Namir's R 202 Time Series Tutorial

by

Namir Shammas

Table of Contents

"If youth knew; if age could."

Henri Estienne

“Predicting the future is tricky business”

Niels Jensen, cofounder of Borland, at an OOP conference in 1990

Introduction

This tutorial complements *Namir’s R 101 Tutorial, Namir’s R 102 Plotting Tutorial,* and *Namir’s R 201 RegressionTutorial*. This tutorial focuses a few R functions that perform various time series analysis.

My hope is that this tutorial gets your feet wet, so to speak. I encourage you to further experiment with the different time series functions to learn more about how to customize the results they produce. The bottom line is that learning any system must involve working and tinkering with that system. Cursory glances at R documentation or books contribute very little to learning R proficiently.

This tutorial assumes that you are familiar with the theoretical background of time series, especially the Box-Jenkins ARIMA models. If you are new to R, I recommend the tutorials that I mentioned above as well as many other tutorials that you can easily find by searching the web--which is probably how you find this tutorial. The more you know about R the more comfortable you get with working with R functions and reading the code in this tutorial.

This tutorial relies completely on simulated time series. As such, each time you run the time series simulator functions, you get a different time series. Therefore, you should keep in mind that just about all of the graphs in this tutorial are unique and that it is highly unlikely that you, or even I, can reproduce them on our computers.

The Time Series Type

R offers the time series data type that is used in time series calculations. A variable of the time series type or class is essentially a vector that is prepped for time series functions.

The function ts() creates a univariate time series (which I will simply call time series in this tutorial) from a vector. The declaration for this function is:

```
ts(data = NA, start = 1, end = numeric(0), frequency = 1,
   deltat = 1, ts.eps = getOption("ts.eps"), class = , names = )
```

The parameter **data** is a vector, matrix, or data frame that supplies the time series values. The parameter **start** represents the time of the initial observation. The parameter **end** is the time for the last value in the time series. The parameter **frequency** is the number of time series values in each time unit. The parameter **deltat** is the fraction of the sampling period between successive observations. For example the value 1/4 represents quarterly data. When calling function ts() you can provide either the parameter frequency or deltat, but not both. The parameter **ts.esp** is the value used for time comparison. The parameter **class** designates the class you tag to the result, or none if you specify NULL or "none". The default **parameter** value is "ts" for a single series, c("mts", "ts") for multiple series. The parameter **names** is a string vector of names for the series in a multiple series.

Here is an example for using the ts() function. Enter the following function in a script window and save it as file ts.creat1.r. Then load the function using the source() function:

```
ts.create1 = function(n=1000, price.init=300, price.mean=20, price.sd=20)
{
  # n is the number of time series values to generate
  # price.init is the initial stock price
  # price.mean is the mean for price fluctutation
  # price.sd is the standard deviation for price fluctuation

  # Function returns time series

  t = 1:n
  price = rep(0, n)
  price[1] = price.init
  for (i in 2:n) {
    price[i] = 0.95 * price[i-1] + rnorm(1, price.mean, price.sd)
  }
  return (ts(price))
}
```

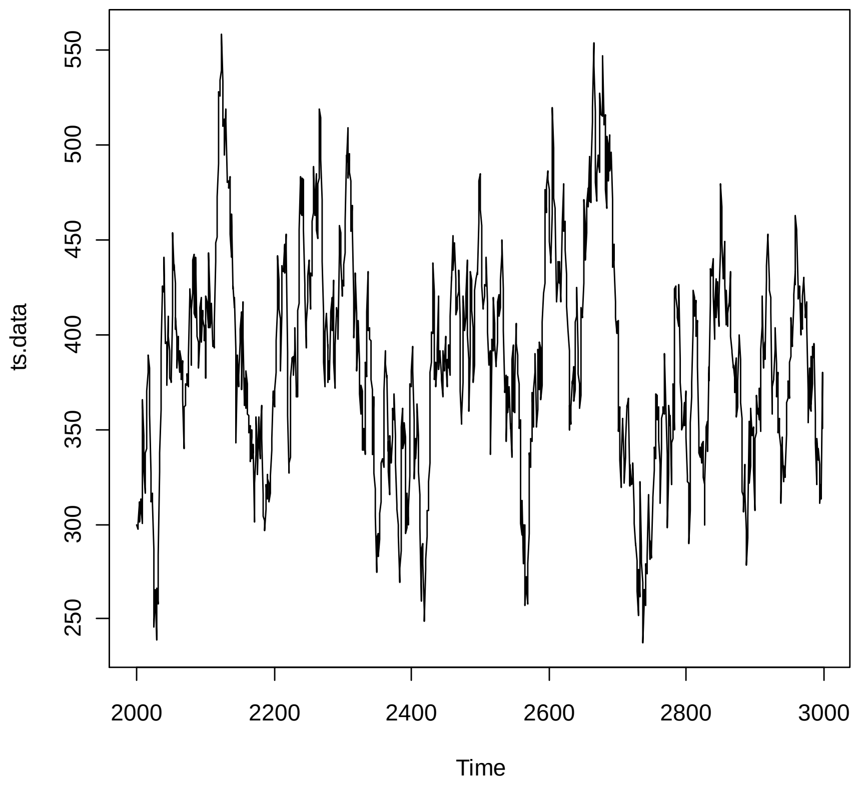
The above function creates a vector of stock prices and converts that vector into a time series. The conversion occurs in the function's return statement. The function creates by default 1000 values with an initial value of 300. The random error introduced by the function is normally distributed and has by default a mean of 20 and standard deviation of 20.

Execute the following commands to test the function ts.create1(), store the time series in variable ts.data, view the data, and plot the data:

```
> ts.data=ts.create1()
> n=length(ts.data)
> ts.data[1:20]
[1] 300.0 297.2 300.1 312.1 301.5 313.0 308.5 300.6 365.8 346.7 327.5 316.7
[13] 327.9 337.8 340.2 369.9 379.8 389.6 382.4 357.8
> ts.data[(n-20):n]
[1] 382.4 362.2 359.5 388.8 363.3 373.8 393.8 387.4 394.9 357.9 342.6 321.1
[13] 345.0 334.1 339.6 331.3 310.8 320.6 313.3 380.3 350.2
> plot(ts.data)
```

The above commands show that the time series start at 300 and samples the first and last 20 values in the time series. Figure 1 shows a plot for the time series which can pass as a graph for a real stock price.

Figure 1. Example of generating a time series with the ts() function.



Trends in Data

Many time series calculations require that the values be stationary. Linear and polynomial regression offer simple tools to determine the presence of trends and their magnitudes. One approach for converting non-stationary data into stationary is to subtract the trend.

