# Appendix R

# R Supplement

## R.1 First Things First

If you do not already have R, point your browser to the Comprehensive R Archive Network (CRAN), `http://cran.r-project.org/` and download and install it. The installation includes help files and some user manuals. You can find helpful tutorials by following CRAN's link to *Contributed Documentation*. If you are a novice, then RStudio (`https://www.rstudio.com/`) will make using R easier.

## R.2 astsa

There is an R package for the text called `astsa` (*Applied Statistical Time Series Analysis*), which was the name of the software distributed with the first and second editions of Shumway and Stoffer (2000), and the original version, Shumway [183]. The package can be obtained from CRAN and its mirrors in the usual way. To download and install `astsa`, start R and type

```
install.packages("astsa")
```

You will be asked to choose the closest CRAN mirror to you. As with all packages, you have to load `astsa` before you use it by issuing the command

```
library(astsa)
```

All the data are loaded when the package is loaded. If you create a `.First` function as follows,

```
.First <- function(){library(astsa)}
```

and save the workspace when you quit, `astsa` will be loaded at every start until you change `.First`.

R is not consistent with help files across different operating systems. The best help system is the html help, which can be started issuing the command `help.start()`

R.H. Shumway, D.S. Stoffer, *Time Series Analysis and Its Applications*,
Springer Texts in Statistics, DOI 10.1007/978-3-319-52452-8

and then following the *Packages* link to `astsa`. A useful command to see all the data files available to you, including those loaded with `astsa`, is

```
data()
```

## R.3 Getting Started

The convention throughout the text is that R code is in `blue`, output is `purple` and comments are *`# green`*. Get comfortable, then start her up and try some simple tasks.

```
2+2            # addition
 [1] 5
5*5 + 2        # multiplication and addition
 [1] 27
5/5 - 3        # division and subtraction
 [1] -2
log(exp(pi))   # log, exponential, pi
 [1] 3.141593
sin(pi/2)      # sinusoids
 [1] 1
exp(1)^(-2)    # power
 [1] 0.1353353
sqrt(8)        # square root
 [1] 2.828427
1:5            # sequences
 [1] 1 2 3 4 5
seq(1, 10, by=2)    # sequences
 [1] 1 3 5 7 9
rep(2, 3)       # repeat 2 three times
 [1] 2 2 2
```

Next, we'll use *assignment* to make some *objects*:

```
x <- 1 + 2  # put 1 + 2 in object x
x = 1 + 2   # same as above with fewer keystrokes
1 + 2 -> x  # same
x           # view object x
 [1]  3
(y = 9 * 3)    # put 9 times 3 in y and view the result
 [1] 27
(z = rnorm(5))  # put 5 standard normals into z and print z
 [1]  0.96607946  1.98135811 -0.06064527  0.31028473  0.02046853
```

In general, `<-` and `=` are not the same; `<-` can be used anywhere, whereas the use of `=` is restricted. But when they are the same, we prefer to code using the least number of keystrokes.

It is worth pointing out R's *recycling rule* for doing arithmetic. In the code below, `c()` [concatenation] is used to create a vector. Note the use of the semicolon for multiple commands on one line.

```
x = c(1, 2, 3, 4); y = 2*x; z = c(10, 20); w = c(8, 3, 2)
x * y     # 1*2, 2*4, 3*6, 4*8
 [1]  2  8 18 32
x + z     # 1+10, 2+20, 3+10, 4+20
 [1] 11 22 13 24
x + w     # what happened here?
```

```
 [1]  9  5  5 12
 Warning message:
  In y + w : longer object length is not a multiple of
   shorter object length
```

To work your objects, use the following commands:

```
ls()                  # list all objects
 "dummy" "mydata" "x" "y" "z"
ls(pattern = "my")  # list every object that contains "my"
 "dummy" "mydata"
rm(dummy)             # remove object "dummy"
rm(list=ls())         # remove almost everything (use with caution)
help.start()          # html help and documentation
data()                # list of available data sets
help(exp)             # specific help  (?exp is the same)
getwd()               # get working directory
setwd()               # change working directory
q()                   # end the session (keep reading)
```

When you quit, R will prompt you to save an image of your current workspace. Answering *yes* will save the work you have done so far, and load it when you next start R. We have never regretted selecting *yes*, but we have regretted answering *no*.

