a function returns the value created by the last command in the function body. R provides a powerful computer language for doing statistics. Please consult the documents on the R Website to learn more about R programming.

```
# Exhibit 7.1 on page 152.
data(ma1.2.s)
```

This loads a simulated MA(1) series.

```
estimate.ma1.mom(ma1.2.s)
```

This computes the MA(1) coefficient estimate by the method of moments using the user-created `estime.ma1.mom` function above!

```
data(ar1.s)
```

This loads a simulated AR(1) series from the TSA package.

```
ar(ar1.s,order.max=1,AIC=F,method='yw')
```

This computes the AR coefficient estimates for the `ar1.s` series. The `ar` function estimates the AR model for the centered data (that is, mean-corrected data), so the intercept must be zero and not estimated or printed out in the output. The `ar` function requires the user to specify the maximum AR order through the `order.max` argument. The AR order may be estimated by choosing the order, between 0 and the maximum order, whose model has the smallest AIC. This option can be specified by setting the AIC argument to take the true value, that is, `AIC=T`. Or we can switch off order selection by specifying `AIC=F`. In the latter case, the AR order is set to the maximum AR order. The `ar` function can estimate the AR model using a number of methods, including solving the Yule-Walker equations, ordinary least squares, and maximum likelihood estimation (assuming normally distributed white noise error terms). These correspond to setting the option `method='yw'`, `method='ols'`, or `method='mle'`, respectively. In particular, the preceding R command fits an AR(1) model for the `ar1.s` series by solving the Yule-Walker equation.

We digress briefly to discuss the concept of a logical variable, which can take the value TRUE or FALSE. These values can be abbreviated as T and F. In binary representation, T is also represented by 1 and F by 0. R adopts the useful convention that a logical variable appearing in an arithmetic expression will be automatically converted to 1 if it is a T and 0 otherwise.

```
# Exhibit 7.6, page 165.
data(arma11.s)
arima(arma11.s, order=c(1,0,1),method='CSS')
```

The `arima` function estimates an ARIMA(p,d,q) model for the time series passed to it as the first argument. The ARIMA order is specified by the `order` argument, `order=c(p,d,q)`, so the command above fits an ARMA(1,1) model to the data. Estimation can be carried out by the conditional sum-of-squares method (`method='CSS'`) or maximum likelihood (`method='ML'`). The default estimation method is maximum likelihood, with initial values determined by the CSS method. The `arima` function prints out a summary of the fitted model. The fitted model may also be saved as an object that can be further manipulated, for example, for model diagnostics. By default, if $d = 0$, a stationary ARMA model will be fitted. Also, the fitted model is in the centered form; that is, an ARMA model fitted to the series minus its sample mean. The intercept term reported in the output of the `arima` function is a misnomer, as it is in fact the mean! However, the mean so estimated generally differs slightly from the sample mean.

```
# Exhibit 7.10 on page 168.
res=arima(sqrt(hare),order=c(3,0,0))
```

This saves the fitted AR(3) model in the object named `res`. The output of the `arima` function is a `list`. Run the command `str(res)` to find out what is saved in `res`. You will find that most of the things in `res` are not directly useful. Instead, the output of the `arima` function has to be processed by other functions for more informed summaries. For example, (raw) residuals from the fitted model can be computed by the `residuals` function via the command `residuals(res)`. Fitted values can be obtained by running `fitted(res)`. Other useful functions for processing a fitted ARIMA model from the `arima` function will be discussed below.

The empirical approach of using the bootstrap to do inference is illustrated below.

```
set.seed(12345)
```

This initializes the seed of the random number generator so that the simulation study can be repeated.

```
coefm.cond.norm=arima.boot(res,cond.boot=T,is.normal=T,
   B=1000,init=sqrt(hare))
```

The `arima.boot` function carries out a bootstrap analysis based on a fitted ARIMA model. Its first argument is a fitted ARIMA model, that is, the output from the `arima` function. Four different bootstrap methods are available: The bootstrap series can be initialized by a supplied value (`cond.boot=T`) or not (`cond.boot=F`), and a nonparametric bootstrap (`is.normal=F`) or a parametric bootstrap assuming normal innovations (`is.normal=T`) can be used. For a conditional bootstrap, the initial values can be supplied as a vector (the `arima.boot` function will use the initial values from the supplied vector). The bootstrap sample size, say 1000, is specified by the `B=1000` option. The function `arima.boot` outputs a matrix with each row being the bootstrap estimate of the ARIMA coefficients obtained by maximum likelihood estimation with the bootstrap data. So, if `B=1000` and the model is an AR(3), then the output is a 1000 by 4 matrix where each row consists of the bootstrap AR(1), AR(2), and AR(3) coefficients plus the mean estimate in that order ($\hat{\phi}_1$, $\hat{\phi}_2$, $\hat{\phi}_3$, $\hat{\mu}$).

```
signif(apply(coefm.cond.norm,2,function(x)
   {quantile(x,c(.025,.975),na.rm=T)}),3)
```

This is a compound R statement. It is equivalent to the two commands

```
temp=apply(coefm.cond.norm,2,function(x)
   {quantile (x,c(.025,.975),na.rm=T)})
signif(temp,3)
```

except that the temporary variable `temp` is not created in the original compound statement. Recall that the `apply` function is a general-purpose function for processing a matrix. Here the `apply` function processes the matrix `coefm.cond.norm` column by column, with each column supplied to the no-name user-supplied function

```
function(x){quantile(x,c(.025,.975),na.rm=T)}
```

This no-name function has one input, called `x`, that is processed by the `quantile` function. The `quantile` function takes a vector and computes the sample quantiles with the corresponding probability specified in the second argument. The third argument of the quantile function is specified as `na.rm=T` (`na` stands for not available and `rm` means remove), which means that any missing values in the input are discarded before computing the quantiles. This specification is pivotal because by default any quantile of a dataset with some missing values is defined to be a missing value (`NA`) in R. (Some bootstrap series may have convergence problems upon fitting an ARIMA model and hence the output of the bootstrap function may contain some missing values.) To return to the interpretation of the command on the right-hand side of `temp`, it instructs R to compute the 2.5th and 97.5th percentiles of each bootstrap coefficient estimate. To enable precise calculations, R maintains many significant digits in the numbers stored in an object. The printed version, however, usually requires fewer significant digits for clarity. This can be done by the `signif` function. The `signif` function outputs the object passed into it as first argument, but only to the number of significant digits specified in the second argument, which is three in the example. Altogether, the compound R command computes the 95% bootstrap confidence intervals for each AR coefficient.

Chapter 8 R Commands

```
# Exhibit 8.2 on page 177.
data(hare)
m1.hare=arima(sqrt(hare),order=c(3,0,0))
m1.hare
```

This prints the fitted AR(3) model for the square-root-transformed hare data. The AR(2) coefficient estimate ($\hat{\phi}_2$) turns out not to be significant. Note that the AR(2) coefficient is the second element in the coefficient vector, as shown in the printout of the fitted model. A constrained ARIMA model with some elements fixed at certain values can be fitted by using the `fixed` argument in the `arima` function. The `fixed` argument should be a vector of the same length as the coefficient vector and its elements set to `NA` for all of the free elements but set to zero (or another fixed value) for all of the constrained coefficients. For example, here the AR(2) coefficient is constrained to be zero ($\phi_2 = 0$) and hence `fixed=c(NA,0,NA,NA)`, that is, the AR(1), AR(3), and the ‘‘intercept’’ term are free parameters, whereas the AR(2) is fixed at 0. Remember that the ‘‘intercept’’ term is last. Below is the command for fitting the constrained AR(3) model for the hare data.

```
m2.hare=arima(sqrt(hare),order=c(3,0,0),
   fixed=c(NA,0,NA,NA))
m2.hare
```

Note that the intercept term is actually the mean in the centered form of the ARMA model; that is, if y = sqrt(hare) – intercept, then the model is

$$y_t = 0.919y_{t-1} - 0.5313y_{t-3} + e_t$$

so the “true” estimated intercept equals 5.6889*(1 – 0.919 + 0.5313) = 3.483, as stated in the text!

```
plot(rstandard(m2.hare),
   ylab='Standardized Residuals',type='b')
```