Linear Trend

Consider the variable ts.data that you created in the last section. Execute the following commands to create a time variable t, perform a linear regression, display the regression summary, and plot the time series and linear regression line:

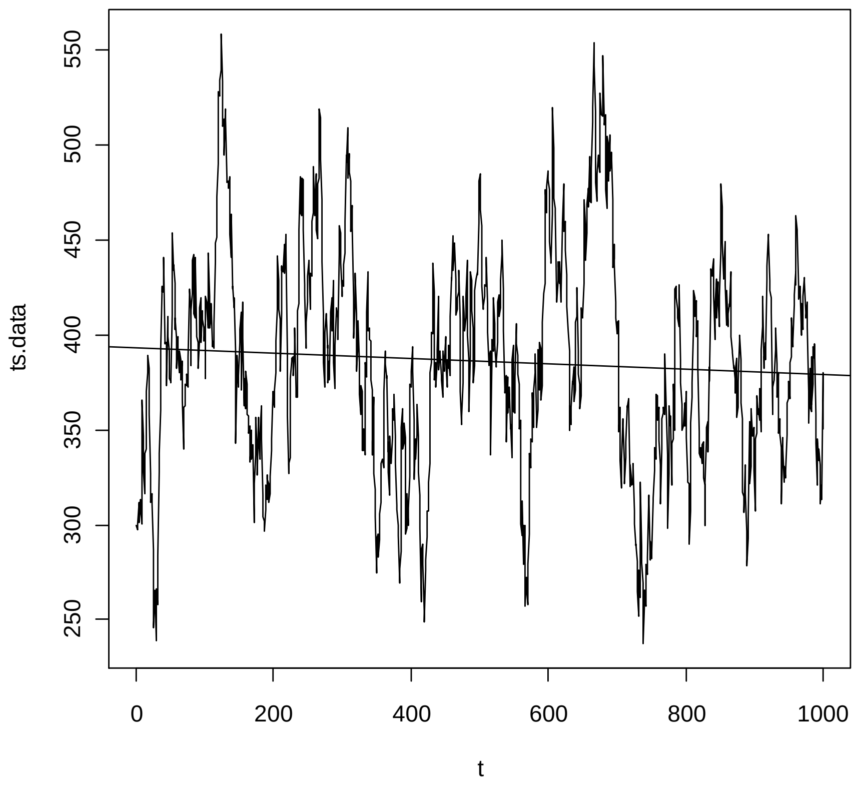
```
> t=1:n
> lr=lm(ts.data~t)
> summary(lr)
> plot(t, ts.data, type="l")
> abline(lr)
```

The above commands yield the following linear regression summary:

```
Call:
lm(formula = ts.data ~ t)

Residuals:
      Min        1Q    Median        3Q       Max
-153.4471  -39.2994   -0.9529   36.9056  170.5421

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept) 392.923139   3.661840 107.302   <2e-16 ***
t            -0.014153   0.006338  -2.233   0.0258 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 57.86 on 998 degrees of freedom
Multiple R-squared: 0.004972,   Adjusted R-squared: 0.003975
F-statistic: 4.987 on 1 and 998 DF,  p-value: 0.02576
```

In addition, Figure 2 shows the time series data and the linear regression line. Examining both the regression summary and the graph one can say that there is a mild trend. Interestingly, the linear regression intercept is a value close to 400, compared ton the actual initial value of 300. While the price jumps quickly above 300, it does create a general losing streak.

Figure 2. Time series and best regression line.



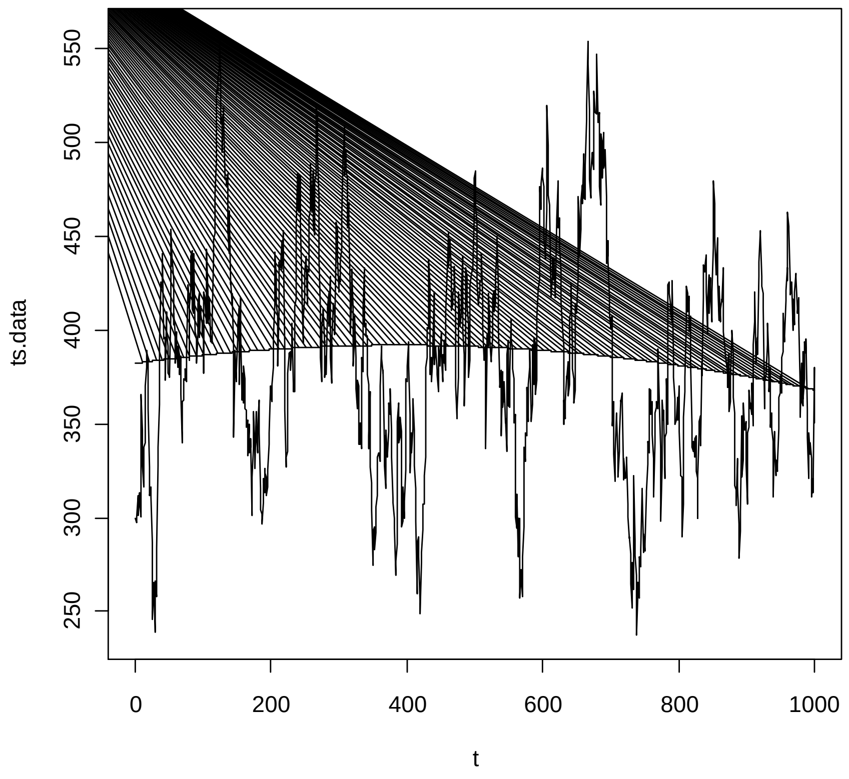
Quadratic Trend

You can also perform a quadratic fit on the time series. The following commands perform a quadratic fit, display the regression summary, and plot the time series showing the quadratic polynomial:

```
> qlr=lm(ts.data~t+I(t^2))
> summary(qlr)
> plot(t, ts.data, type="l")
> t2=seq(t[1], t[n], length.out=100*n)
> ts.data2=qlr$coefficient[1] + qlr$coefficient[2] * t2 + qlr$coefficient[3]
* t2^2
> lines(t2, ts.data2)


Call:
lm(formula = ts.data ~ t + I(t^2))

Residuals:
      Min        1Q    Median        3Q       Max
-146.5033  -38.7215   -0.6742   37.6543  170.8171

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)  3.822e+02  5.484e+00  69.704  < 2e-16 ***
t            4.985e-02  2.530e-02   1.970  0.04909 *
I(t^2)      -6.394e-05  2.447e-05  -2.612  0.00913 **
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 57.69 on 997 degrees of freedom
Multiple R-squared: 0.01174,    Adjusted R-squared: 0.009755
F-statistic: 5.921 on 2 and 997 DF,  p-value: 0.002779
```

Figure 3 shows the time series with the quadratic curve fit.

Figure 3. Time series with a quadratic fit.