To create your own data set inside R, you can make a data vector as follows:

```
mydata = c(1,2,3,2,1)
```

Now you have an object called `mydata` that contains five elements. R calls these objects *vectors* even though they have no dimensions (no rows, no columns); they do have order and length:

```
mydata         # display the data
 [1] 1 2 3 2 1
mydata[3]      # the third element
 [1] 3
mydata[3:5]    # elements three through five
 [1] 3 2 1
mydata[-(1:2)] # everything except the first two elements
 [1] 3 2 1
length(mydata) # number of elements
 [1] 5
dim(mydata)    # no dimensions
 NULL
mydata = as.matrix(mydata)  # make it a matrix
dim(mydata)    # now it has dimensions
 [1] 5 1
```

If you have an external data set, you can use `scan` or `read.table` (or some variant) to input the data. For example, suppose you have an ASCII (text) data file called `dummy.txt` in your working directory, and the file looks like this:

```
1 2 3 2 1
9 0 2 1 0
```

```
(dummy = scan("dummy.txt") )        # scan and view it
 Read 10 items
  [1] 1 2 3 2 1 9 0 2 1 0
(dummy = read.table("dummy.txt") )  # read and view it
 V1 V2 V3 V4 V5
```

```
  1  2  3  2  1
  9  0  2  1  0
```

There is a difference between `scan` and `read.table`. The former produced a data vector of 10 items while the latter produced a *data frame* with names `V1` to `V5` and two observations per variate. In this case, if you want to list (or use) the second variate, `V2`, you would use

```
dummy$V2
 [1] 2 0
```

and so on. You might want to look at the help files `?scan` and `?read.table` now. Data frames (`?data.frame`) are "used as the fundamental data structure by most of R's modeling software." Notice that R gave the columns of `dummy` generic names, `V1, ..., V5`. You can provide your own names and then use the names to access the data without the use of `$` as above.

```
colnames(dummy) = c("Dog", "Cat", "Rat", "Pig", "Man")
attach(dummy)
Cat
 [1] 2 0
Rat*(Pig - Man)  # animal arithmetic
 [1] 3 2
head(dummy)      # view the first few lines of a data file
detach(dummy)    # clean up (if desired)
```

R is case sensitive, thus `cat` and `Cat` are different. Also, `cat` is a reserved name (`?cat`) in R, so using `"cat"` instead of `"Cat"` may cause problems later. You may also include a *header* in the data file to avoid `colnames()`. For example, if you have a *comma separated values* file `dummy.csv` that looks like this,

```
Dog,Cat,Rat,Pig,Man
1,2,3,2,1
9,0,2,1,0
```

then use the following command to read the data.

```
(dummy = read.csv("dummy.csv"))
    Dog Cat Rat Pig Man
  1   1   2   3   2   1
  2   9   0   2   1   0
```

The default for `.csv` files is `header=TRUE`; type `?read.table` for further information on similar types of files.

Some commands that are used frequently to manipulate data are `c()` for *concatenation*, `cbind()` for *column binding*, and `rbind()` for *row binding*.

```
x = 1:3;  y = 4:6
(u = c(x, y))          # an R vector
  [1] 1 2 3 4 5 6
(u1 = cbind(x, y))     # a 3 by 2 matrix
       x y
  [1,] 1 4
  [2,] 2 5
  [3,] 3 6
(u2 = rbind(x ,y))     # a 2 by 3 matrix
    [,1] [,2] [,3]
  x    1    2    3
  y    4    5    6
```

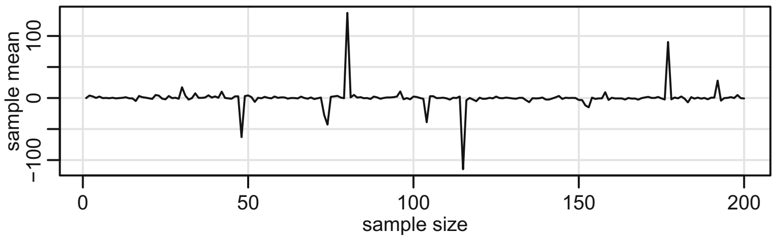

**Fig. R.1.** Crazy example

For example, `u1[,2]` is the second column of the matrix `u1`, whereas `u2[1,]` is the first row of `u2`.