The function `rstandard` computes the standardized residuals; that is, the raw residuals normalized by the estimated noise standard deviation.

```
abline(h=0)
```

adds a horizontal line to the plot with zero y-intercept. Use the help in R to find out how to add a vertical line with *x*-intercept = 10.

```
# Exhibit 8.12 on page 185 (prefaced by some commands in
   Exhibit 8.1 on page 176)
data(color)
m1.color=arima(color,order=c(1,0,0))
tsdiag(m1.color,gof=15,omit.initial=F)
```

The `tsdiag` function in the TSA package has been modified from that in the `stats` package of R. It performs model diagnostics on a fitted model. The argument `gof` specifies the maximum number of lags in the `acf` function used in the model diagnostics. Setting the argument `omit.initial=T` omits the few initial residuals from the analysis. This option is especially useful for checking seasonal models where the initial residuals are close to zero by construction and including them may skew the model diagnostics. In the example, the `omit.initial` argument is set to be `F` so that the diagnostics are done with all residuals. Recall that the Ljung-Box (portmanteau) test statistic equals the weighted sum of the squared residual autocorrelations from lags 1 to `K`, say; see Equation (8.1.12) on page 184. Assuming that the ARIMA orders are correctly specified, the validity of the approximate chi-square distribution for the Ljung-Box test statistic requires that `K` be larger than the lag beyond which the original time series has negligible autocorrelation. The modified `tsdiag` function in the TSA package checks this requirement; consequently the Ljung-Box test is only computed for sufficiently large `K`. If the required `K` is larger than the specified maximum lag, `tsdiag` will return an error message. This problem can be solved by increasing the maximum lag asked for. Use `?tsdiag` to learn more about the modified `tsdiag` function.

Chapter 9 R Commands

```
# Exhibit 9.2 on page 205.
data(tempdub)
   tempdub1=ts(c(tempdub,rep(NA,24)),start=start(tempdub),
   freq=frequency(tempdub))
```

This appends two years of missing values to the `tempdub` data, as we want to forecast the temperature for two years into the future. The function `start` extracts the starting date of a time series. The function `frequency` extracts the frequency of the time series passed to it, here being 12. Hence, `tempdub1` contains the Dubuque temperature series augmented by two years of missing data, with the same starting date and frequency of sampling per unit time interval.

```
har.=harmonic(tempdub,1)
```

This creates the first pair of harmonic functions.

```
m5.tempdub=arima(tempdub,order=c(0,0,0),xreg=har.)
```

This fits the harmonic regression model using the `arima` function. The covariates are passed to the function through the `xreg` argument. In the example, `har.` is the covariate and the `arima` function fits a linear regression model of the response variable on the covariate, with the errors assumed to follow an ARIMA model. Because the specified ARIMA orders $p = d = q = 0$, the presumed error structure is white noise; that is, the `arima` function fits an ordinary linear regression model of `tempdub` on the first pair of harmonic functions. Note that the result is the same as that from the fit using the `lm` function, which can be verified by the following commands:

```
har.=harmonic(tempdub,1); model4=lm(tempdub~har.)
summary(model4)
```

The `xreg` argument expects the covariate input either as a matrix or a `data.frame`. A `data.frame` can be thought of as a matrix made up by binding together several covariates column by column. It can be created by the `data.frame` function with multiple arguments, each of which takes the form `covariate.name = R statement` for computing the covariate. If the `covariate.name` is omitted, the `R statement` becomes the covariate name, which may be undesirable for a complex defining statement. If the R statement is a matrix, its columns are taken as covariates with the column names taken as the covariate names. Consider the example of augmenting the harmonic regression model above by a linear time trend. The augmented model can be fitted by the command

```
arima(tempdub,order=c(0,0,0),
   xreg=data.frame(har.,trend=time(tempdub)))
```

```
m5.tempdub
```

This prints the fitted model.

We now illustrate prediction with an example.

```
newhar.=harmonic(ts(rep(1,24), start=c(1976,1),freq=12),1)
```

This creates the harmonic functions over two years starting from January 1976. Remember that the `tempdub` series ends in December 1975.

```
plot(m5.tempdub,n.ahead=24,n1=c(1972,1),newxreg=newhar.,
   col='red', type='b',ylab='Temperature',xlab='Year')
```

This computes and plots the forecasts based on the fitted model passed as the first argument. Here, we specify a forecast for 24 steps ahead through the argument `n.ahead=24`. The covariate values over the period of forecast have to be supplied by the `newxreg` argument. The `newxreg` argument should match the `xreg` argument in terms of the covariates except that their values are from different periods. The plot may be drawn with a starting date different from the start date of the time series data by using the `n1` argument. Here, `n1=c(1972,1)` specifies January 1972 as the start date for the plot. For nonseasonal data (that is, frequency = 1), `n1` should be a scalar. The `col` and `type` arguments refer to the color and style of the plotted lines.

```
# Exhibit 9.3 on page 206.
data(color)
m1.color=arima(color,order=c(1,0,0))
```

```
plot(m1.color,n.ahead=12,col='red',type='b',xlab='Year',
   ylab='Temperature')
abline(h=coef(m1.color)
   [names(coef(m1.color))=='intercept'])
```

The final command adds the horizontal line at the estimated mean (intercept). This is a complex statement. The expression `coef(m1.color)` extracts the coefficient vector. The components of the coefficient vector are named. The names of a vector can be extracted by the names function, so `names(coef(m1.color))` returns the vector of names of the components of the coefficient vector. The == operator compares the two vectors on its two sides element by element, resulting in a vector consisting of TRUEs and FALSEs depending on whether the elements are equal or not. (If the vectors under comparison are of unequal length, R recycles the shorter one repeatedly to match the longer one.) Hence, the command

```
[names(coef(m1.color))== 'intercept']
```

returns a vector with the TRUE value in the position in which the “intercept” component lies and with all other elements FALSE. Finally, the intercept coefficient estimate is extracted by the “bracket” operation:

```
coef(m1.color)[names(coef(m1.color))=='intercept']
```

The operation within brackets subsets a vector using one of two mechanisms. Let `v` be a vector. A subvector of it can be formed by the command `v[s]`, where `s` is a Boolean vector, (that is, consisting of TRUEs and FALSEs) that is of the same length as `v`. The vector `v[s]` is then a sub-vector of `v` consisting of those elements of `v` for which the corresponding element in `s` is TRUE; elements in `v` whose corresponding element in `s` is FALSE are discarded from `v[s]`.

A second way to subset a vector is to construct `s` so that it contains the position of the elements to be retained and `v[s]` will return the desired subvector. A variation of this approach is to form a subvector by deletion. Unwanted elements are designated by giving their positions multiplied by `-1`. An illustration follows.

```
> v=1:5
```

This creates a vector containing the first five positive integers.

```
> v
[1] 1 2 3 4 5
```

```
> names(v)
  NULL
```

By default, the components of `v` are unnamed, so `names(v)` returns an empty vector denoted by the object `NULL`.

```
> names(v)=c('A','B','C','D','E')
```

This is the method of assigning names to the components of a vector.

```
> v
 A B C D E
 1 2 3 4 5
```

The command

```
> names(v)=='C'
```

```
 [1] FALSE FALSE TRUE FALSE FALSE
```

finds which components of `names(v)` is “C.”
The command

```
> v[names(v)=='C']
 C
 3
```

subsets `v` by Boolean extraction.
The command

```
> v[3]
 C
 3
```

subsets `v` by supplying the positions of the retained elements.
The command

```
> v[-3]
 A B D E
 1 2 4 5
```

subsets `v` by supplying the positions of the unwanted elements.