Logarithmic Trend

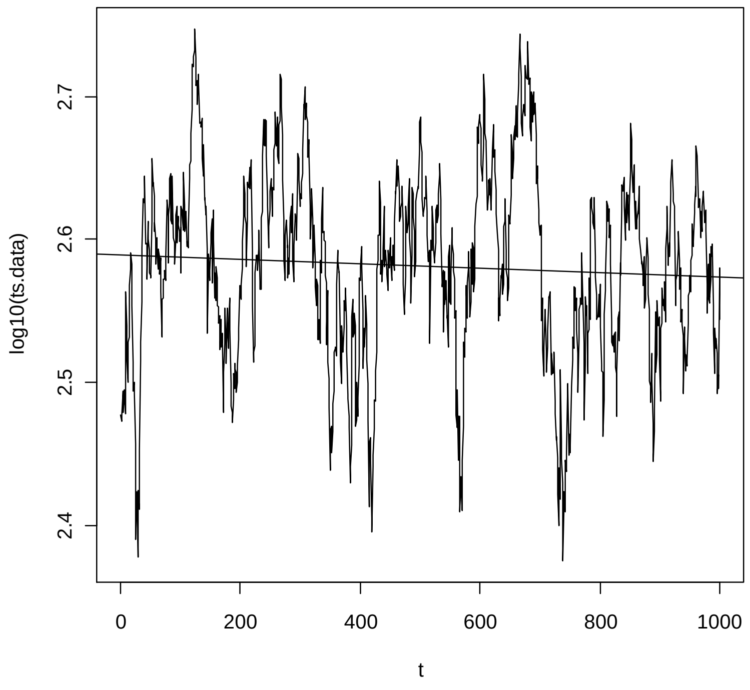
Many researchers work with the logarithm of time series vs. time. With R, such as task is very easy. Execute the following commands to perform a logarithmic fit, display the regression summary, plot the time series using logarithmic values, and draw the regression line:

```
> loglr=lm(log10(ts.data)~t)
> summary(loglr)
> plot(t, log10(ts.data), type="l")
> abline(loglr)

Call:
lm(formula = log10(ts.data) ~ t)

Residuals:
      Min        1Q    Median        3Q       Max
-0.210065 -0.042188  0.003877  0.044814  0.164635

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)  2.589e+00  4.191e-03 617.800   <2e-16 ***
t           -1.519e-05  7.253e-06  -2.094   0.0365 *
---
```

```
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.06621 on 998 degrees of freedom
Multiple R-squared: 0.004376,   Adjusted R-squared: 0.003379
F-statistic: 4.387 on 1 and 998 DF,  p-value: 0.03647
```

Figure 4 shows the logarithmic plot of the time series. The summary and graph show a minor trend.

Figure 4. Plot for log10(time series) vs. time.



Data Smoothing and Filtering

Many time series include noise that comes from errors or are intrinsic to the data. Filtering and smoothing data are techniques that allow researchers to shave off noise in data and thus handle data with less variance.

Moving Averages

Regular moving averages offer one technique for filtering data. The number of data points to average affects the level of filtering. The more data you use to average a time series, the smoother is the resulting values. This comes at a cost that the filtered data is slower in showing changes in the original time series.

R offers the function filter(data, filter) to filter or smooth time series. The parameter **data** represents the time series. The parameter **filter** is the vector for averaging the data. Typically, the argument for this parameter is the call to function rep(1/n, n). The value of n is the number of data points used in calculating the moving average. Hence, the equal weight used for these points is 1/n.

Before I show you an example for filtering data, here is a helper function that allows you to peek at the leading and trailing values of vectors with a large number of data:

```
peek.vector = function(x, numelems=20)
{
  cat("Leading values:", head(x, numelems), "\n")
  cat("Trailing values:", tail(x. numelems), "\n")
  return (TRUE)
}
```

Now, here is an example for filtering time series. The following commands filter the time series:

```
> x.data2=ts.create1(n=200)
> t=1:length(x.data2)
> x.data2.ma41 = filter(x.data2, rep(1/41,41))
> x.data2.ma81 = filter(x.data2, rep(1/81,81))
> peek.vector(x.data2.ma41, 40)
Leading values: NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
336.6710 336.3846 336.8137 337.5357 338.2839 338.9337 339.2462 340.9332
342.9823 345.6708 347.2346 348.2371 349.8961 350.3169 349.3894 348.8601
347.6730 345.5223 344.2594 341.8401
Trailing values: 480.8381 475.2990 469.7596 463.7828 459.0882 455.2695
451.6945 448.2608 445.0154 442.0578 438.822 436.6517 435.3701 434.3328
433.411 431.4713 429.0094 426.689 425.2437 423.393 420.9705 NA NA NA NA NA NA
NA NA NA NA NA NA NA NA NA NA NA NA NA NA
[1] TRUE
> peek.vector(x.data2.ma81, 80)
Leading values: NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA 313.7866 313.2862
313.3117 313.6555 314.4133 315.3717 316.2796 316.6631 317.1994 318.0534
318.4140 318.7339 319.1482 319.2556 319.2213 319.5334 319.904 319.7981
320.0845 320.0872 320.1061 320.3045 320.8658 321.5071 322.3558 322.3893
322.7382 323.1273 323.4931 324.084 324.6595 325.5261 326.5251 328.2574
330.5893 332.6341 334.3673 335.553 337.0365 338.4557 339.6074 341.0256
342.3611 344.0802 345.6984 347.8134 350.6119 353.4352 355.9906 358.6202
360.7462 362.9442 364.9372 367.0116 369.7378 373.1568 376.1509 379.4226
382.6097 385.4002
Trailing values: 385.4002 388.7474 392.7198 396.9679 400.2846 403.5724
407.5415 411.3026 415.4409 419.3368 422.9439 426.448 429.3736 432.3512
435.5632 439.1912 442.6728 445.9482 449.1571 452.2373 454.7874 457.0994
459.6876 462.3037 464.6057 466.1867 467.666 468.9421 470.2512 472.0896
473.7101 474.6078 475.4536 475.9633 476.3945 477.0495 477.6465 477.265
476.8941 476.5610 476.146 475.3747 474.6614 473.9744 473.9593 473.9055
473.6378 474.0137 473.9267 474.1404 474.3959 474.9988 475.561 475.7876
476.3714 476.0656 475.0855 474.5687 474.224 473.9943 473.2497 NA NA NA NA NA
NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
NA NA NA NA NA NA NA NA NA
[1] TRUE
```

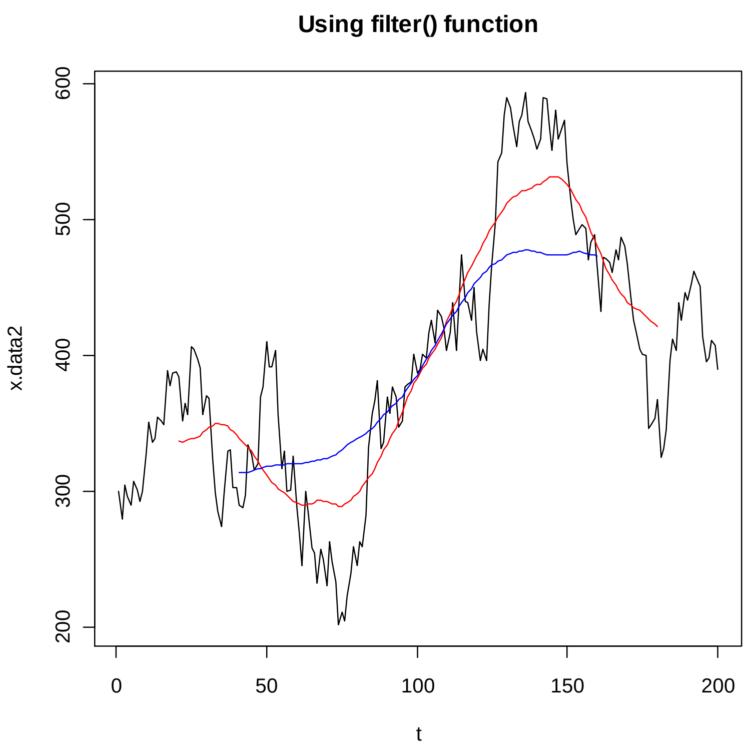
```
> plot(t, x.data2, type="l", main="Using my.filter() function")
> lines(t,x.data2.ma41, col="red")
> lines(t,x.data2.ma81, col="blue")
```

