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 new to R/S-PLUS, then *R for Beginners* by Emmanuel Paradis is a great introduction. There is also a lot of advice out there in cyberspace, but some of it will be outdated because R goes through many revisions.

Next, point your browser to `http://www.stat.pitt.edu/stoffer/tsa3/`, the website for the text, or one of its mirrors, download `tsa3.rda` and put it in a convenient place (e.g., the working directory of R).[1] This file contains the data sets and scripts that are used in the text. Then, start R and issue the command

```
load("tsa3.rda")
```

Once you have loaded `tsa3.rda`, all the files will stay in R as long as you save the workspace when you close R (details in §R.2). If you don't save the workspace, you will have to reload it. To see what is included in `tsa3.rda` type

```
ls()           # to get a listing of your objects, and
tsa3.version   # to check the version
```

Periodically check that your version matches the latest version number (year-month-day) on the website. Please note that `tsa3.rda` is subject to change.

You are free to use the data or to alter the scripts in any way you see fit. We only have two requests: reference the text if you use something from it, and contact us if you find any errors or have suggestions for improvement of the code.

[1] See §R.2, page 567, on how to get the current working directory and how to change it, or page 569 on how to read files from other directories.

R.1.1 Included Data Sets

The data sets included in `tsa3.rda`, listed by the chapter in which they are first presented, are as follows.

CHAPTER 1

- jj - Johnson & Johnson quarterly earnings per share, 84 quarters (21 years) measured from the first quarter of 1960 to the last quarter of 1980.
- EQ5 - Seismic trace of an earthquake [two phases or arrivals along the surface, the primary wave ($t = 1, \ldots, 1024$) and the shear wave ($t = 1025, \ldots, 2048$)] recorded at a seismic station.
- EXP6 - Seismic trace of an explosion (similar details as EQ5).
- gtemp - Global mean land-ocean temperature deviations (from 1951-1980 average), measured in degrees centigrade, for the years 1880-2009; data taken from `http://data.giss.nasa.gov/gistemp/graphs/`
- fmri1 - A data frame that consists of FMRI BOLD signals at eight locations (in columns 2-9, column 1 is time period), when a stimulus was applied for 32 seconds and then stopped for 32 seconds. The signal period is 64 seconds and the sampling rate was one observation every 2 seconds for 256 seconds ($n = 128$).
- soi - Southern Oscillation Index (SOI) for a period of 453 months ranging over the years 1950-1987.
- rec - Recruitment (number of new fish) associated with SOI.
- speech - A small .1 second (1000 points) sample of recorded speech for the phrase $aaa \cdots hhh$.
- nyse - Returns of the New York Stock Exchange (NYSE) from February 2, 1984 to December 31, 1991.
- soiltemp - A 64×36 matrix of surface soil temperatures.

CHAPTER 2

- oil - Crude oil, WTI spot price FOB (in dollars per barrel), weekly data from 2000 to mid-2010. For definitions and more details, see `http://tonto.eia.doe.gov/dnav/pet/pet_pri_spt_s1_w.htm`.
- gas - New York Harbor conventional regular gasoline weekly spot price FOB (in cents per gallon) over the same time period as `oil`.
- varve - Sedimentary deposits from one location in Massachusetts for 634 years, beginning nearly 12,000 years ago.
- cmort - Average weekly cardiovascular mortality in Los Angeles County; 508 six-day smoothed averages obtained by filtering daily values over the 10 year period 1970-1979.
- tempr - Temperature series corresponding to cmort.
- part - Particulate series corresponding to cmort.
- so2 - Sulfur dioxide series corresponding to cmort.

CHAPTER 3

- prodn - Monthly Federal Reserve Board Production Index (1948-1978, $n = 372$ months).
- unemp - Unemployment series corresponding to prodn.
- ar1boot - Data used in Example 3.35 on page 137.

`gnp` - Quarterly U.S. GNP from 1947(1) to 2002(3), $n = 223$ observations.

`birth` - Monthly live births (adjusted) in thousands for the United States, 1948-1979.

Chapter 4

`sunspotz` - Biannual smoothed (12-month moving average) number of sunspots from June 1749 to December 1978; $n = 459$. The "z" on the end is to distinguish this series from the one included with R (called `sunspots`).

`salt` - Salt profiles taken over a spatial grid set out on an agricultural field, 64 rows at 17-ft spacing.

`saltemp` - Temperature profiles corresponding to `salt`.

Chapter 5

`arf` - 1000 simulated observations from an ARFIMA(1, 1, 0) model with $\phi = .75$ and $d = .4$.