Chapter 10 R Commands

The theoretical ACF of a stationary ARMA process can be computed by the `ARMAacf` function. The ar parameter vector, if present, is to be passed into the function via the `ar` argument. Similarly, the ma parameter vector is passed into the function via the `ma` argument. The maximum lag may be specified by the `lag.max` argument. Setting the `pacf` argument to `TRUE` computes the theoretical pacf; otherwise the function computes the theoretical acf. Consider as an example the seasonal MA model:

$$Y_t = (1 + 0.5B)(1 + 0.8B^{12})e_t$$

Note that $(1 + 0.5B)(1 + 0.8B^{12}) = (1 + 0.5B + 0.8B^{12} + 0.4B^{13})$ so the ma coefficients are specified by the option `ma=c(0.5,rep(0,10),0.8,0.4)`. Its theoretical ACF is displayed on the left side of Exhibit 10.3, which can be done by the following R commands.

```
plot(y=ARMAacf(ma=c(0.5,rep(0,10),0.8,0.4),
   lag.max=13)[-1],x=1:13,type='h',
xlab='Lag k',ylab=expression(rho[k]),axes=F,ylim=c(0,0.6))
points(y=ARMAacf(ma=c(0.5,rep(0,10),0.8,0.4),
   lag.max=13)[-1],x=1:13,pch=20)
abline(h=0)
axis(1,at=1:13,
   labels=c(1,NA,3,NA,5,NA,7,NA,9,NA,11,NA,13))
axis(2)
text(x=7,y=.5,labels=expression(list(theta&=&-0.5,
   Theta&=&-0.8)))
```

As the labeling of the figure requires Greek alphabets and subscripts, the label information has to be passed via the expression function. Run the help menu

`?plotmath` to learn more about how to do mathematical annotations in R.

```
# Exhibit 10.10 on page 237
m1.co2=arima(co2,order=c(0,1,1),
   seasonal=list(order=c(0,1,1),period=12))
```

The argument `seasonal` supplies the information on the seasonal part of the seasonal ARIMA model. It expects a `list` with the seasonal order supplied in the component named `order` and the seasonal period entered via the `period` component, so the command above instructs the `arima` function to fit a seasonal ARIMA $(0,1,1) \times (0,1,1)_{12}$ model to the `co2` series.

```
m1.co2
```

This prints a summary of the fitted seasonal ARIMA model.

Chapter 11 R Commands

```
# Exhibit 11.5 on page 255.
acf(as.vector(diff(diff(window(log(airmiles),
   end=c(2001,8)),12))),lag.max=48)
```

The expression `window(log(airmiles),end=c(2001,8))` subsets the `log(airmiles)` time series by specifying a new end date of August 2001. The sub-time series is first seasonally differenced with lag 12 and then regularly differenced. The doubly differenced series is then passed to the `acf` function for computing the sample ACF out to 48 lags.

```
# Exhibit 11.6 on page 255.
air.m1=arimax(log(airmiles),order=c(0,1,1),seasonal=
   list(order=c(0,1,1),period=12),
   xtransf=data.frame(I911=1*(seq(airmiles)==69),
   I911=1*(seq(airmiles)==69)),
   transfer=list(c(0,0),c(1,0)),
   xreg=data.frame(Dec96=1*(seq(airmiles)==12),
   Jan97=1*(seq(airmiles)==13),
   Dec02=1*(seq(airmiles)==84)),method='ML')
```

The `arimax` function extends the `arima` function so that it can handle intervention analysis and outliers (both AO and IO) in time series. It is assumed that the intervention affects the mean function of the process, with the deviation from the unperturbed mean function modeled as the sum of the outputs of an ARMA filter of a number of covariates; the deviation is known as the transfer function. The covariates making up the transfer function are passed to the `arimax` function via the `xtransf` argument in the form of a matrix or a `data.frame`. For each such covariate, its contribution to the transfer function takes the form of a dynamic response given by

$$\frac{(a_0 + a_1 B + \cdots + a_q B^q)}{(1 - b_1 B - b_2 B^2 - \cdots - b_p B^p)} covariate_t$$

The transfer function is the sum of the dynamic responses, in the form of some ARMA filter, of all covariates in the `xtransf` argument. The ARMA order of the filter is denoted by the vector `c(p,q)`. If p = q = 0 (that is, `c(p,q) = c(0,0)`), the contribution of the covariate is of the form $a_0 covariate_t$. If `c(p,q) = c(1,0)`, the output becomes

$$\frac{a_0}{(1-b_1B)}covariate_t = a_0(covariate_t + b_1covariate_{t-1} + b_1^2covariate_{t-2} + \cdots)$$

The ARMA orders for the dynamic components of the transfer function are supplied via the `transf` argument as a `list` containing the vectors of ARMA orders in the order of the covariates defined in the `xtransf` argument. Hence, the options:

```
xtransf=data.frame(I911=1*(seq(airmiles)==69),
   I911=1*(seq(airmiles)==69)),
   transfer=list(c(0,0),c(1,0))
```

instruct the `arimax` function to create two identical covariates called `I911`, which is an indicator variable, say P_t, that equals 1 in September 2001 and 0 otherwise, and the transfer function is the sum of two ARMA filters of the `9/11` indicator variable of orders c(0,0) and c(1,0) respectively. Hence the transfer function equals

$$\omega_0 P_t + \frac{\omega_1}{(1-\omega_2B)}P_t$$

This is equivalent to an ARMA(1,1) filter of the form

$$\frac{\{(\omega_0+\omega_1)-\omega_0\omega_2B\}}{(1-\omega_2B)}P_t$$

which can be specified by the following options

```
xtransf=data.frame(I911=1*(seq(airmiles)==69)),
   transfer=list(c(1,1))
```

Additive outliers (AO) in a time series can be incorporated as indicator variables passed to the `xreg` argument. For example, three potential AOs are included in the model by the following supplied argument:

```
xreg=data.frame(Dec96=1*(seq(airmiles)==12),
   Jan97=1*(seq(airmiles)==13),
   Dec02=1*(seq(airmiles)==84))
```

Note that the first potential outlier occurs in December 1996. The corresponding indicator variable is labeled as `Dec96` and is computed by the formula `1*(seq(airmiles)==12)`, which results in a vector that equals 0 except its twelfth element, which equals 1, and the vector is of the same length as `airmiles`. Some specifics of this "simple" command follow. The function `seq` creates a vector consisting of the first n positive integers, where n is the length of the vector passed to the `seq` function. The expression `seq(airmiles)==12` creates a vector of the same length as `airmiles`, and its elements are all FALSE except that the twelfth element is TRUE. Then `1*(seq(airmiles)==12)` is an arithmetic expression for which R automatically converts any imbedded Boolean vector `(seq(airmiles)==12)` to a binary vector. Recall that the TRUE values are converted to 1s and the FALSE values to 0s.

Multiplying by 1 does not alter the converted binary vector. Indeed, multiplication is employed to trigger the conversion from the Boolean values to binary values.

For this example, the unperturbed process is assumed to be an IMA(1,1) process, as is evident from the supplied argument `order=c(0,1,1)`. In general, a seasonal ARIMA unperturbed process is specified in the same way that it is specified for the `arima` function.

`air.m1`

This prints out the fitted intervention model, as displayed below.

```
> air.m1
Call: arimax(x=log(airmiles),order=c(0,1,1),seasonal=
   list(order=c(0,1,1),period=12),xreg=data.frame(Dec96=
   1*(seq(airmiles)==12),Jan97=1*(seq(airmiles)==13),
   Dec02=1*(seq(airmiles)==84)),method='ML',
   xtransf=data.frame(I911=1*(seq(airmiles)==69),I911=1*
   (seq(airmiles)==69)),transfer=list(c(0,0),c(1,0)))

Coefficients:
         ma1     sma1   Dec96    Jan97   Dec02  I911-MA0  I911.1-AR1  I911.1-MA0
     -0.3825  -0.6499  0.0989  -0.0690  0.0810   -0.0949      0.8139     -0.2715
s.e.  0.0926   0.1189  0.0228   0.0218  0.0202    0.0462      0.0978      0.0439
sigma^2 estimated as 0.000672: log likelihood=219.99, aic=-423.98
```

Note that the parameter in the transfer-function component defined by the first instance of the indicator variable `I911` is labeled as `I911-MA0`; that is, the MA(0) coefficient. The transfer-function components defined by the second instance of the indicator variable `I911` are labeled as `I911.1-AR1` and `I911.1-MA0`. These are the AR(1) and MA(0) coefficient estimates.