The above commands perform the following tasks:

- Create a time series and store it in the variable x.data2.
- Create the time values for the time series and store them in variable t.
- Calculate the moving average for 41 points and store the result in variable x.data2.ma41.
- Calculate the moving average for 81 points and store the result in variable x.data2.ma81.
- Peek at the data in variables x.data2.ma41 and x.data2.ma81. This task calls function peek.vector() for each of these variables.
- Plot the original time series by calling function plot().
- Add the lines for the filtered values in variables x.data2.ma41 and x.data2.ma81. This task calls function lines().

Figure 5 shows the time series data and the two filtered series. Notice that the filtered data has fewer values than the original time series. The more points you use for averaging with function filter() the fewer filtered values you get!

Figure 5. Filtered data using the filter() function.



Exponential Moving Average

Another approach to moving average is to use exponential moving averages. Such an averaging algorithm does continuous averaging such that the effects of older point gradually fade. The equation used to calculate the exponential moving average is:

$$\text{EMA}[i] = \text{EMA}[i-1] + \alpha\,(\text{Data}[i] - \text{EMA}[i-1])$$

Where $\alpha = 2/(N+1)$ and N is the number of points to average.

Rather than using R's function's for exponential moving average, I will present my own implementation—the function my.filter(). In fact I will use this function to achieve two goals: calculate both regular moving averages and exponential moving averages. The function my.filter() returns filtered data that has the same number of points as the source time series data.

```
my.filter = function(x, average.period=5, calc.ema=FALSE)
{
  # moving average function
  #
  # x the data to be averaged
  # average.period is the number of points to average
  # calc.ema is flag to perform exponential moving average
  #
  # Function returns moving average vector of same length
  # as vector x

  n = length(x)
  m = average.period
  # return x if the moving average period exceeds the number of
  # values in vector x. Also if period is less than 2
  if (m < 2 | m >= n) return (x)
  if (calc.ema) {
    ema = rep(0, n)
    alpha = 2/(m + 1)
    # first value equals x[1]
    ema[1] = x[1]
    # initialize the first few ema elements
    for (i in 2:5)
      ema[i] = mean(x[1:i])
    # calculate exponential moving average from m+1 to n
    for (i in 6:n)
      ema[i] = ema[i-1] + alpha * (x[i] - ema[i-1])
    return (ema)
  }
  else {
    ma = rep(0, n)
    # first value equals x[1]
    ma[1] = x[1]
    # calculate partial leading values for index 2 to m
    for (i in 2:m)
      ma[i] = mean(x[1:i])
    # calculate moving average from m+1 to n
    i1 = 1
    for (i in (m+1):n) {
      i1 = i1 + 1
      ma[i] = mean(x[i1:i])
    }
    return (ma)
  }
}
```

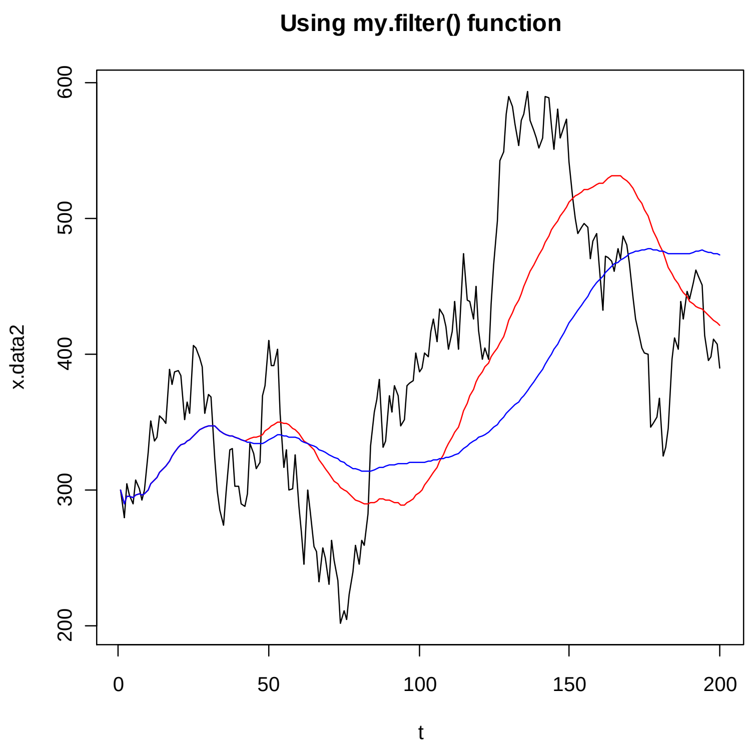
The parameter represents the **x** the data to be averaged. The parameter **average.period** is the number of points to average. The Boolean parameter **calc.ema** is flag that tells the function to work with exponential moving averages. The function calculates tail-end moving averages. Unlike the R function, filter() which calculates central averages, which explains why the graphs based on function filter() have leading and trailing parts missing.

Let's use the function my.filter() to calculate moving averages. Execute the following commands to get the moving average of the time series in variable x.data2:

```
> x.data2.ma41 = my.filter(x.data2, 41)
> x.data2.ma81 = my.filter(x.data2, 81)
> peek.vector(x.data2.ma41, 20)
Leading values: 300 289.9441 294.8550 295.2270 294.0349 296.2689 296.8665
296.3555 296.7248 299.8128 304.4389 307.0641 309.4947 312.7001 315.2929
317.3868 321.5849 324.6910 327.9654 330.9649
Trailing values: 480.8381 475.2990 469.7596 463.7828 459.0882 455.2695
451.6945 448.2608 445.0154 442.0578 438.822 436.6517 435.3701 434.3328
433.411 431.4713 429.0094 426.689 425.2437 423.393 420.9705
[1] TRUE
> peek.vector(x.data2.ma81, 20)
Leading values: 300 289.9441 294.8550 295.2270 294.0349 296.2689 296.8665
296.3555 296.7248 299.8128 304.4389 307.0641 309.4947 312.7001 315.2929
317.3868 321.5849 324.6910 327.9654 330.9649
Trailing values: 476.146 475.3747 474.6614 473.9744 473.9593 473.9055
473.6378 474.0137 473.9267 474.1404 474.3959 474.9988 475.561 475.7876
476.3714 476.0656 475.0855 474.5687 474.224 473.9943 473.2497
[1] TRUE
> plot(t, x.data2, type="l", main="Using my.filter() function")
> lines(t,x.data2.ma41, col="red")
> lines(t,x.data2.ma81, col="blue")
```

Figure 6 shows the filtered data for 41 and 81 points. The graph shows the filtered series run the entire range as does the original time series.