`flu` - Monthly pneumonia and influenza deaths per 10,000 people in the United States for 11 years, 1968 to 1978.

`sales` - Sales (with `lead`, a leading indicator), 150 months; taken from Box & Jenkins (1970).

`lead` - See `sales`.

`econ5` - Data set containing quarterly U.S. unemployment, GNP, consumption, and government and private investment, from 1948-III to 1988-II.

Chapter 6

`ar1miss` - Data for Problem 6.14 on page 403.

`gtemp2` - Similar to `gtemp` but the data are based only on surface air temperature data obtained from meteorological stations.

`qinfl` - Quarterly inflation rate in the Consumer Price Index from 1953-I to 1980-II, $n = 110$ observations; from Newbold and Bos (1985).

`qintr` - Quarterly interest rate recorded for Treasury bills over the same period as `qinfl`.

`WBC` - Measurements made for 91 days on the three variables, log(white blood count) [`WBC`], log(platelet) [`PLT`] and hematocrit [`HCT`]; taken from Jones (1984).

`PLT` - See `WBC`.

`HCT` - See `WBC`.

Chapter 7

`beamd` - Infrasonic signals from a nuclear explosion. This is a data frame consisting of three columns (which are not time series objects) that are data from different channels. The series names are `sensor1, sensor2, sensor3`. See Example 7.2 on page 421 for more information.

`bnrf1ebv` - Nucleotide sequence of the BNRF1 gene of the Epstein-Barr virus (EBV): 1=A, 2=C, 3=G, 4=T. The data are used in §7.9.

`bnrf1hvs` - Nucleotide sequence of the BNRF1 gene of the herpes virus saimiri (HVS); same codes as for EBV.

`fmri` - Data (as a vector list) from an fMRI experiment in pain, listed by location and stimulus. The specfic locations of the brain where the signal was measured were [1] Cortex 1: Primary Somatosensory, Contralateral, [2] Cortex 2: Primary Somatosensory, Ipsilateral, [3] Cortex 3: Secondary Somatosensory, Contralateral, [4] Cortex 4: Secondary Somatosensory, Ipsilateral, [5] Caudate, [6] Thalamus 1: Contralateral, [7] Thalamus 2: Ipsilateral, [8] Cerebellum 1: Contralateral and [9] Cerebellum 2: Ipsilateral. The stimuli (and number of subjects in each condition) are [1] Awake-Brush (5 subjects), [2] Awake-Heat (4 subjects), [3] Awake-Shock (5 subjects), [4] Low-Brush (3 subjects), [5] Low-Heat (5 subjects), and [6] Low-Shock (4 subjects). Issue the command `summary(fmri)` for further details. As an example, `fmri$L1T6` (location 1, treatment 6) will show the data for the four subjects receiving the Low-Shock treatment at the Cortex 1 location; note that `fmri[[6]]` will display the same data. See Examples 7.7–7.9 for examples.

`climhyd` - Lake Shasta inflow data; see Example 7.1. This is a data frame with column names: `Temp, DewPt, CldCvr, WndSpd, Precip, Inflow`.

`eqexp` - This is a data frame of the earthquake and explosion seismic series used throughout the text. The matrix has 17 columns, the first eight are earthquakes, the second eight are explosions, and the last column is the Novaya Zemlya series. The column names are: `EQ1, EQ2,...,EQ8; EX1, EX2,...,EX8; NZ`.

R.1.2 Included Scripts

The following scripts are included in `tsa3.rda`. At the end of the description of each script, a text example that demonstrates the use of the script is given.

`lag.plot2(series1, series2, max.lag=0, corr=TRUE, smooth=TRUE)`

Produces a grid of scatterplots of one series versus another. If (x_t, y_t) is a vector time series, then `lag.plot2(x,y,m)` will generate a grid of scatterplots of x_{t-h} versus y_t for $h = 0, 1, ..., m$, along with the cross-correlation values (`corr=TRUE`) and a lowess fit (`smooth=TRUE`) assuming x_t is in `x` and y_t is in `y`. Note that the first series, x_t, is the one that gets lagged. If you just want the scatterplots and nothing else, then use `lag.plot2(x,y,m,corr=FALSE,smooth=FALSE)`. See Example 2.7 on page 64 for a demonstration.

`lag.plot1(series, max.lag=1, corr=TRUE, smooth=TRUE)`

Produces a grid of scatterplots of a series versus lagged values of the series. Similar to `lag.plot2`, the call `lag.plot1(x,m)` will generate a grid of scatterplots of x_{t-h} versus x_t for $h = 1, ..., m$, along with the autocorrelation values (`corr=TRUE`) and a lowess fit (`smooth=TRUE`). The defaults are the same as `lag.plot2`; if you don't want either the correlation values or the lowess fit, you can either use `lag.plot1(x,m,corr=FALSE,smooth=FALSE)` or R's `lag.plot`. See Example 2.7 on page 64 for a demonstration.