We can also try the equivalent parameterization of specifying an ARMA(1,1) filter on the `9/11` indicator variable.

```
> air.m1a=arimax(log(airmiles),order=c(0,1,1),
   seasonal=list(order=c(0,1,1),period=12),
   xtransf=data.frame(I911=1*(seq(airmiles)==69)),
   transfer=list(c(1,1)),
   xreg=data.frame(Dec96=1*(seq(airmiles)==12),
   Jan97=1*(seq(airmiles)==13),
   Dec02=1*(seq(airmiles)==84)),method='ML')
> air.m1a
Call: arimax(x=log(airmiles),order=c(0,1,1),seasonal=
   list(order=c(0,1,1),period=12),xreg=data.frame(Dec96=1
   *(seq(airmiles)==12),Jan97=1*(seq(airmiles)==13),Dec02=
   1*(seq(airmiles)==84)),method='ML',xtransf=
   data.frame(I911=1*(seq(airmiles)==69)),transfer=
   list(c(1,1)))

Coefficients:
         ma1     sma1   Dec96    Jan97   Dec02  I911-AR1  I911-MA0  I911-MA1
     -0.3601  -0.6130  0.0949  -0.0840  0.0802    0.8094   -0.3660    0.0741
s.e.  0.0926   0.1261  0.0222   0.0229  0.0194    0.0924    0.0233    0.0424
sigma^2 estimated as 0.000648: log likelihood=221.76, aic=-427.52
```

Note that the parameter estimates of this model are similar to those of the previous model but this model has a better fit, which may happen as the optimization is done numerically.

```
# Exhibit 11.8 on page 256.
Nine11p=1*(seq(airmiles)==69)
```

This defines the `9/11` indicator variable.

```
plot(ts(Nine11p*(-0.0949)+ filter(Nine11p,filter=.8139,
   method='recursive',side=1)*(-0.2715),
   frequency=12,start=1996),type='h',ylab='9/11 Effects')
```

The command

```
Nine11p*(-0.0949)+filter(Nine11p,filter=.8139,
   method='recursive',side=1)*(-0.2715)
```

computes the estimated transfer function. Note that the command

```
filter(Nine11p,filter=.8139,method='recursive',side=1)
```

computes `(1-0.8139*B)Nine11p`. The function `filter` performs an MA or AR filtering on the input sequence passed to it as the first argument. Suppose the input is a vector $\boldsymbol{x} = c(x_1,\mathrm{x}_2,\ldots,x_n)$. Then the output $\boldsymbol{y} = c(y_1,y_2,\ldots,y_n)$ defined by the MA filter

$$y_t = c_0x_t + c_1x_{t-1} + \cdots + c_qx_{t-q}$$

can be computed by the command

```
filter(x,filter=c(c0,c1,...,cq),side=1).
```

The argument `side=1` specifies that the MA operator works on current and past values when computing an output value. To compute `y1`, the value of `x0` is needed. Since the latter is not observed, the filter sets it to `NA`, and hence `y1` is also `NA`. In this case, `y2`, `y3`, and so forth can be computed. For an AR filtering with the output defined recursively by the equation

$$y_t = x_t + c_1y_{t-1} + \ldots + c_py_{t-p}$$

the R command is

```
filter(x,filter=c(c1,c2,...,cp),method='recursive',
   side=1)
```

Note that, unlike the case of the MA filter, the filter vector starts with c1 and there is no c0 in the equation. The argument `method='recursive'` signifies an AR type of filtering. For the AR filter, the initial values cannot be set to `NA`, lest all output values be `NA`! The default initial values are zeros although other initial values may be specified via the `init` argument.

```
abline(h=0)
```

adds a horizontal line with zero *y*-intercept.

```
# Exhibit 11.9 on page 259.
set.seed(12345)
y=arima.sim(model=list(ar=.8,ma=.5),n.start=158,n=100)
```

This simulates an ARMA(1,1) series of sample size 100. To remove transient effects of the initial values, a burn-in of size 158 is specified. A large burn-in of the order of hundreds should generally ensure that the simulated process is approximately stationary. The number 158 is chosen for no particular good reason.

```
y[10]
```

This prints out the tenth simulated value.

```
y[10]=10
```

This alters the tenth value to be 10; that is, it becomes an additive outlier, mimicking the effect of a clerical recording mistake, for example!

```
y=ts(y,freq=1,start=1); plot(y,type='o')
acf(y)
pacf(y)
eacf(y)
```

This exploratory analysis suggests an AR(1) model.

```
m1=arima(y,order=c(1,0,0)); m1; detectAO(m1)
```

This detects the presence of any additive outliers (AO) in the fitted AR(1) model. The test requires an estimate of the standard deviation of the error (innovation) term, which by default is estimated by a robust estimation scheme, resulting in a more powerful test. The robust estimation scheme can be switched off by the argument `robust=F`, as illustrated in the command below.

```
detectAO(m1, robust=F)
```

This verifies that a nonrobust procedure is less powerful.

```
detectIO(m1)
```

This detects the presence of any innovative outliers (IO) in the fitted AR(1) model. As an AO is found in the tenth case, it is incorporated as an indicator covariate in the following model.

```
m2=arima(y,order=c(1,0,0),xreg=data.frame(AO=seq(y)==10))
m2
```

```
# Exhibit 11.10 on page 260
data(co2)
m1.co2=arima(co2,order=c(0,1,1),seasonal=list
   (order=c(0,1,1),period=12))
m1.co2
detectAO(m1.co2)
detectIO(m1.co2)
```

As an IO is found in the 57th data case, it is incorporated in the model.

```
m4.co2=arimax(co2,order=c(0,1,1),
   seasonal=list(order=c(0,1,1),period=12),io=c(57))
```

The epochs of IOs are passed to the `arimax` function via the `io` argument, which expects a `list` containing the positions of the IOs either as the time index of the IO or as a vector in the form of `c(year,month)` that gives the year and month of the IO for seasonal data; the latter format also works similarly for seasonal data of other types. For a single IO, it is not necessary to enclose the single vector of index in a list before passing it to the `io` argument.

```
# Exhibit 11.11 on page 262.
set.seed(12345)
X=rnorm(105)
Y=zlag(X,2)+.5*rnorm(105)
```

The command `zlag(X,2)` computes the second lag of `X`.

```
X=ts(X[-(1:5)],start=1,freq=1)
```

This omits the first five values of `X` and converts the remaining values to form a time series.

```
Y=ts(Y[-(1:5)],start=1,freq=1)
ccf(X,Y,ylab='CCF')
```

This computes the cross-correlation function of `X` and `Y`. The `ylab` argument is supplied in lieu of the default *y*-label of the `ccf` function that is "ACF".

```
# Exhibit 11.14 on page 264.
data(milk)
data(electricity)
milk.electricity=ts.intersect(milk,log(electricity))
```

The `ts.intersect` function merges several time series into a matrix (panel) of time series over the time frame where each series has data. The object `milk.electricity` is a matrix of two time series, the first column of which is the milk series and the second the log of electricity, over the time period when these two series overlap.

```
plot(milk.electricity,yax.flip=T)
```

The option `yax.flip=T` flips the label for the *y*-axis for the series alternately so as to make the labeling clearer.

```
# Exhibit 11.15 on page 265.
ccf(milk.electricity[,1],milk.electricity[,2],
   main='milk & electricity',ylab='CCF')
```

The expression `milk.electricity[,1]` extracts the milk series and `milk.electricity[,2]` the log electricity series.