Figure 6. Using function my.filter() to graph moving averages.

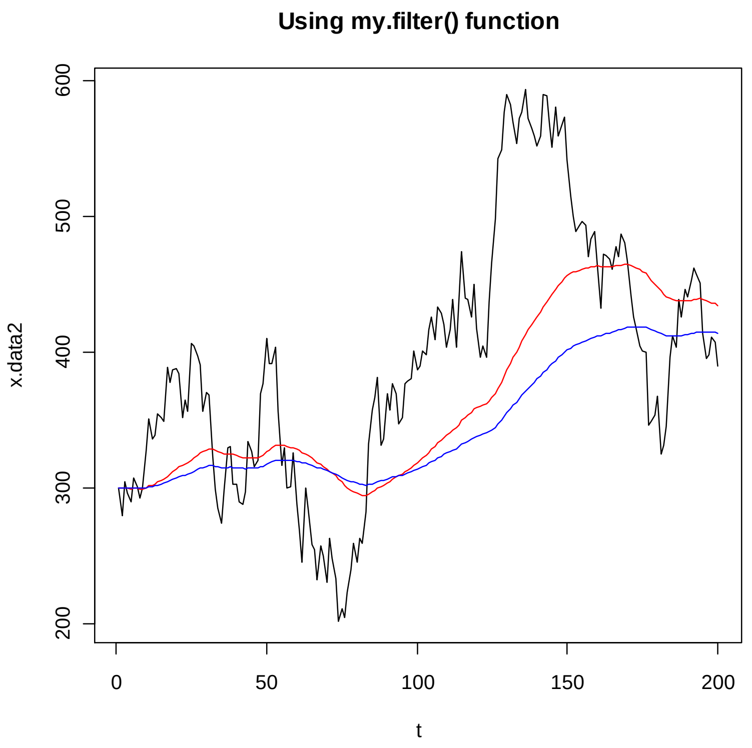


Next, execute the following set of similar commands to calculate and graph the exponential moving averages using function my.filter():

```
> x.data2.ema41 = my.filter(x.data2, 41, ema=TRUE)
> x.data2.ema81 = my.filter(x.data2, 81, ema=TRUE)
> peek.vector(x.data2.ema41, 20)
Leading values: 300 299.5211 299.6439 299.5653 299.3201 299.5134 299.5358
299.3749 299.3821 300.0541 301.2599 302.0857 302.9565 304.1807 305.3095
306.3449 308.3070 309.9543 311.7865 313.6001
Trailing values: 448.2911 445.3472 442.6416 440.3152 439.2694 438.6246
437.7965 437.8302 437.545 437.7477 437.8127 438.1643 438.7309 439.164
439.4469 438.8429 437.7972 436.8472 436.2256 435.5305 434.4425
[1] TRUE
> peek.vector(x.data2.ema81, 20)
Leading values: 300 299.7547 299.8148 299.7724 299.6443 299.7394 299.7481
299.6631 299.6633 300.004 300.6222 301.0530 301.5116 302.1562 302.7591
303.3205 304.3624 305.2542 306.25 307.2464
Trailing values: 415.0147 413.9126 412.9102 412.0812 411.8899 411.8935
411.7954 412.1297 412.297 412.7087 413.0474 413.5295 414.1202 414.6421
415.0861 415.0738 414.8280 414.6216 414.5742 414.4823 414.1817
[1] TRUE
> plot(t, x.data2, type="l", main="Using my.filter() function")
> lines(t,x.data2.ema41, col="red")
> lines(t,x.data2.ema81, col="blue")
```

Figure 7 shows the exponential moving averages.

Figure 7. Using function my.filter() to graph exponential moving averages.



Experiment with changing the number of points to average and study the smoothing effect of the resulting curves.

Working with ARIMA Models

Simulating ARIMA Timer Series

Software toolboxes that deal with the Box-Jenkins ARIMA models invariably include functions that simulate ARIMA-based time series. The value of such functions for testing cannot be overstated.

R offers the function arima.sim() to simulate ARIMA-based time series. the declaration for this function is:

```
arima.sim(model, n, rand.gen = rnorm, innov = rand.gen(n, ...),
          n.start = NA, start.innov = rand.gen(n.start, ...),
          ...)
```

The parameter **model** is a list with component AR and/or ma that specify the AR and MA coefficients, respectively. The function considers an empty list as the ARIMA(0, 0, 0) model—pure white noise. The parameter **n** specifies the length of output series, before reversedifferencing. The parameter **rand.gen** specifies a function to generate the innovations. The parameter **innov** is an optional times series of innovations. If you do not supply an argument for this parameter the function uses parameter rand.gen instead. The parameter **n.start** specifies the length of 'burn-in' period. The default value of NA causes the function to compute a suitable value for parameter nstart. The parameter **start.innov** is an optional times series of innovations to be used for the burn-in period. If you supply an argument for this parameter, you must also supply an argument for at least n.start values (and the function computes n.start by default).

Using function arima.sim(), despite the long list of parameters, is easy and straightforward. Here is the function demo.arima.sim100() which plots four sample for the ARIMA(1,00) model:

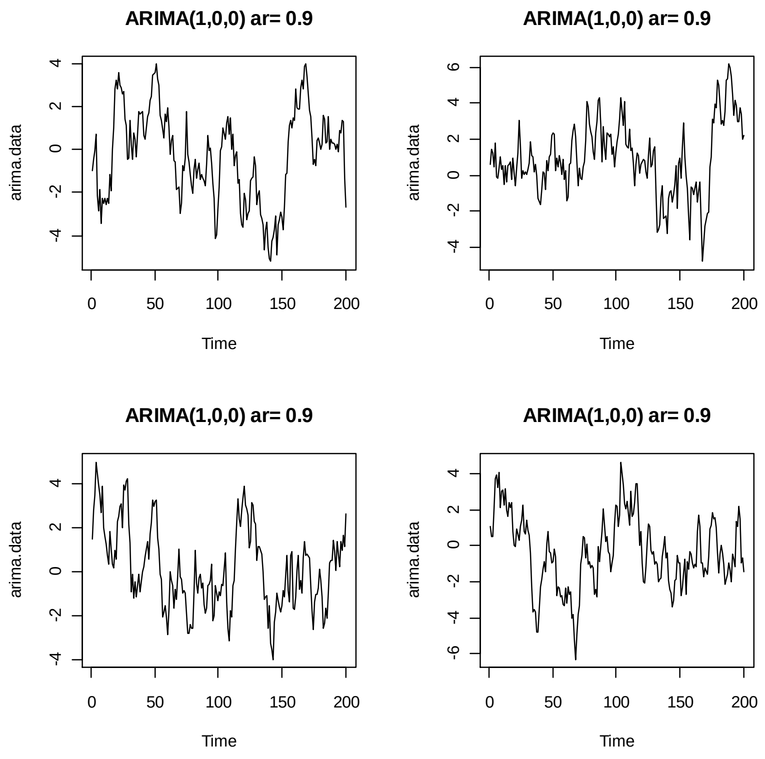
```
demo.arima.sim100 = function(ar.val = 0.9)
{
  par(mfrow=c(2,2))
  for (i in 1:4) {
    arima.data = arima.sim(list(order=c(1,0,0), ar=ar.val), 200)
    plot(arima.data, type="l", main=paste("ARIMA(1,0,0) ar=",
                                    as.character(ar.val)))
  }
  par(mfrow=c(1,1))
  return (TRUE)
}
```

The function has the parameter ar.val that passes the value for the AR coefficient. The default value is 0.9. Execute the following command to obtain a sample of four ARIMA plots. Remember that your plots will be different, each time you execute the next command:

```
> demo.arima.sim100()
```

Figure 8 shows 4 sample ARIMA(1,0,0) curves. Each time you execute the above command you get a different set. In fact each graph should be unique.