`acf2(series, max.lag=NULL)`

Produces a simultaneous plot (and a printout) of the sample ACF and PACF on the same scale. If `x` contains n observations, `acf2(x)` will print and plot the ACF and PACF of `x` to the default lag of $\sqrt{n} + 10$ (unless n is smaller than 50). The number of lags may be specified, e.g., `acf2(x, 33)`. See Example 3.17 on page 108.

```
sarima(series, p, d, q, P=0, D=0, Q=0, S=-1, details=TRUE,
       tol=sqrt(.Machine$double.eps), no.constant=FALSE)
```

Fits ARIMA models including diagnostics in a short command. If your time series is in `x` and you want to fit an ARIMA(p, d, q) model to the data, the basic call is `sarima(x,p,d,q)`. The results are the parameter estimates, standard errors, AIC, AICc, BIC (as defined in Chapter 2) and diagnostics. To fit a seasonal ARIMA model, the basic call is `sarima(x,p,d,q,P,D,Q,S)`. So, for example, `sarima(x,2,1,0)` will fit an ARIMA$(2, 1, 0)$ model to the series in `x`, and `sarima(x,2,1,0,0,1,1,12)` will fit a seasonal ARIMA$(2, 1, 0) \times (0, 1, 1)_{12}$ model to the series in `x`. If you want to look at the innovations (i.e., the residuals) from the fit, they're stored in `innov`.

There are three additional options that can be included in the call.

- `details` turns on/off the output from the nonlinear optimization routine, which is `optim`. The default is `TRUE`, use `details=FALSE` to turn off the output; e.g., `sarima(x,2,1,0,details=FALSE)`.
- `tol` controls the relative tolerance (reltol) used to assess convergence in `sarima` and `sarima.for`. The default is `tol=sqrt(.Machine$double.eps)`, the R default. For details, see the help file for `optim` under the `control` arguments. For example, `sarima(rec,2,0,0,tol=.0001)` will speed up the convergence. If there are many parameters to estimate (e.g., seasonal models), the analysis may take a long time using the default.
- `no.constant` can control whether or not `sarima` includes a constant in the model. In particular, with `sarima`, if there is no differencing ($d = 0$ and $D = 0$) you get the mean estimate. If there's differencing of order one (either $d = 1$ or $D = 1$, but not both), a constant term is included in the model; this may be overridden by setting this to `TRUE`; e.g., `sarima(x,1,1,0,no.constant=TRUE)`. In any other situation, no constant or mean term is included in the model. The idea is that if you difference more than once ($d + D > 1$), any drift is likely to be removed.

See Examples 3.38, 3.39, 3.40, 3.42, and 3.46 on pages 145–159 for demonstrations.

```
sarima.for(series, n.ahead, p, d, q, P=0, D=0, Q=0, S=-1,
           tol=sqrt(.Machine$double.eps), no.constant=FALSE)
```

Gives ARIMA forecasts. Similar to `sarima`, to forecast `n.ahead` time points from an ARIMA fit to the data in `x`, the form is `sarima.for(x, n.ahead, p, d, q)` or `sarima.for(x, n.ahead, p, d, q, P, D, Q, S)` for a seasonal model. For example, `sarima.for(x,5,1,0,1)` will forecast five time points ahead for an ARMA(1,1) fit to `x`. The output prints the forecasts and the standard errors of the forecasts, and supplies a graphic of the forecast with ± 2 prediction error

bounds. The options `tol` and `no.constant` are also available. See Example 3.46 on page 159.

`spec.arma(ar=0, ma=0, var.noise=1, n.freq=500, ...)`

Gives the ARMA spectrum (on a log scale), tests for causality, invertibility, and common zeros. The basic call is `spec.arma(ar, ma)` where `ar` and `ma` are vectors containing the model parameters. Use `log="no"` if you do not want the plot on a log scale. If the model is not causal or invertible an error message is given. If there are common zeros, a spectrum will be displayed and a warning will be given; e.g., `spec.arma(ar= .9, ma= -.9)` will yield a warning and the plot will be the spectrum of white noise. The variance of the noise can be changed with `var.noise`. Finally, the frequencies and the spectral density ordinates are returned invisibly, e.g., `spec.arma(ar=.9)$freq` and `spec.arma(ar=.9)$spec`, if you're interested in the actual values. See Example 4.6 on page 184.