The `as.vector` function strips the time series attribute from the time series. This is done to nullify the default way that the `ccf` function plots the cross-correlations. You may want to repeat the command without the `as.vector` function to see the default labels of the lags according to the period of the data.

```
ccf((milk.electricity[,1]),(milk.electricity[,2]),
   main='milk & electricity',ylab='CCF')
```

The bracket operator extracts a submatrix from a matrix, say `M`, in the form of `M[v1,v2]`, where `v1` indicates which rows are kept and `v2` indicates which columns are retained. Consequently, the submatrix `M[v1,v2]` contains all elements of `M` in the intersection of the retained rows and columns. If `v1` (`v2`) is missing, then all rows (columns) are retained. Hence, `M[,1]` is simply the submatrix consisting of the first column of `M`. However, R adopts the convention that a submatrix with a single row or column is "demoted" to a vector; that is, it loses one dimension. This convention makes sense in most cases. However, if you do matrix algebra in R, this convention may result in strange error messages! To prevent automatic dimension reduction, use `M[v1,v2,drop=F]`. Instead of specifying which rows or columns are to be retained in the submatrix, you can specify which rows or columns are to be deleted by specifying the negative of their positions. Or `v1` (`v2`) can be specified as a Boolean vector, where the positions to be retained (eliminated) are denoted by TRUE (FALSE).

```
# Exhibit 11.16 on page 267.
me.dif=ts.intersect(diff(diff(milk,12)),
   diff(diff(log(electricity),12)))
prewhiten(as.vector(me.dif[,1]),as.vector(me.dif[,2]),
   ylab='CCF')
```

The `prewhiten` function expects two time series input via the `x` and `y` arguments. Both series will be filtered according to an ARIMA model. The ARIMA model can be supplied via the `x.model` argument and should be the output of the `arima` function. If no ARIMA model is supplied, an AR model will be fitted to the *x* series, with the AR order selected by minimizing the AIC. The `prewhiten` function computes and plots the cross-correlation function (CCF) of the residuals of the *x* series and those of the *y* series from the same (supplied or fitted) model.

Chapter 12 R Commands

Below, we show how to implement the Jarque-Bera test for normality in two different ways. First, we show the direct approach.

```
skewness(r.cref)
```

This computes the skewness of the r.cref series.

```
kurtosis(r.cref)
```

This computes the kurtosis of the data.

```
length(r.cref)*skewness(r.cref)^2/6
```

The function `length` returns the length of the vector (time series) passed into it, so the expression above computes the first part of the Jarque-Bera statistic.

```
length(r.cref)*kurtosis(r.cref)^2/24
```

computes the second half of the Jarque-Bera statistic.

```
JB=length(r.cref)*(skewness(r.cref)^2/6 +
   kurtosis(r.cref)^2/24)
```

The object `JB` then contains the Jarque-Bera statistic and the command `JB` prints out the statistic. The command `1-pchisq(JB,df=2)` computes the *p*-value of the Jarque-Bera test for normality. The function `pchisq` computes the cumulative probability of a chi-square distribution being less than or equal to the value in the first argument. The `df` argument of the `pchisq` function specifies the degrees of freedom for the chi-square distribution. Because the *p*-value equals the right tail area, it equals 1 minus the cumulative probability. Besides `pchisq`, other functions associated with the chi-square distribution include `qchisq`, which computes quantiles; `dchisq`, which computes the probability density; and `rchisq`, which simulates realizations from the chi-square distributions. Use Help in R to learn more about these functions. For other probability distributions, similar functions are available. Associated with the normal distributions are `rnorm`, `pnorm`, `dnorm`, and `qnorm`. Check out the usages of the relevant functions for the binomial (binom), Poisson, and other distributions.

```
library(tseries)
```

This loads the `tseries` library, which contains a number of functions needed for the analysis reported in this chapter. Run `library(help=tseries)` for more information about the `tseries` package.

```
jarque.bera.test(r.cref)
```

This carries out the Jargue-Bera test for normality with the time series `r.cref`.

```
# Exhibit 12.9 on page 283.
McLeod.Li.test(y=r.cref)
```

This performs the McLeod-Li test for presence of ARCH in the daily CREF returns. The first two arguments of the function are `object` and `y`, respectively. For the test with raw data, the time series is supplied to the function via the `y` argument. Then, the function computes the Box-Ljung statistics with the autocorrelations of the squared data to detect for conditional heteroscedascity. The test is carried out with the first m autocorrelations of the squared data, with m ranging from 1 to the maximum lag specified by the `gof.lag` argument. If the `gof.lag` argument is missing, the default is set to $n\log_{10}(n)$ where n is the sample size.

The McLeod-Li test can also be applied to residuals from an ARMA model fitted to the data. For example, the US dollar/Hong Kong dollar exchange rate data was found to admit an AR(1) + outlier model. The need for incorporating ARCH in the model for the exchange rate data can be tested by the command

```
McLeod.Li.test(arima(hkrate,order=c(1,0,0),
   xreg=data.frame(outlier1)))
```

Note that object is the first argument so in the above command, the fitted AR(1) + outlier model is passed into the function. The function then computes the test statistics based on the squared residuals from the fitted AR(1) + outlier model. If the object argument is supplied explicitly or implicitly, the `y` argument is ignored by the function even if it is supplied. Remember that to apply the test to raw data, the `y` argument must be supplied and the object argument suppressed.

```
# Exhibit 12.11 on page 286.
set.seed(1235678)
garch01.sim=garch.sim(alpha=c(.01,.9),n=500)
```

The `garch.sim` function simulates a GARCH process, with the ARCH coefficients supplied via the `alpha` argument and the GARCH coefficients via the `beta` argument. The sample size is passed into the function via the `n` argument. In the example above, `alpha=c(.01,.9)` specifies that the constant term is 0.01 and the ARCH(1) coefficient equals 0.9. So, `garch01.sim` saves a realization from an ARCH(1) process.

```
# Exhibit 12.25 on page 300.
m1=garch(x=r.cref,order=c(1,1))
```

This fits a GARCH(1,1) model with the `r.cref` series. The `garch` function estimates a GARCH model by maximum likelihood. The time series is supplied into the function by the `x` argument and the GARCH order by the `order` argument. The `order` takes the form `c(p,q)` where `p` is the GARCH order and `q` the ARCH order.

```
summary(m1)
```

This summarizes the fitted GARCH(1,1) model. Ignore the Box-Ljung test results reported in the summary, as the generalized portmanteau tests should be used; see the book.

```
# Exhibit 12.29 on page 305.
gBox(m1,method='squared')
```

The `gBox` function computes the generalized portmanteau test for checking whether or not there is any residual heteroscedasticity in the residuals of a fitted GARCH model. It requires supplying the fitted GARCH model from the `garch` function through the first argument (the model argument, the first argument of the function). By default, the tests are carried out with the squared residuals from the fitted GARCH model. To inspect absolute residuals, use the option `method='absolute'`. By default, the test is carried out for the ACF for lags from 1 to, say, `K`, where `K` runs from 1 to 20. The collection of `K`'s can be specified by the lags argument. For example, to carry out the test for `K` ranging from 1 to 30, supply the option `lags=1:30`.

```
gBox(m1,lags=20,plot=F,x=r.cref, method='squared')$pvalue
```

prints out the *p*-values of the generalized portmanteau test with the squared residuals and `K` = 20; that is, it tests any residual heteroscedasticity based on the first 20 lags of residual ACF of the squared residuals from the fitted GARCH model. Plotting is switched off by the `plot=F` option. The `gBox` function returns a list, an element of which is named `pvalue` and contains the *p*-values of the test for each `K`. Thus, the command prints out the *p*-value for the test with `K` = 20.

```
# Exhibit 12.30 on page 306.
acf(abs(residuals(m1)),na.action=na.omit)
```

As the initial residuals from a fitted GARCH model may be missing, it is essential to instruct the ACF to omit all missing values through the argument `na.action=na.omit` (the preferred action when encountering a missing value is to omit it). If this argument is omitted, the `acf` function uses all data and will return missing values if there are any missing data.

Overfitting the GARCH(1,2) model to the CREF returns can be carried out by the following command

```
m2=garch(x=r.cref,order=c(1,2))
summary(m2,diagnostics=F)
```

The summary is based on the `summary.garch` function in the `tseries` package. Note that the *p*-values of the Ljung-Box test from the summary are invalid; the generalized portmanteau tests should be used instead. Hence, the diagnostics are turned off.

```
AIC(m2)
```

This computes the AIC of the fitted GARCH model `m1`.

```
# Exhibit 12.31 on page 306.
gBox(m1,x=r.cref,method='absolute')
```

This carries out the generalized portmanteau test based on the absolute residuals.

```
shapiro.test(na.omit(residuals(m1)))
```

This computes the Shapiro-Wilk test for normality with the residuals from the fitted model m1. The function `na.omit` strips all missing values from the residuals. Thus, the test is carried out with the nonmissing residuals. Without preprocessing the residuals by the `na.omit` function, the test may return a missing value if some of the residuals are missing!

```
# Exhibit 12.32 on page 307.
plot((fitted(m1)[,1])^2,type='l',
   ylab='conditional variance',xlab='t')
```

The `fitted` function is a smart function that processes differently depending on the fitted model passed to it as the first argument. If the fitted model is some output from the `garch` function, the default output from the fitted function is a two-column matrix whose first column contains the one-step-ahead conditional standard deviations. Hence, their squares are the conditional variances. So `(fitted(m1)[,1])^2` computes the time series of estimated one-step-ahead conditional variances based on the model `m1`.