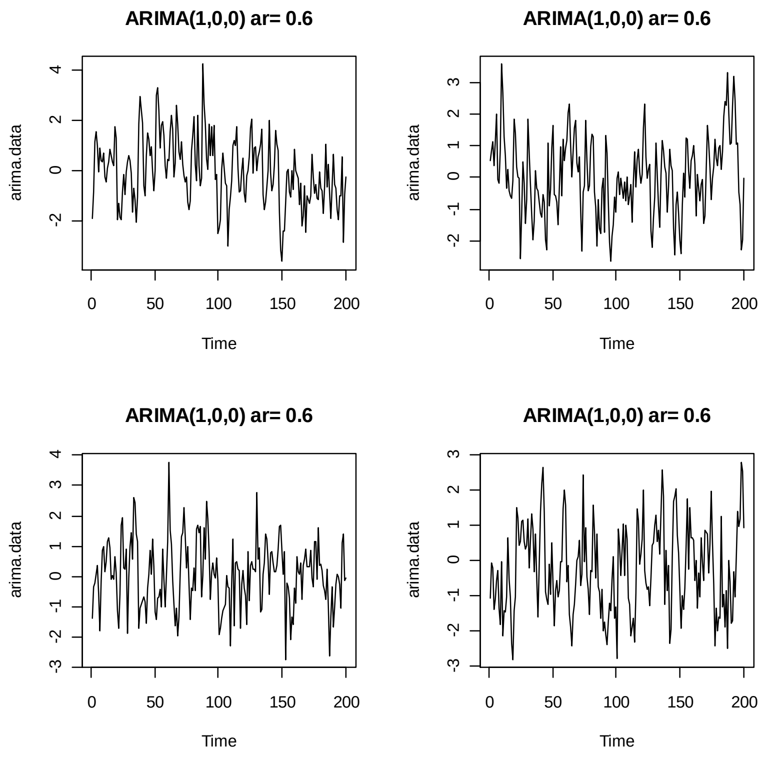
Figure 8. Four sample ARIMA(1,0,0) curves.



Execute the following command, supplying the value of 0.6 to the ar.val parameter:

```
> demo.arima.sim100(0.6)
```

Figure 9. ARIMA(1,0,0) graphs for AR=0.6.



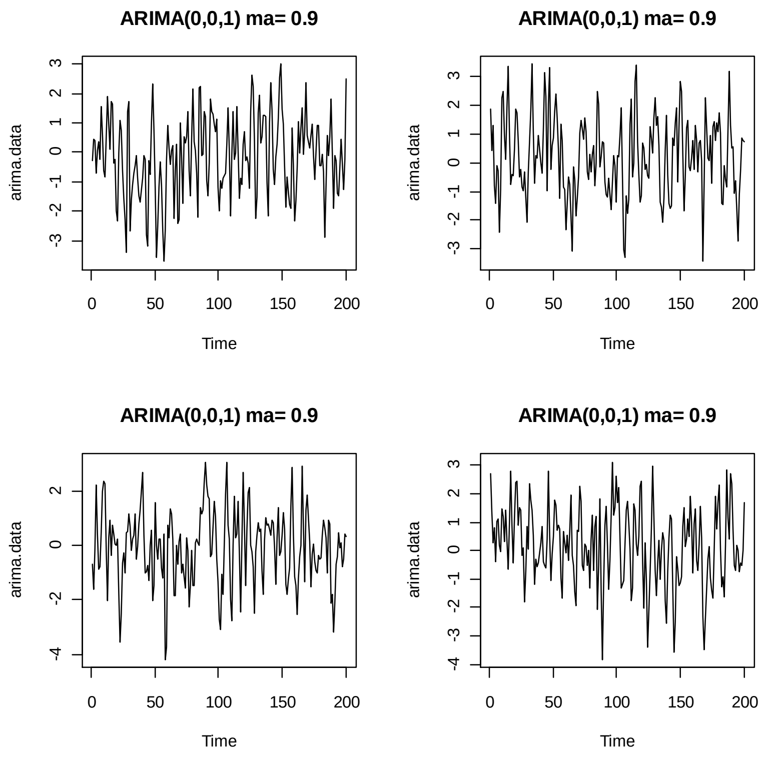
Here is the function demo.arima.sim001() which draws four ARIMA(0,0,1) graphs:

```
demo.arima.sim001 = function(ma.val = 0.9)
{
  par(mfrow=c(2,2))
  for (i in 1:4) {
    arima.data = arima.sim(list(order=c(0,0,1), ma=ma.val), 200)
    plot(arima.data, type="l", main=paste("ARIMA(1,0,0) ma=",
                                          as.character(ma.val)))
  }
  par(mfrow=c(1,1))
  return (TRUE)
}
```

Execute the following command to draw a set of four ARIMA(0,0,1) graphs shown in Figure 10:

```
> demo.arima.sim001()
```

Figure 10. Sample ARIMA(0,0,1) graphs.



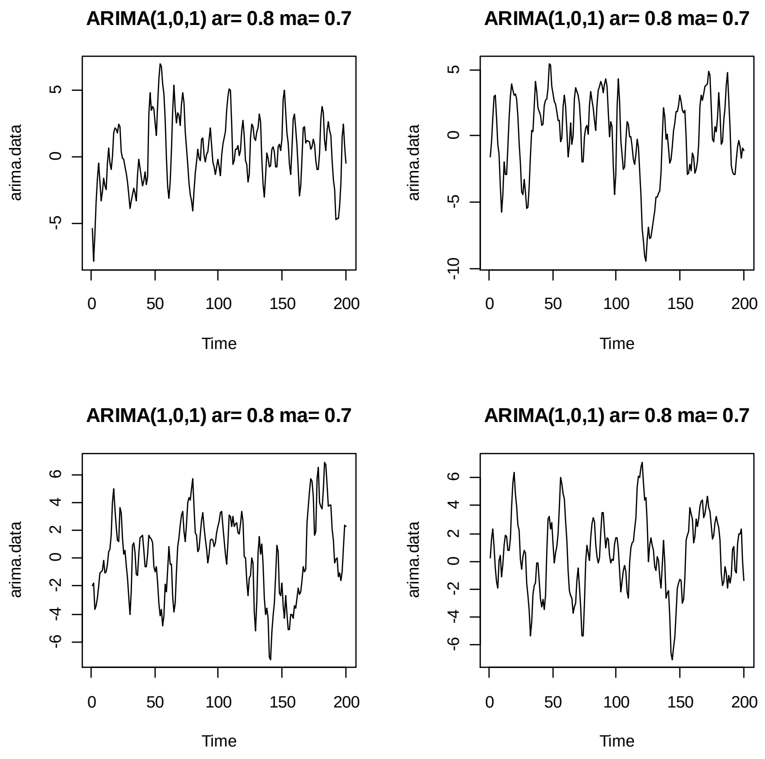
Here is the function demo.arima.sim101() which draws four ARIMA(1,0,1) graphs:

```
demo.arima.sim101 = function(ar.val = 0.9, ma.val = 0.9)
{
  par(mfrow=c(2,2))
  for (i in 1:4) {
    arima.data = arima.sim(list(order=c(1,0,1), ar=ar.val, ma=ma.val), 200)
    plot(arima.data, type="l", main=paste("ARIMA(1,0,1) ar=",
          as.character(ar.val), "ma=", as.character(ma.val)))
  }
  par(mfrow=c(1,1))
  return (TRUE)
}
```