`LagReg(input, output, L=c(3,3), M=20, threshold=0, inverse=FALSE)`

Performs lagged regression as discussed in Chapter 4, §4.10. For a bivariate series, `input` is the input series and `output` is the output series. The degree of smoothing for the spectral estimate is given by `L`; see `spans` in the help file for `spec.pgram`. The number of terms used in the lagged regression approximation is given by `M`, which must be even. The `threshold` value is the cut-off used to set small (in absolute value) regression coefficients equal to zero (it is easiest to run LagReg twice, once with the default threshold of zero, and then again after inspecting the resulting coefficients and the corresponding values of the CCF). Setting `inverse=TRUE` will fit a forward-lagged regression; the default is to run a backward-lagged regression. The script is based on code that was contributed by Professor Doug Wiens, Department of Mathematical and Statistical Sciences, University of Alberta. See Example 4.24 on page 244 for a demonstration.

`SigExtract(series, L=c(3,3), M=50, max.freq=.05)`

Performs signal extraction and optimal filtering as discussed in Chapter 4, §4.11. The basic function of the script, and the default setting, is to remove frequencies above 1/20 (and, in particular, the seasonal frequency of 1 cycle every 12 time points). The time series to be filtered is `series`, and its sampling frequency is set to unity ($\Delta = 1$). The values of `L` and `M` are the same as in `LagReg` and `max.freq` denotes the truncation frequency, which must be larger than 1/M. The filtered series is returned silently; e.g., `f.x = SigExtract(x)` will store the extracted signal in `f.x`. The script is based on code that was contributed by Professor Doug Wiens, Department of Mathematical and Statistical Sciences, University of Alberta. See Example 4.25 on page 249 for a demonstration.

`Kfilter0(n, y, A, mu0, Sigma0, Phi, cQ, cR)`

Returns the filtered values in Property 6.1 on page 326 for the state-space model, (6.1)–(6.2). In addition, the script returns the evaluation of the likelihood at the given parameter values and the innovation sequence. The inputs are `n`: number of

observations; `y`: data matrix; `A`: observation matrix (assumed constant); `mu0`: initial state mean; `Sigma0`: initial state variance-covariance matrix; `Phi`: state transition matrix; `cQ`: Cholesky decomposition of Q [`cQ=chol(Q)`]; `cR`: Cholesky decomposition of R [`cR=chol(R)`]. Note: The script requires only that Q or R may be reconstructed as `t(cQ)%*%cQ` or `t(cR)%*%cR`, which offers a little more flexibility than requiring Q or R to be positive definite. For demonstrations, see Example 6.6 on page 336, Example 6.8 on page 342, and Example 6.10 on page 350.

`Ksmooth0(n, y, A, mu0, Sigma0, Phi, cQ, cR)`

Returns both the filtered values in Property 6.1 on page 326 and the smoothed values in Property 6.2 on page 330 for the state-space model, (6.1)–(6.2). The inputs are the same as `Kfilter0`. For demonstrations, see Example 6.5 on page 331, and Example 6.10 on page 350.

`EM0(n, y, A, mu0, Sigma0, Phi, cQ, cR, max.iter=50, tol=.01)`

Estimation of the parameters in the model (6.1)–(6.2) via the EM algorithm. Most of the inputs are the same as for `Ksmooth0` and the script uses `Ksmooth0`. To control the number of iterations, use `max.iter` (set to 50 by default) and to control the relative tolerance for determining convergence, use `tol` (set to .01 by default). For a demonstration, see Example 6.8 on page 342.

`Kfilter1(n, y, A, mu0, Sigma0, Phi, Ups, Gam, cQ, cR, input)`

Returns the filtered values in Property 6.1 on page 326 for the state-space model, (6.3)–(6.4). In addition, the script returns the evaluation of the likelihood at the given parameter values and the innovation sequence. The inputs are `n`: number of observations; `y`: data matrix; `A`: observation matrix, an array with `dim=c(q,p,n)`; `mu0`: initial state mean; `Sigma0`: initial state variance-covariance matrix; `Phi`: state transition matrix; `Ups`: state input matrix; `Gam`: observation input matrix; `cQ`: Cholesky decomposition of Q; `cR`: Cholesky decomposition of R [the note in `Kfilter0` applies here]; `input`: matrix of inputs having the same row dimension as `y`. Set `Ups` or `Gam` or `input` to 0 (zero) if they are not used. For demonstrations, see Example 6.7 on page 338 and Example 6.9 on page 348.