Chapter 13 R Commands

```
# Exhibit 13.3 on page 323.
```

The periodogram of a time series can be computed and plotted by the function `periodogram` into which the data are passed as its first argument.

```
sp=periodogram(y); abline(h=0);
   axis(1,at=c(0.04167,.14583))
```

The function `periodogram` has several useful arguments. Setting `log='yes'` tells R to plot on a log scale, whereas `log='no'` (the default) says to plot on a linear scale. Other arguments for the plot function may be passed into the function to make better graphs. The function `axis` draws an axis with the first argument specifying the side on which the axis is drawn. The sides are labeled from 1 to 4 starting from the bottom in a clockwise direction. The vector of locations of the tick marks can be specified by the `at` argument. The command above instructs R to draw an (additional) axis on the bottom of the figure with tick marks placed at 0.04167 and 0.14583.

```
# Exhibit 13.9 on page 333.
theta=.9 # Reset theta for other MA(1) plots
ARMAspec(model=list(ma=-theta))
```

The function `ARMAspec` calculates and plots the theoretical spectral density function of the ARMA model supplied to the function as the first argument. Recall that R uses the plus convention in the MA specification, so the minus sign is added to theta. The format of the model is the same as that for the `arima` function.

Chapter 14 R Commands

```
# Exhibit 14.2 on page 353.
```

The `spec` function can estimate the spectral density function by locally averaging the periodogram via some suitable kernel function. The function `spec` has several useful arguments. Setting `log='yes'` tells R to plot on a log scale whereas `log='no'` says to plot on a linear scale. Data may be detrended (fitting a linear time trend) by setting `detrend=T`, and tapering may be enforced by setting taper to some fraction between 0 and 0.5. The default options are: `taper=0` and `detrend=F`.

```
k=kernel('daniell',m=15)
```

Here, the object `k` contains the Daniell kernel function with halfwidth 15. Use Help in R to learn more about the `kernel` function.

```
sp=spec(y,kernel=k,log='no',sub='',
   xlab='Frequency',ylab='Smoothed Sample Spectral Density')
```

Specifying the kernel to be the Daniell kernel function instructs R to compute and plot the spectral density estimate, where the estimate at a certain frequency is obtained by averaging the current (raw) periodogram value, the neighboring 15 periodogram values on its left, and another 15 periodogram values on its right. More or less local averaging can be specified through the `m` argument in the kernel function.

```
lines(sp$freq,ARMAspec(model=list(ar=phi),freq=sp$freq,
   plot=F)$spec,lty='dotted')
```

This adds the theoretical spectral density function.

```
# Exhibits 14.11 and 14.12, page 364.
# Spectral analysis of simulated series
set.seed(271435)
n=100
phi1=1.5; phi2=-.75 # Reset parameter values to obtain
   Exhibits 14.13 & 14.14
y=arima.sim(model=list(ar=c(phi1,phi2)),n=n)
```

This simulates an AR(2) time series of length 100.

```
sp1=spec(y,spans=3,sub='',lty='dotted', xlab='Frequency',
   ylab='Log(Estimated Spectral Density)')
```

This estimates the special density function using the modified Daniell kernel (the default kernel when the kernel argument is missing and the `spans` argument is supplied). The `spans` argument supplies the width of the kernel function; that is, it is twice the `m` argument in the kernel function plus 1. Here, `spans=3` specifies local averaging of three consecutive periodogram values. Note that local averaging may be repeated by passing a vector as the value of `spans`. For example, setting `spans=c(3,5)` performs local averaging twice. The estimated function obtained by local averaging with `spans=3` is then averaged again locally with `spans=5`. Repeated averaging with a modified Daniell (rectangular) kernel is similar to averaging using a bell-shaped kernel due to the Central Limit effect.

```
sp2=spec(y,spans=9,plot=F)
```

This computes the spectrum estimate using a wider window encompassing nine periodogram values without plotting via the `plot=F` argument. The output of the `spec` function is saved into an object named `sp2`.

```
sp3=spec(y,spans=15,plot=F)
```

This uses an even wider window. How many periodogram values are included in each local averaging?

```
lines(sp2$freq,sp2$spec,lty='dashed')
```

This plots the smoother spectrum estimate (`spans=9`) as a dashed line.

```
lines(sp3$freq,sp3$spec,lty='dotdash')
```

This plots the smoothest spectrum estimate (`spans=15`) as a dotdash line.

```
f=seq(0.001,.5,by=.001)
```

This creates an arithmetic sequence starting from 0.001 and ending at 0.5, with increments 0.001, which is then saved into the object `f`.

```
lines(f,ARMAspec(model=list(ar=c(phi1,phi2)),freq=f,
   plot=F)$spec,lty='solid')
```

This plots the theoretical spectral density function for the specified ARMA model as connected line segments on top of the estimated spectral density plot.

```
# Exhibit 14.12 on page 365.
sp4=spec(y,method='ar',lty='dotted', xlab='Frequency',
   ylab='Log(Estimated AR Spectral Density)')
```

This estimates the spectral density function using the theoretical spectral density function of an AR model fitted to the data by minimizing the AIC.

```
f=seq(0.001,.5,by=.001)
lines(f,ARMAspec(model=list(ar=c(phi1,phi2)),
   freq=f,plot=F)$spec,lty='solid')
```

This plots the theoretical spectral density function.

```
sp4$method
```

This displays the order of the AR model selected.

Chapter 15 R Commands

```
# Exhibit 15.1 on page 386.
set.seed(2534567)
par(mfrow=c(3,2))
y=arima.sim(n=61,model=list(ar=c(1.6,-0.94),ma=-0.64))
```

This simulates an ARMA(2,1) series of sample size 61.

```
lagplot(y)
```

This plots the lagged regression plots, where the time series is plotted against its lags and a smooth curve is superimposed on each scatter diagram. The smooth curves are obtained by local linear fits to the data. By increasing the value specified in the `nn` argument (default `nn=0.7`), the local fitting scheme uses more local data, resulting in a smoother fit that is likely to be more biased but less variable due to more smoothing. On the contrary, decreasing the value in the `nn` argument leads to a rougher fit that is less biased but more variable due to less smoothing. The smooth curve in the scatter diagram of the time series response versus its lag *j* estimates the conditional mean response given its lag *j* as a function of the value of the lag *j* of the response. By default, `lagplot` plots the lagged regression plot for lags 1 to 6. More lags can be computed via the `lag.max` argument. For instance, `lag.max=12` computes the lagged regression plots for lags 1 through 12. Note that the `lagplot` function requires the installation of the `locfit` package of R.

```
# Exhibit 15.2 on page 387.
data(veilleux)
```

The dataset `veilleux` is a matrix consisting of two time series. Its first column is the series of *Didinium* abundance and the second column the series of *Paramecium* abundance, each counted every 12 hours. The basic time unit is days, so these are series of frequency 2, as they are sampled twice per day.

```
predator=veilleux[,1]
```

This defines the predator series as the abundance series of *Didinium*.

```
plot(log(predator),lty=2,type='b',xlab='Day',
   ylab='Log(predator)')
```

This plots the entire log-transformed predator series as a dashed line.

```
predator.eq=window(predator,start=c(7,1))
```

This subsets the "stationary" part of the predator series that appears to begin on the seventh day of the experiment. Subsequent analyses of the predator series reported in the text were done with this log-transformed stationary subseries.

```
lines(log(predator.eq))
```