Execute the following command to draw a set of four ARIMA(1,0,1) graphs shown in Figure 11:

```
> demo.arima.sim101(0.8,0.7)
```

Figure 11. Sample graphs for ARIMA(1,0,1).



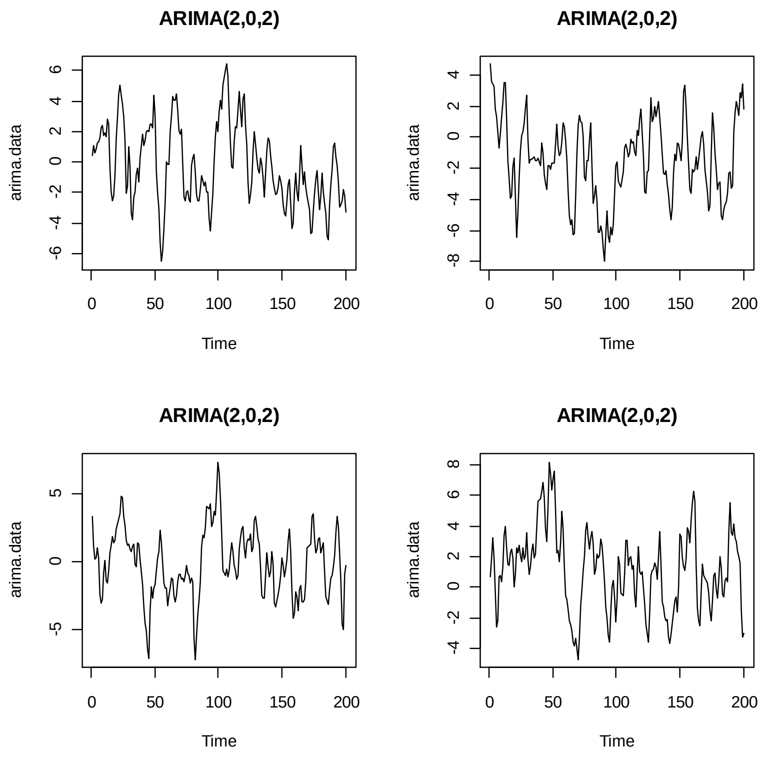
Finally, here is the function demo.arima.sim202() which draws four ARIMA(2,0,2) graphs:

```
demo.arima.sim202 = function(ar.val = c(0.9, -0.1), ma.val = c(0.7,-0.1))
{
  par(mfrow=c(2,2))
  for (i in 1:4) {
    arima.data = arima.sim(list(order=c(2,0,2), ar=ar.val, ma=ma.val), 200)
    plot(arima.data, type="l", main="ARIMA(2,0,2)")
  }
  par(mfrow=c(1,1))
  return (TRUE)
}
```

Execute the following command to draw a set of four ARIMA(2,0,2) graphs shown in Figure 12:

```
demo.arima.sim202(ar.val = c(0.9, -0.1), ma.val = c(0.75,-0.1))
```

Figure 12. Sample graphs for ARIMA(2,0,2).



Obtaining Stationary Data

Basic ARIMA calculations assume that the data is stationary—it has no trend or seasonal influences. While some time series are stationary, others are not. using differences in the time series values in one approach that has proven to be effective.

Using Linear Differences

The simplest differencing scheme is to apply the differences between subsequent observed time series values. If taking the differences of a time series fails to yield stationary values, you take the second difference, and so on.

R offers the diff() function to generate first, second, third differences, and so on. The declaration for the diff() function is:

```
diff(x, lag = 1, differences = 1, ...)
```

Where parameter **x** is the source data. The parameter **lag** is an integer that specifies the lag to use. The parameter **differences** specifies which level of differences to return.

Here is a sample function that demonstrates calculating and plotting the differences. The function plots a sample ARIMA(2,2,2) graph and shows its first three differences:

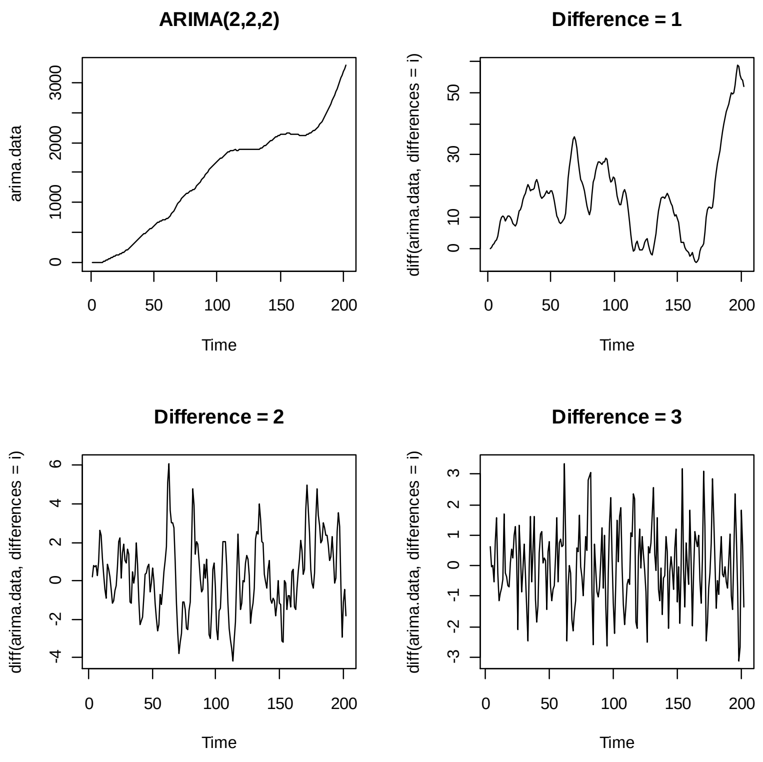
```
demo.arima.sim222.diff = function(ar.val = c(0.8, -0.1),
                                                ma.val = c(0.5,-0.1))
{
  par(mfrow=c(2,2))
  arima.data = arima.sim(list(order=c(2,2,2), ar=ar.val, ma=ma.val), 200)
  plot(arima.data, type="l", main="ARIMA(2,2,2)")
  for (i in 1:3)
    plot(diff(arima.data, differences=i), type="l",
       main=paste("Difference =", as.character(i)))
  par(mfrow=c(1,1))
  return (TRUE)
}
```

Execute the following command to use the above function and obtain the graphs for the ARIMA(2,2,2) and its three differences:

```
> demo.arima.sim222.diff()
```

Figure 13 shows the graphs generated by the above command. Notice that the ARIMA(2,2,2) has a clear trend. The first difference does not show stationary behavior. The graphs for the second and third differences do exhibit stationary behavior.