`Ksmooth1(n, y, A, mu0, Sigma0, Phi, Ups, Gam, cQ, cR, input)`

Returns both the filtered values in Property 6.1 on page 326 and the smoothed values in Property 6.2 on page 330 for the state-space model, (6.3)–(6.4). The inputs are the same as `Kfilter1`. See Example 6.7 on page 338 and Example 6.9 on page 348.

`EM1(n, y, A, mu0, Sigma0, Phi, Ups, Gam, cQ, cR, input, max.iter=50, tol=.01)`

Estimation of the parameters in the model (6.3)–(6.4) via the EM algorithm. Most of the inputs are the same as for `Ksmooth1` and the script uses `Ksmooth1`. To control the number of iterations, use `max.iter` (set to 50 by default) and to control the relative tolerance for determining convergence, use `tol` (set to .01 by default). For a demonstration, see Example 6.12 on page 357.

`Kfilter2(n, y, A, mu0, Sigma0, Phi, Ups, Gam, Theta, cQ, cR, S, input)`

Returns the filtered values in Property 6.5 on page 354 for the state-space model, (6.97)–(6.99). In addition, the script returns the evaluation of the likelihood at the given parameter values and the innovation sequence. The inputs are similar to `Kfilter1`, except that the noise covariance matrix, `S` must be included. For demonstrations, see Example 6.11 on page 356 and Example 6.13 on page 361.

`Ksmooth2(n, y, A, mu0, Sigma0, Phi, Ups, Gam, Theta, cQ, cR, S, input)`

This is the smoother companion to `Kfilter2`.

`SVfilter(n, y, phi0, phi1, sQ, alpha, sR0, mu1, sR1)`

Performs the special case switching filter for the stochastic volatility model, (6.173), (6.175)–(6.176). The state parameters are `phi0, phi1, sQ` $[\phi_0, \phi_1, \sigma_w]$, and `alpha, sR0, mu1, sR1` $[\alpha, \sigma_0, \mu_1, \sigma_1]$ are observation equation parameters as presented in Section 6.9. See Example 6.18 page 380 and Example 6.19 page 383.

```
mvspec(x, spans = NULL, kernel = NULL, taper = 0, pad = 0, fast = TRUE,
        demean = TRUE, detrend = FALSE, plot = FALSE,
        na.action = na.fail, ...)
```

This is `spec.pgram` with a few changes in the defaults and written so you can extract the estimate of the multivariate spectral matrix as `fxx`. For example, if `x` contains a p-variate time series (i.e., the p columns of `x` are time series), and you issue the command `spec = mvspec(x, spans=3)` say, then `spec$fxx` is an array with dimensions `dim=c(p,p,nfreq)`, where `nfreq` is the number of frequencies used. If you print `spec$fxx`, you will see `nfreq` $p \times p$ spectral matrix estimates. See Example 7.12 on page 461 for a demonstration.

`FDR(pvals, qlevel=0.001)`

Computes the basic false discovery rate given a vector of p-values; see Example 7.4 on page 427 for a demonstration.

`stoch.reg(data, cols.full, cols.red, alpha, L, M, plot.which)`

Performs frequency domain stochastic regression discussed in §7.3. Enter the entire data matrix (`data`), and then the corresponding columns of input series in the full model (`cols.full`) and in the reduced model (`cols.red`; use `NULL` if there are no inputs under the reduced model). The response variable should be the *last* column of the data matrix, and this need not be specified among the inputs. Other arguments are `alpha` (test size), `L` (smoothing), `M` (number of points in the discretization of the integral) and `plot.which = coh` or `F.stat`, to plot either the squared-coherencies or the F-statistics. The coefficients of the impulse response function are returned and plotted. The script is based on code that was contributed by Professor Doug Wiens, Department of Mathematical and Statistical Sciences, University of Alberta. See Example 7.1 on page 417 for a demonstration.

R.2 Getting Started

If you are experienced with R/S-PLUS you can skip this section, and perhaps the rest of this appendix. Otherwise, it is essential to have R up and running before you start this tutorial. The best way to use the rest of this appendix is to start up R and enter the example code as it is presented. Also, you can use the results and help files to get a better understanding of how R works (or doesn't work). The character # is used for comments.

The convention throughout the text is that R code is in `typewriter font` with a small line number in the left margin. Get comfortable, then start her up and try some simple tasks.