This draws the stationary part as a solid line.

```
index1=zlag(log(predator.eq),3)<=4.661
```

The command `zlag(log(predator.eq),3)` returns the lag 3 of the (log-transformed) predator series. The expression `zlag(log(predator.eq),3)<=4.661` computes a Boolean vector whose elements are TRUE if and only if their corresponding element of the lag 3 of the predator series is less than or equal to 4.661. The Boolean vector is saved in an object named `index1`. Other comparison operators, including `>=`, `>`, `<`, and `==`, can be used to compare the vectors on the two sides of the comparison operator. In the example above, the left-hand side of `<=` is a vector, but its right-hand side is a scalar! The discrepancy is resolved by the recycling rule, that R replicates the shorter vector repeatedly to match its longer part. Note that the equality operator is denoted by the double equal sign `==`, as the single equal sign represents the assignment operator!

```
points(y=log(predator.eq)[index1],(time(predator.eq))
   [index1],pch=19)
```

This draws as solid circles `(pch=19)` those data points whose lag 3 of the predator abundance is less than or equal to 4.661. Run the command `?points` to learn other styles for plotting data points.

```
# Tests for nonlinearity, page 390.
Keenan.test(sqrt(spots))
```

This carries out Keenan's test for linearity. The working order of the AR process under the null hypothesis of linearity can be supplied via the order argument. For example, `order=2` sets the working AR order to 2. If the order argument is missing, the order is automatically determined by minimizing the AIC via the `ar` function. The `ar` function by default estimates the models by solving the Yule-Walker equations. But other estimation methods may be used by including the `method` argument when calling the `Keenan.test` function; for example, `method='mle'` specifies using maximum likelihood in the `ar` function.

```
Tsay.test(sqrt(spots)), page 390.
```

This implements Tsay's test for linearity; see Tsay (1986). The design of the `Tsay.test` function and its arguments are similar to those of the `Keenan.test` function.

```
# Exhibit 15.6 on page 400.
y=qar.sim(n=100,const=0.0,phi0=3.97,
   phi1=-3.97,sigma=0,init=.377)
```

The function `qar.sim` simulates a time series realization from a first-order quadratic AR model where `phi0` is the coefficient of the lag 1 and `phi1` is that of the square of lag 1. The default intercept is zero, otherwise it can be set by the `const` argument. The innovation standard deviation is passed into the function via the `sigma` argument. Here, `sigma=1` sets the standard deviation to be 1. The argument `n=15` sets the sample size to 15. Finally, the argument `init=.377` sets the initial value to be 0.377. The default initial value is 0.

```
plot(x=1:100,y=y,type='l',ylab=expression(Y[t]),xlab='t')
```

The output of the `qar.sim` function is a vector. To draw the time sequence plot, both the *x*-variable and the *y*-variable have to be specified.

```
# Exhibit 15.8 on page 411.
set.seed(1234579)
y=tar.sim(n=100,Phi1=c(0,0.5),Phi2=c(0,-1.8),p=1,d=1,
   sigma1=1,thd=-1,sigma2=2)$y
```

The function `tar.sim` simulates time series realizations from a two-regime TAR model. The order of the model is specified by the `p` argument, so `p=1` specifies a first-order model. The delay is passed into the function by the `d` argument, so `d = 1` specifies the delay to be 1. The AR coefficient vector for the lower (upper) regime, with the intercept being the first component, is supplied via the `Phi1` (`Phi2`) argument. The `thd=-1` argument imposes the threshold parameter of −1. The innovation standard deviations for the lower and upper regimens are specified via the `sigma1` and `sigma2` arguments, respectively. The simulated TAR model in the example is conditionally heteroscedastic, as the innovation standard deviation for the upper regime is twice that for the lower regime. The sample size is set to 100 by the `n=100` argument.

The likelihood ratio test for threshold nonlinearity, assuming normally distributed innovations, can be carried out by the `tlrt` function, with which the data enter into the function as the first argument. Other required information includes the `order` and `delay` arguments. Also, the threshold parameter must be searched over a finite interval from the *a* times 100 percentile to the *b* times 100 percentile of the data. Often, data have to be transformed before testing for nonlinearity, which can be specified by supplying the transformed data or supplying the raw data with the transform argument set to one of the available options: '`no`' (means no transformation, the default), '`log`', '`log10`', or '`sqrt`'. For example, the following command does the likelihood ratio test of the null hypothesis that the square root transformation of relative sunspot data is an AR(5) process versus the alternative that it follows a threshold model with delay 1, order 5, and with the threshold parameter searched from the first to the third quartile of the (transformed) data.

```
tlrt(sqrt(spots),p=5,d=1,a=0.25,b=0.75)
```

The `tlrt` function outputs a list containing the test statistic and its *p*-value. In practice, the true delay of the threshold model is unknown, although it is likely to be between 1 and the order of the model. (The delay may be specified to some value greater than the order if this is deemed appropriate.) The command above can be replicated a number of times for each possible delay value. A more elegant way is to use a `for` loop as follows.

```
# Tests for threshold nonlinearity, page 400.
pvaluem=NULL
```

This defines an empty object named `pvaluem`.

```
for (d in 1:5)
   {res=tlrt(sqrt(spots),p=5,d=d,a=0.25,b=0.75); pvaluem=
   cbind(pvaluem,c(d,res$test.statistic,res$p.value))}
```

The statements within the curly brackets are repeated for each value the variable `d` takes sequentially from the vector `1:5`, which contains the first five positive integers. Thus, `d` is first set to 1, and the likelihood ratio test for threshold nonlinearity is carried out, with its output stored in an object named `res`. The command `c(d,res$test.statistic,res$p.value)` creates a vector containing the value 1, the likelihood ratio test statistic, and its *p*-value. The vector so created is then augmented to the right-hand side of `pvaluem` to form a matrix. So, after the first loop, `pvaluem` is a matrix consisting of the test results for `d=1`. Then the loop sets `d` to the second value, namely 2; carries out the threshold likelihood ratio test for `d=2`; augments the test results for `d=2` to the right-hand side of `pvaluem`; and so forth until the loop exhausts all possible values for `d` and `n` and then R exits from the loop.

```
rownames(pvaluem)=c('d','test statistic','p-value')
```

This labels the rows of the `pvaluem` matrix, with the first row labeled as "`d`", the second "test statistic", and the third row "*p*-value".

```
round(pvaluem,3)
```

This prints out the matrix (table) of test results, with the numbers rounded to three decimal places. Note that the computational efficiency of the R code above can be improved by declaring `pvaluem` as a matrix with appropriate dimension (for example, `pvaluem= matrix('NA',nrow=3,ncol=5)`) in which the test results are saved.

```
# Exhibit 15.12 on page 405.
predator.tar.1=tar(y=log(predator.eq),p1=4,p2=4,d=3,a=.1,
   b=.9,print=T)
```

This fits a threshold model with the (log-transformed) `predator.eq` series with maximum AR order to be 4 for both lower and upper regimes, `d=3`, and the threshold parameter searched from the tenth to the ninetieth percentiles. The fitted model is printed out if the print argument is set to `T`. By default, the function uses the MAIC (minimum AIC) method for estimation, with the AR orders estimated as well. Another method of estimation is conditional least squares, which can be specified by the `method='CLS'`, as illustrated in the next command.