Summary statistics are fairly easy to obtain. We will simulate 25 normals with $\mu = 10$ and $\sigma = 4$ and then perform some basic analyses. The first line of the code is `set.seed`, which fixes the seed for the generation of pseudorandom numbers. Using the same seed yields the same results; to expect anything else would be insanity.

```
set.seed(90210)         # so you can reproduce these results
x = rnorm(25, 10, 4)    # generate the data
c( mean(x), median(x), var(x), sd(x) )  # guess
 [1]  9.473883  9.448511  13.926701  3.731850
c( min(x), max(x) )  # smallest and largest values
 [1]  2.678173 17.326089
which.max(x)    # index of the max (x[25] in this case)
 [1] 25
summary(x)     # a five number summary with six numbers
     Min. 1st Qu. Median   Mean  3rd Qu.    Max.
   2.678   7.824   9.449  9.474   11.180  17.330
boxplot(x);  hist(x);  stem(x)    # visual summaries (not shown)
```

It can’t hurt to learn a little about programming in R because you will see some of it along the way. Consider a simple program that we will call `crazy` to produce a graph of a sequence of sample means of increasing sample sizes from a Cauchy distribution with location parameter zero.

```
crazy <- function(num) {
  x <- c()
  for (n in 1:num) { x[n] <- mean(rcauchy(n)) }
  plot(x, type="l", xlab="sample size", ylab="sample mean")
  }
```

The first line creates the function `crazy` and gives it one argument, `num`, that is the sample size that will end the sequence. Line 2 makes an empty vector, `x`, that will be used to store the sample means. Line 3 generates `n` random Cauchy variates [`rcauchy(n)`], finds the mean of those values, and puts the result into `x[n]`, the $n$-th value of `x`. The process is repeated in a “do loop” `num` times so that `x[1]` is the sample mean from a sample of size one, `x[2]` is the sample mean from a sample of size two, and so on, until finally, `x[num]` is the sample mean from a sample of size `num`. After the do loop is complete, the fourth line generates a graphic (see Fig. R.1). The fifth line closes the function. To use `crazy` ending with sample of size of 200, type

```
crazy(200)
```

and you will get a graphic that looks like Fig. R.1.

Finally, a word of caution: `TRUE` and `FALSE` are reserved words, whereas `T` and `F` are initially set to these. Get in the habit of using the words rather than the letters `T` or `F` because you may get into trouble if you do something like

```
F = qf(p=.01, df1=3, df2=9)
```

so that `F` is no longer `FALSE`, but a quantile of the specified $F$-distribution.

## R.4 Time Series Primer

In this section, we give a brief introduction on using R for time series. *We assume that* `astsa` *has been loaded.* To create a time series object, use the command `ts`. Related commands are `as.ts` to coerce an object to a time series and `is.ts` to test whether an object is a time series. First, make a small data set:

```
(mydata = c(1,2,3,2,1) ) # make it and view it
  [1] 1 2 3 2 1
```

Now make it a time series:

```
(mydata = as.ts(mydata) )
  Time Series:
  Start = 1
  End = 5
  Frequency = 1
  [1] 1 2 3 2 1
```

Make it an annual time series that starts in 1950:

```
(mydata = ts(mydata, start=1950) )
  Time Series:
  Start = 1950
  End = 1954
  Frequency = 1
  [1] 1 2 3 2 1
```

Now make it a quarterly time series that starts in 1950-III:

```
(mydata = ts(mydata, start=c(1950,3), frequency=4) )
       Qtr1 Qtr2 Qtr3 Qtr4
  1950                1    2
  1951    3    2    1
time(mydata)  # view the sampled times
          Qtr1    Qtr2    Qtr3    Qtr4
  1950                 1950.50 1950.75
  1951 1951.00 1951.25 1951.50
```

To use part of a time series object, use `window()`:

```
(x = window(mydata, start=c(1951,1), end=c(1951,3) ))
       Qtr1 Qtr2 Qtr3
  1951    3    2    1
```