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

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

```
1 x <- 1 + 2  # put 1 + 2 in object x
2 x = 1 + 2   # same as above with fewer keystrokes
3 1 + 2 -> x  # same
4 x           # view object x
  [1] 3
5 (y = 9*3)   # put 9 times 3 in y and view the result
  [1] 27
6 (z = rnorm(5,0,1))  # put 5 standard normals into z and print z
  [1]  0.96607946  1.98135811 -0.06064527  0.31028473  0.02046853
```

To list your objects, remove objects, get help, find out which directory is current (or to change it) or to quit, use the following commands:

```
ls()       # list all objects
  [1] "dummy" "mydata" "x" "y" "z"
ls(pattern = "my")  # list every object that contains "my"
  [1] "dummy" "mydata"
rm(dummy)  # remove object "dummy"
help.start()  # html help and documentation (use it)
help(exp)  # specific help  (?exp is the same)
getwd()    # get working directory
setwd("/TimeSeries/")  # change working directory to TimeSeries
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 all the work you have done so far, and load it up when you next start R. Our suggestion is to answer "yes" even though you will also be loading irrelevant past analyses every time you start R. Keep in mind that you can remove items via `rm()`. If you do not save the workspace, you will have to reload `tsa3.rda` as described in §R.1.

To create your own data set, 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
```

It is worth pointing out R's *recycling rule* for doing arithmetic. The rule is extremely helpful for shortening code, but it can also lead to mistakes if you are not careful. Here are some examples.

```
x = c(1, 2, 3, 4);  y = c(2, 4, 6, 8);  z = c(10, 20)
x*y  # it's 1*2, 2*4, 3*6, 4*8
  [1]  2  8 18 32
x/z  # it's 1/10, 2/20, 3/10, 4/20
  [1] 0.1 0.1 0.3 0.2
```

```
4 x+z  # guess
  [1] 11 22 13 24
```

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

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

```
1 dummy = scan("dummy.dat")  # if TimeSeries is the working directory
2 (dummy = scan("/TimeSeries/dummy.dat"))  # if not, do this
  Read 10 items
  [1] 1 2 3 2 1 9 0 2 1 0
3 (dummy = read.table("/TimeSeries/dummy.dat"))
  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

```
4 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 in line 4 above.

```
5 colnames(dummy) = c("Dog", "Cat", "Rat", "Pig", "Man")
6 attach(dummy)
7 Cat
  [1] 2 0
```

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 using line 5 above; type `?read.table` for further information.

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. The code is:

```
1 crazy <- function(num) {
2   x <- rep(NA, num)
3   for (n in 1:num) x[n] <- mean(rcauchy(n))
4   plot(x, type="l", xlab="sample size", ylab="sample mean")
5   }
```

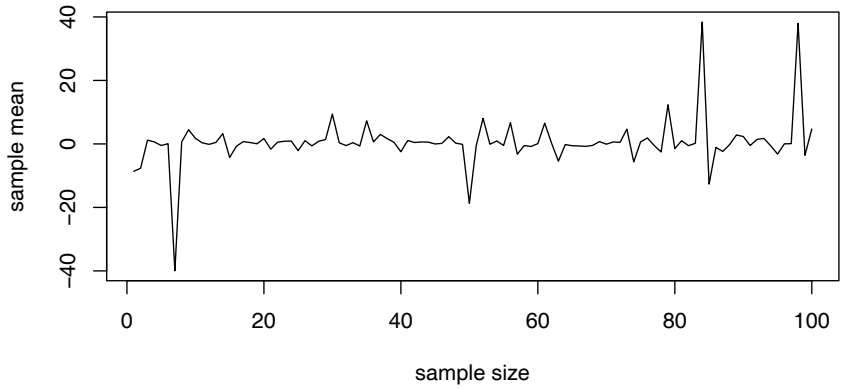

Fig. R.1. Crazy example.

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 a vector, `x`, of `num` missing values `NA`, 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 Figure R.1). The fifth line closes the function. To use `crazy` with a limit sample size of 100, for example, type

```
6 crazy(100)
```

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

You may want to use one of the R packages. In this case you have to first download the package and then install it. For example,

```
1 install.packages(c("wavethresh", "tseries"))
```

will download and install the packages `wavethresh` that we use in Chapter 4 and `tseries` that we use in Chapter 5; you will be asked to choose the closest mirror to you. To use a package, you have to load it at each start up of R, for example:

```
2 library(wavethresh)  # load the wavethresh package
```

A good way to get help for a package is to use html help

```
3 help.start()
```

and follow the *Packages* link.

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.3 Time Series Primer

In this section, we give a brief introduction on using R for time series. We assume that `tsa3.rda` 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 (we display the time series attributes in a single row of the output to save space).

```
cbind(x, lag(x), lag(x,-1))
```

```
Time Series:  Start = 0  End = 6  Frequency = 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 this:

```
diff(diff(x))
```

and so on for higher order differencing.

For graphing time series, there a few standard plotting mechanisms that we use repeatedly. If `x` is a time series, then `plot(x)` will produce a time plot. If `x` is not a time series object, then `plot.ts(x)` will coerce it into a time plot as will `ts.plot(x)`. There are differences, which we explore in the following. It would be a good idea to skim the graphical parameters help file (`?par`) while you are here.[2] See Figure R.2 for the resulting graphic.