In the command below, we repeat the estimation but using the CLS method. Note that the CLS method does not estimate the AR orders of the two regimes. Instead, the AR orders are set as the maximum orders specified through the `p1` and `p2` arguments! That is why the values of `p1` and `p2` are set differently from the previous command and in fact set as the orders estimated from the model using the MAIC method.

```
tar(y=log(predator.eq),p1=1,p2=4,d=3,a=.1,b=.9,print=T,
   method='CLS')
```

```
# Exhibit 15.13 on page 408.
tar.skeleton(predator.tar.1)
```

This computes the skeleton of a TAR model supplied as the first argument, with a default sample size of 500 values, a burn-in of 500 values, and plots the time sequence plot of the last 50 values of the skeleton. The TAR model is usually the output of that of the `object` argument of the tar function. Alternatively, the model parameters can be specified in a format similar to the `tar.sim` function. The function also prints a summary statement on the long-run behavior of the skeleton.

```
# Exhibit 15.14 on page 408.
set.seed(356813)
plot(y=tar.sim(n=57,object=predator.tar.1)$y,x=1:57,
   ylab=expression(Y[t]),xlab=expression(t),type='o')
```

This plots a simulated time series from the fitted TAR(2;1,4) model to the predator series. The fitted model is supplied via the `object` argument.

```
# Exhibit 15.20 on page 414.
tsdiag(predator.tar.1,gof.lag=20)
```

This carries out several model diagnostics on the fitted TAR(2;1,4) model to the predator series. The function plots a time sequence plot of the standardized residuals, the residual ACF, and the *p*-value plots of the generalized portmanteau tests. The argument `gof.lag=20` specifies that the last two plots use a maximum lag of 20.

```
# Exhibit 15.21 on page 415.
qqnorm(predator.tar.1$std.res)
```

This plots the quantile-quantile normal score plot for the standardized residuals from the TAR(2;1,4) model fitted to the predator series.

```
qqline(predator.tar.1$std.res)
```

adds the reference line on the Q-Q plot.

```
# Exhibit 15.22 on page 417.
set.seed(2357125)
pred.predator=predict(predator.tar.1,n.ahead=60,
   n.sim=1000)
```

This simulates a time series from the conditional distribution of the future values given the data and a threshold model (usually the output of the tar function, here being `predator.tar.1`), with a forecast horizon of a maximum sixty-step-ahead predictions. The point predictors and their 95% prediction limits are computed by simulation. The simulation size is specified as `n.sim=1000`. The output of the `predict` function is a list that contains the prediction means as a vector in the component (element) named `fit` and the lower and upper prediction limits as a matrix in the `pred.interval` component. The function `predict` is a smart function and recognizes that the first argument is a TAR model, on the basis of which it computes the prediction. To learn more about the `predict` function for TAR models, run `?predict.TAR`. The extension TAR signifies the particular predict function for processing prediction based on a TAR model.

```
yy=ts(c(log(predator.eq),pred.predator$fit),frequency=2,
   start=start(predator.eq))
```

This augments the point prediction values to the data.

```
plot(yy,type='n',
   ylim=range(c(yy,pred.predator$pred.interval)),
   ylab='Log Prey', xlab=expression(t))
```

This sets up a plot of the data and the predicted future values without actual plotting (`type='n'`). We anticipate superimposing the prediction intervals, so the range of the *y*-axis is specified through the `ylim` argument to the vector containing the minimum and maximum of the combined vector of the observed + predicted values (`yy`) and the prediction limits (`pred.predator$pred.interval`), computed via the `range` function.

```
lines(log(predator.eq))
```

This draws the data as a solid line.

```
lines(window(yy, start=end(predator.eq)+c(0,1)),lty=2)
```

This adds the curve of the predicted values as a dashed line.

```
lines(ts(pred.predator$pred.interval[2,],
   start=end(predator.eq)+c(0,1),freq=2),lty=2)
```

This adds the upper prediction limits.

```
lines(ts(pred.predator$pred.interval[1,],
   start=end(predator.eq)+c(0,1),freq=2),lty=2)
```

This adds the lower prediction limits.

```
# Exhibit 15.24 on page 419.
qqnorm(pred.predator$pred.matrix[,3])
```

The output of the `predict` function is a list that contains another component, named `pred.matrix`, which is a matrix containing all simulated future values, with the first column consisting of the simulated one-step-ahead values, the second column those of the two-steps-ahead values, and so forth.

```
qqnorm(pred.predator$pred.matrix[,3])
```

This extracts all 1000 simulated three-steps-ahead values, which are then passed into the `qqnorm` function to make the Q-Q normal score plot for these data.

```
qqline(pred.predator$pred.matrix[,6])
```

This adds the reference straight line for checking the normality of the three-steps-ahead conditional distribution.

Finally, here is a listing and brief description of all the new or enhanced functions that are contained in the TSA package.

New or Enhanced Functions in the TSA Library

Function	Description
`acf`	Computes and plots the sample autocorrelation function starting with lag 1.
`arima`	This command has been amended to compute the AIC according to our definition.
`arima.boot`	Bootstraps time series according to a fitted ARMA(p,d,q) model.
`arimax`	Extends the `arima` function, allowing the incorporation of transfer functions and innovative and additive outliers.
`ARMAspec`	Computes and plots the theoretical spectrum of an ARMA model.
`armasubsets`	Finds “best subset” ARMA models.
`BoxCox.ar`	Finds a power transformation so that the transformed time series is approximately an AR process with normal error terms.
`detectAO`	Detects additive outliers in time series.
`detectIO`	Detects innovative outliers in time series.
`eacf`	Computes and displays the extended autocorrelation function of a time series.
`garch.sim`	Simulates a GARCH process.
`gBox`	Performs a goodness-of-fit test for fitted GARCH models.
`harmonic`	Creates a matrix of the first m pairs of harmonic functions for fitting a harmonic trend (cosine-sine trend, Fourier regression) model with a time series response.

New or Enhanced Functions in the TSA Library (Continued)

Function	Description
`Keenan.test`	Carries out Keenan's test for nonlinearity against the null hypothesis that the time series follows some AR process.
`kurtosis`	Calculates the (excess) coefficient of kurtosis.
`lagplot`	Computes and plots nonparametric regression functions of a time series against its various lags.
`periodogram`	Computes the periodogram of a time series.
`LB.test`	Computes the Ljung-Box or Box-Pierce tests checking whether or not the residuals from an ARIMA model appear to be white noise.
`McLeod.Li.test`	Perform the McLeod-Li test for conditional heteroscedascity (ARCH).
`plot.Arima`	Plots a time series and its predictions (forecasts) with 95% prediction bounds based on a fitted ARIMA model.
`predict.TAR`	Calculates predictions based on a fitted TAR model. The errors are assumed to be normally distributed and the predictive distributions are approximated by simulation.
`prewhiten`	Bivariate time series are prewhitened according to an AR model fitted to the *x*-component of the bivariate series. Alternatively, if an ARIMA model is provided, it is used to prewhiten both series. The CCF of the prewhitened bivariate series is then computed and plotted.
`qar.sim`	Simulates a first-order quadratic AR model with normally distributed white noise error terms.
`rstandard.Arima`	Computes internally standardized residuals from a fitted ARIMA model.
`runs`	Tests the independence of a sequence of values by checking whether there are too many or too few runs above (or below) the median.
`season`	Extracts season information from a time series and creates a vector of the season information. For example, for monthly data, the function outputs a vector containing the months of the data.
`skewness`	Calculates the skewness coefficient of a dataset.
`spec`	Allows the user to invoke either the `spec.pgram` function or the `spec.ar` function in the `stats` package. The seasonal attribute of the data, if it exists, is surpressed for our preferred way of presenting the output. Alters defaults to `demean=T`, `detrend=F`, `taper=0`, and permits plotting of confidence interval bands.
`summary.armasub-sets`	Summary method for class armasubsets, that is useful for ARMA subset selection.
`tar`	Estimates a two-regime TAR model.

New or Enhanced Functions in the TSA Library (Continued)

Function	Description
`tar.sim`	Simulates a two-regime TAR model.
`tar.skeleton`	Obtains the skeleton of a TAR model by suppressing the noise term in the TAR model.
`tlrt`	Carries out the likelihood ratio test for threshold nonlinearity, with the null hypothesis being a normal AR process and the alternative hypothesis being a TAR model with homogeneous, normally distributed errors.
`Tsay.test`	Carries out Tsay's test for quadratic nonlinearity in a time series.
`tsdiag.Arima`	Modifies the `tsdiag` function of the `stats` package suppressing initial residuals and displaying Bonferroni bounds. It also checks the condition for the validity of the chi-square asymptotics for the portmanteau tests.
`tsdiag.TAR`	Displays the time series plot and the sample ACF of the standardized residuals. Also, portmanteau tests for detecting autocorrelations in the standardized residuals are computed and displayed.
`zlag`	Computes the lag of a vector, with missing elements replaced by NA.