Next, we'll look at lagging and differencing. First make a simple series, $x_t$:

```
x = ts(1:5)
```

Now, column bind (`cbind`) lagged values of $x_t$ and you will notice that `lag(x)` is *forward* lag, whereas `lag(x, -1)` is *backward* lag.

```
cbind(x, lag(x), lag(x,-1))
       x lag(x) lag(x, -1)
  0   NA      1         NA
  1    1      2         NA
  2    2      3          1
  3    3      4          2 <- in this row, for example, x is 3,
  4    4      5          3    lag(x) is ahead at 4, and
  5    5     NA          4    lag(x,-1) is behind at 2
  6   NA     NA          5
```

Compare `cbind` and `ts.intersect`:

```
ts.intersect(x, lag(x,1), lag(x,-1))
  Time Series:  Start = 2  End = 4  Frequency = 1
      x lag(x, 1) lag(x, -1)
  2   2         3          1
  3   3         4          2
  4   4         5          3
```

To difference a series, $\nabla x_t = x_t - x_{t-1}$, use

```
diff(x)
```

but note that

```
diff(x, 2)
```

is *not* second order differencing, it is $x_t - x_{t-2}$. For second order differencing, that is, $\nabla^2 x_t$, do one of these:

```
diff(diff(x))
diff(x, diff=2)   # same thing
```

and so on for higher order differencing.

We will also make use of regression via `lm()`. First, suppose we want to fit a simple linear regression, $y = \alpha + \beta x + \epsilon$. In R, the formula is written as `y~x`:

```
set.seed(1999)
x = rnorm(10)
y = x + rnorm(10)
summary(fit <- lm(y~x) )
  Coefficients:
              Estimate Std. Error t value Pr(>|t|)
  (Intercept)   0.2576     0.1892   1.362   0.2104
  x             0.4577     0.2016   2.270   0.0529
  --
  Residual standard error: 0.58 on 8 degrees of freedom
  Multiple R-squared: 0.3918,     Adjusted R-squared: 0.3157
  F-statistic: 5.153 on 1 and 8 DF,  p-value: 0.05289
plot(x, y)   # draw a scatterplot of the data (not shown)
abline(fit)  # add the fitted line to the plot (not shown)
```

All sorts of information can be extracted from the `lm` object, which we called `fit`. For example,

```
resid(fit)     # will display the residuals (not shown)
fitted(fit)    # will display the fitted values (not shown)
lm(y ~ 0 + x)  # will exclude the intercept  (not shown)
```

You have to be careful if you use `lm()` for lagged values of a time series. If you use `lm()`, then what you have to do is align the series using `ts.intersect`. Please read the warning *Using time series* in the `lm()` help file [`help(lm)`]. Here is an example regressing `astsa` data, weekly cardiovascular mortality (`cmort`) on particulate pollution (`part`) at the present value and lagged four weeks (`part4`). First, we create `ded`, which consists of the intersection of the three series:

```
ded = ts.intersect(cmort, part, part4=lag(part,-4))
```

Now the series are all aligned and the regression will work.

```
summary(fit <- lm(cmort~part+part4, data=ded, na.action=NULL) )
 Coefficients:
             Estimate Std. Error t value Pr(>|t|)
 (Intercept) 69.01020    1.37498  50.190  < 2e-16
 part         0.15140    0.02898   5.225 2.56e-07
 part4        0.26297    0.02899   9.071  < 2e-16
 ---
 Residual standard error: 8.323 on 501 degrees of freedom
 Multiple R-squared:  0.3091,    Adjusted R-squared:  0.3063
 F-statistic: 112.1 on 2 and 501 DF,  p-value: < 2.2e-16
```

There was no need to rename `lag(part,-4)` to `part4`, it's just an example of what you can do.