```
x = -5:5      # x is NOT a time series object
y = 5*cos(x)  # neither is y
op = par(mfrow=c(3,2))  # multifigure setup: 3 rows, 2 cols
plot(x, main="plot(x)")
plot(x, y, main="plot(x,y)")
plot.ts(x, main="plot.ts(x)")
plot.ts(x, y, main="plot.ts(x,y)")
ts.plot(x, main="ts.plot(x)")
ts.plot(ts(x), ts(y), col=1:2, main="ts.plot(x,y)")
par(op)  # reset the graphics parameters  [see footnote]
```

[2] In the plot example, the parameter set up uses `op = par(...)` and ends with `par(op)`; these lines are used to reset the graphic parameters to their previous settings. Please make a note of this because we do not display these commands ad nauseam in the text.

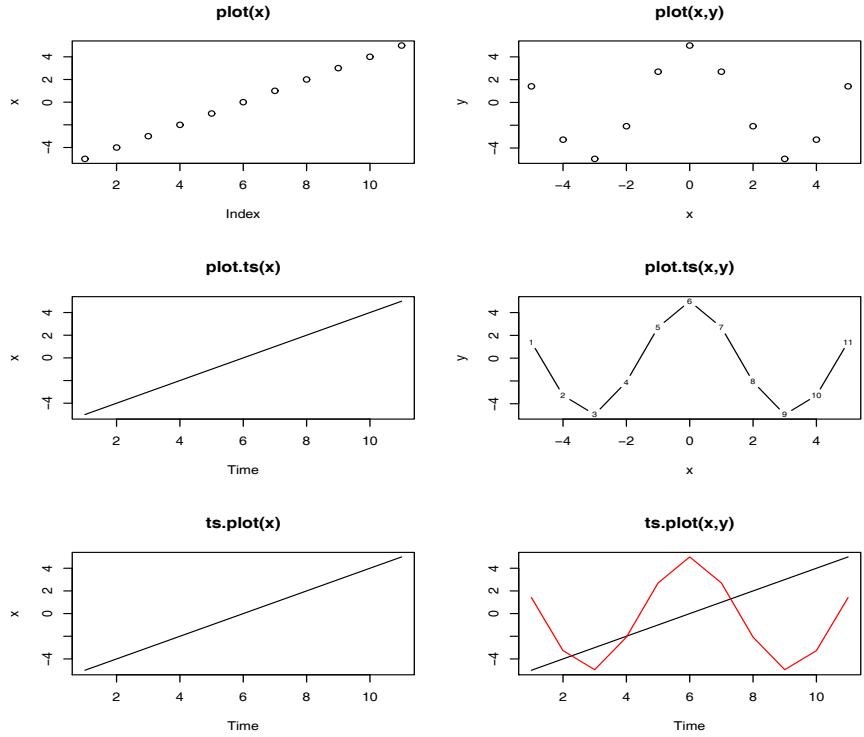

Fig. R.2. Demonstration of different R graphic tools for plotting time series.

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)       # so you can reproduce the result
x = rnorm(10,0,1)
y = x + rnorm(10,0,1)
summary(fit <- lm(y~x))

  Residuals:
      Min       1Q   Median       3Q      Max
  -0.8851 -0.3867   0.1325   0.3896   0.6561

  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 "tie" the series together using `ts.intersect`. If you do not tie the series together, they will not be aligned properly. Please read the warning *Using time series* in the `lm()` help file [`help(lm)`]. Here is an example regressing Chapter 2 data, weekly cardiovascular mortality (`cmort`) on particulate pollution (`part`) at the present value and lagged four weeks (`part4`). First, we create a data frame called `ded` that consists of the three series:

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

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

```
fit = lm(mort~part+part4, data=ded, na.action=NULL)
summary(fit)
```

```
Call: lm(formula=mort~part+part4,data=ded,na.action=NULL)
Residuals:
     Min       1Q   Median       3Q      Max
-22.7429  -5.3677  -0.4136   5.2694  37.8539

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. There is a package called `dynlm` that makes it easy to fit lagged regressions. The basic advantage of `dynlm` is that it avoids having to make a data frame; that is, line 1 would be avoided.