An alternative to the above is the package `dynlm`, which has to be installed. After the package is installed, the previous example may be run as follows:

```
library(dynlm)                          # load the package
fit = dynlm(cmort~part + L(part,4))     # no new data file needed
summary(fit)
```

The output is identical to the `lm` output. To fit another model, for example, add the temperature series `tempr`, the advantage of `dynlm` is that a new data file does not have to be created. We could just run

```
summary(dynlm(cmort~ tempr + part + L(part,4)) )
```

In Problem 2.1, you are asked to fit a regression model

$$x_t = \beta t + \alpha_1 Q_1(t) + \alpha_2 Q_2(t) + \alpha_3 Q_3(t) + \alpha_4 Q_4(t) + w_t$$

where $x_t$ is logged Johnson & Johnson quarterly earnings ($n = 84$), and $Q_i(t)$ is the indicator of quarter $i = 1, 2, 3, 4$. The indicators can be made using `factor`.

```
trend = time(jj) - 1970         # helps to 'center' time
Q     = factor(cycle(jj) )      # make (Q)uarter factors
reg   = lm(log(jj)~0 + trend + Q, na.action=NULL)  # no intercept
model.matrix(reg)               # view the model design matrix
           trend Q1 Q2 Q3 Q4
    1   -10.00  1  0  0  0
    2    -9.75  0  1  0  0
    3    -9.50  0  0  1  0
    4    -9.25  0  0  0  1
    .      .    .  .  .  .
    .      .    .  .  .  .
summary(reg)                     # view the results (not shown)
```

The workhorse for ARIMA simulations is `arima.sim`. Here are some examples; no output is shown here so you're on your own.

```
 x = arima.sim(list(order=c(1,0,0), ar=.9), n=100) + 50    # AR(1) w/mean 50
 x = arima.sim(list(order=c(2,0,0), ar=c(1,-.9)), n=100)   # AR(2)
 x = arima.sim(list(order=c(1,1,1), ar=.9 ,ma=-.5), n=200) # ARIMA(1,1,1)
```

An easy way to fit ARIMA models is to use `sarima` from `astsa`. The script is used in Chap. 3 and is introduced in Sect. 3.7.

### R.4.1 Graphics

We introduced some graphics without saying much about it. Many people use the graphics package `ggplot2`, but for quick and easy graphing of time series, the R base graphics does fine and is what we discuss here. As seen in Chap. 1, a time series may be plotted in a few lines, such as

```
plot(speech)
```

in Example 1.3, or the multifigure plot

```
plot.ts(cbind(soi, rec) )
```

which we made little fancier in Example 1.5:

```
par(mfrow = c(2,1))
plot(soi, ylab='', xlab='', main='Southern Oscillation Index')
plot(rec, ylab='', xlab='', main='Recruitment')
```

But, if you compare the results of the above to what is displayed in the text, there is a slight difference because we improved the aesthetics by adding a grid and cutting down on the margins. This is how we actually produced Fig. 1.3:

```
1 dev.new(width=7, height=4)            # default is 7 x 7 inches
2 par(mar=c(3,3,1,1), mgp=c(1.6,.6,0) )  # change the margins (?par)
3 plot(speech, type='n')
4 grid(lty=1, col=gray(.9)); lines(speech)
```

In line 1, the dimensions are in inches. Line 2 adjusts the margins; see `help(par)` for a complete list of settings. In line 3, the `type='n'` means to set up the graph, but don't actually plot anything yet. Line 4 adds a grid and then plots the lines. The reason for using `type='n'` is to avoid having the grid lines on top of the data plot. You can print the graphic directly to a pdf, for example, by replacing line 1 with something like

```
pdf(file="speech.pdf", width=7, height=4)
```

but you have to turn the device off to complete the file save:

```
dev.off()
```

Here is the code we used to plot two series individually in Fig. 1.5:

```
dev.new(width=7, height=6)
par(mfrow = c(2,1), mar=c(2,2,1,0)+.5, mgp=c(1.6,.6,0) )
plot(soi, ylab='', xlab='', main='Southern Oscillation Index', type='n')
grid(lty=1, col=gray(.9));  lines(soi)
plot(rec, ylab='', main='Recruitment', type='n')
grid(lty=1, col=gray(.9));  lines(rec)
```

For plotting many time series, `plot.ts` and `ts.plot` are available. If the series are all on the same scale, it might be useful to do the following:

```
ts.plot(cmort, tempr, part, col=1:3)
legend('topright', legend=c('M','T','P'), lty=1, col=1:3)
```