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(rep(1:4, 21))  # make (Q)uarter factors
reg = lm(log(jj)~0 + trend + Q, na.action=NULL)  # no intercept
model.matrix(reg)  # view the model 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
.     .    .  .  .  .
.     .    .  .  .  .
83  10.50  0  0  1  0
84  10.75  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)
```

Next, we'll discuss ARIMA estimation. This gets a bit tricky because R is not useR friendly when it comes to fitting ARIMA models. Much of the story is spelled out in the "R Issues" page of the website for the text. In Chapter 3, we use the scripts `acf2`, `sarima`, and `sarima.for` that are included with `tsa3.Rda`. But we will also show you how to use the scripts included with R.

First, we'll fit an ARMA(1,1) model to some simulated data (with diagnostics and forecasting):

```
set.seed(666)
x = 50 + arima.sim(list(order=c(1,0,1), ar=.9, ma=-.5), n=200)
acf(x); pacf(x) # display sample ACF and PACF ... or ...
acf2(x)         # use our script (no output shown)
(x.fit = arima(x, order = c(1, 0, 1)))  # fit the model
```

```
Call:  arima(x = x, order = c(1, 0, 1))
Coefficients:
         ar1     ma1  intercept
      0.8340  -0.432    49.8960
s.e.  0.0645   0.111     0.2452
sigma^2 estimated as 1.070: log likelihood = -290.79, aic = 589.58
```

Note that the reported `intercept` estimate is an estimate of the mean and *not* the constant. That is, the fitted model is

$$\widehat{x}_t - 49.896 = .834(x_{t-1} - 49.896) + \widehat{w}_t$$

where $\widehat{\sigma}_w^2 = 1.070$. Diagnostics can be accomplished as follows,

```
tsdiag(x.fit, gof.lag=20) # ?tsdiag for details (don't use this!!)
```

but the Ljung-Box-Pierce test is not correct because it does not take into account the fact that the residuals are from a fitted model. If the analysis is repeated using the `sarima` script, a partial output would look like the following (`sarima` will also display the correct diagnostics as a graphic; e.g., see Figure 3.17 on page 151):

```
sarima(x, 1, 0, 1)
```

```
Coefficients:
          ar1     ma1    xmean
       0.8340  -0.432  49.8960
s.e.   0.0645   0.111   0.2452
sigma^2 estimated as 1.070: log likelihood = -290.79, aic = 589.58
$AIC [1] 1.097494   $AICc  [1] 1.108519   $BIC  [1] 0.1469684
```

Finally, to obtain and plot the forecasts, you can use the following R code:

```
x.fore = predict(x.fit, n.ahead=10)
U = x.fore$pred + 2*x.fore$se   # x.fore$pred holds predicted values
L = x.fore$pred - 2*x.fore$se   # x.fore$se holds stnd errors
miny = min(x,L);  maxy = max(x,U)
ts.plot(x, x.fore$pred, col=1:2, ylim=c(miny, maxy))
lines(U, col="blue", lty="dashed")
lines(L, col="blue", lty="dashed")
```

Using the script `sarima.for`, you can accomplish the same task in one line.

```
sarima.for(x, 10, 1, 0, 1)
```

Example 3.46 on page 159 uses this script.

We close this appendix with a quick spectral analysis. This material is covered in detail in Chapter 4, so we will not discuss this example in much detail here. We will simulate an AR(2) and then estimate the spectrum via nonparametric and parametric methods. No graphics are shown, but we have confidence that you are proficient enough in R to display them yourself.

```
x = arima.sim(list(order=c(2,0,0), ar=c(1,-.9)), n=2^8) # some data
(u = polyroot(c(1,-1,.9)))  # x is AR(2) w/complex roots
  [1] 0.5555556+0.8958064i 0.5555556-0.8958064i
Arg(u[1])/(2*pi)  # dominant frequency around .16
  [1] 0.1616497
par(mfcol=c(4,1))
plot.ts(x)
spec.pgram(x,spans=c(3,3),log="no") # nonparametric spectral estimate
spec.ar(x, log="no")                # parametric spectral estimate
spec.arma(ar=c(1,-.9), log="no")    # true spectral density
```

The script `spec.arma` is included in `tsa3.rda`. Also, see `spectrum` as an alternative to `spec.pgram`. Finally, note that R tapers and logs by default, so if you simply want the periodogram of a series, the command is `spec.pgram(x, taper=0, fast=FALSE, detrend=FALSE, log="no")`. If you just asked for `spec.pgram(x)`, you would not get the RAW periodogram because the data are detrended, possibly padded, and tapered, even though the title of the resulting graphic would say *Raw Periodogram*. An easier way to get a raw periodogram is:

```
per = abs(fft(x))^2/length(x)
```

This final example points out the importance of knowing the defaults for the R scripts you use.