This produces a plot of all three series on the same axes with different colors, and then adds a legend. We are not restricted to using basic colors; an internet search on 'R colors' is helpful. The following code gives separate plots of each different series (with a limit of 10):

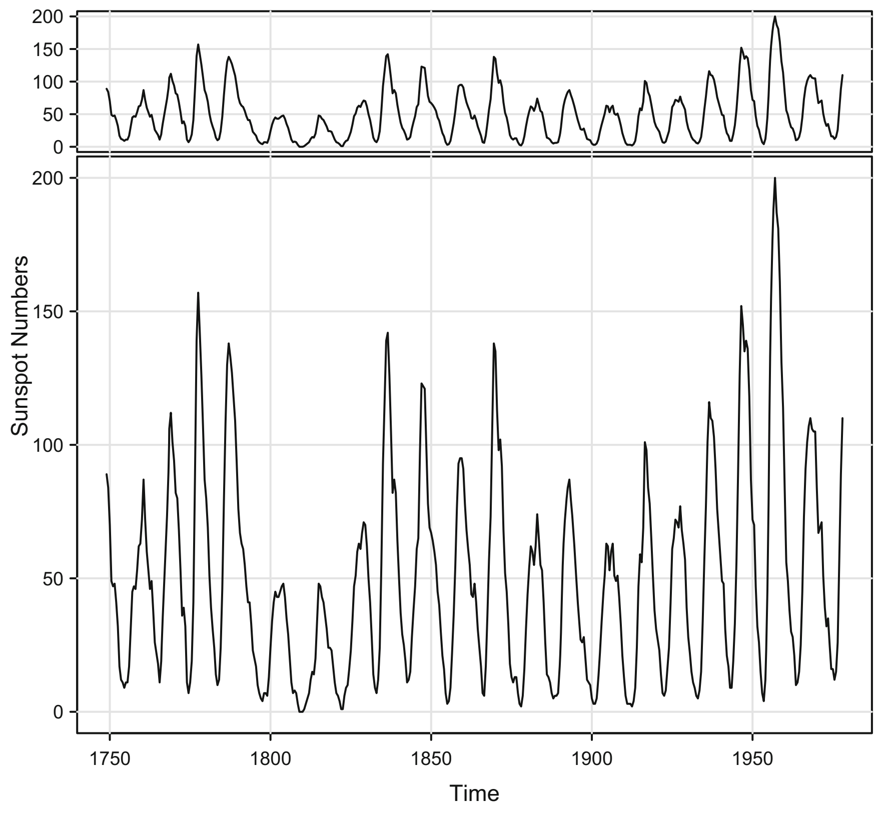

**Fig. R.2.** The sunspot numbers plotted in different-sized boxes, demonstrating that the dimensions of the graphic matters when displaying time series data

```
plot.ts(cbind(cmort, tempr, part) )
plot.ts(eqexp)                        # you will get a warning
plot.ts(eqexp[,9:16], main='Explosions') # but this works
```

Finally, we mention that size matters when plotting time series. Figure R.2 shows the sunspot numbers discussed in Problem 4.9 plotted with varying dimension size as follows.

```
layout(matrix(c(1:2, 1:2), ncol=2), height=c(.2,.8))
par(mar=c(.2,3.5,0,.5), oma=c(3.5,0,.5,0), mgp=c(2,.6,0), tcl=-.3, las=1)
plot(sunspotz, type='n', xaxt='no', ylab='')
  grid(lty=1, col=gray(.9))
  lines(sunspotz)
plot(sunspotz, type='n', ylab='')
  grid(lty=1, col=gray(.9))
  lines(sunspotz)
title(xlab="Time", outer=TRUE, cex.lab=1.2)
mtext(side=2, "Sunspot Numbers", line=2, las=0, adj=.75)
```

The result is shown in Fig. R.2. The top plot is wide and narrow, revealing the fact that the series rises quickly $\uparrow$ and falls slowly $\searrow$. The bottom plot, which is more square, obscures this fact. You will notice that in the main part of the text, we never plotted a series in a square box. The ideal shape for plotting time series, in most instances, is when the time axis is much wider than the value axis.