Figure 13. Graphs for ARIMA(2,2,2) model and its first three differences.



Here is another simulation function that creates graphs for an ARIMA(2,3,2) model and its first three differences:

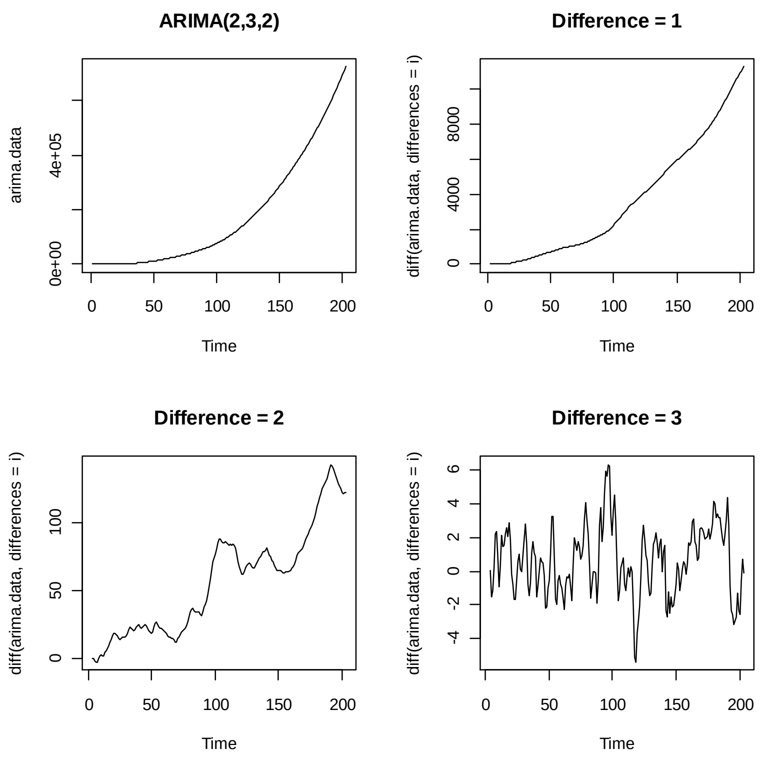
```
demo.arima.sim232.diff = function(ar.val = c(0.8, -0.1), ma.val = c(0.5,-
0.1))
{
  par(mfrow=c(2,2))
  arima.data = arima.sim(list(order=c(2,3,2), ar=ar.val, ma=ma.val), 200)
  plot(arima.data, type="l", main="ARIMA(2,3,2)")
  for (i in 1:3)
    plot(diff(arima.data, differences=i), type="l",
       main=paste("Difference =", as.character(i)))
  par(mfrow=c(1,1))
  return (TRUE)
}
```

Execute the following command to use the above function and obtain the graphs for the ARIMA(2,3,2) and its three differences:

```
> demo.arima.sim232.diff()
```

Figure 14 shows the graphs generated by the above command. Notice that the ARIMA(2,3,2) has clearly a smooth trend. The first two differences do not show stationary behavior. The graphs for the third difference exhibits stationary behavior.

Figure 14. Graphs for ARIMA(2,3,2) model and its first three differences.



Using Logarithmic Differences

Some analysts use the difference in the logarithmic values of time series. This kind of difference is also equal to the logarithm of the ratio between two values.

Here is the function demo.arima.sim111.log.diff() which simulates an ARIMA(1,1,1) model and draws graphs for the logarithmic differences:

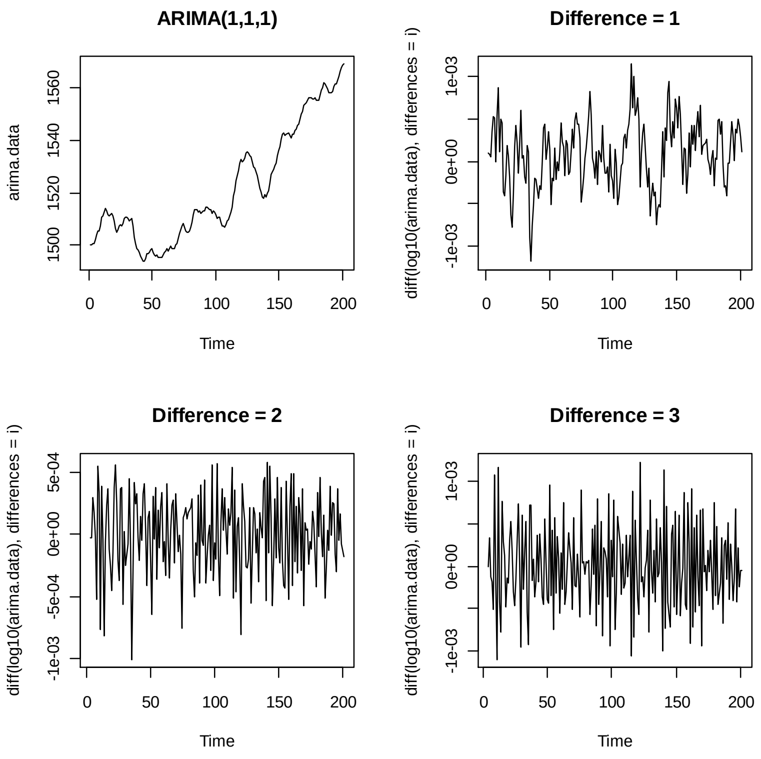
```
demo.arima.sim111.log.diff = function(ar.val = 0.8, ma.val = -0.2)
{
  par(mfrow=c(2,2))
  arima.data = 1500+arima.sim(list(order=c(1,1,1), ar=ar.val, ma=ma.val),
                              200)
  plot(arima.data, type="l", main="ARIMA(1,1,1)")
  for (i in 1:3)
    plot(diff(log10(arima.data), differences=i), type="l",
       main=paste("Difference =", as.character(i)))
  par(mfrow=c(1,1))
  return (TRUE)
}
```

Execute the following command to use the above function and draw graphs for the ARIMA(1,1,1) and the first three logarithmic differences.

```
> demo.arima.sim111.log.diff()
```

Figure 15 shows the graphs generated by the above command. The graphs for the second and third logarithmic differences show clear stationary behavior, while the graph for the first logarithmic difference shows a perhaps debatable stationary trend.

Figure 15. Graphs for ARIMA(1,1,1) model and three logarithmic differences.



Here is the function demo.arima.sim121.log.diff() which simulates an ARIMA(1,2,1) model and draws graphs for the logarithmic differences:

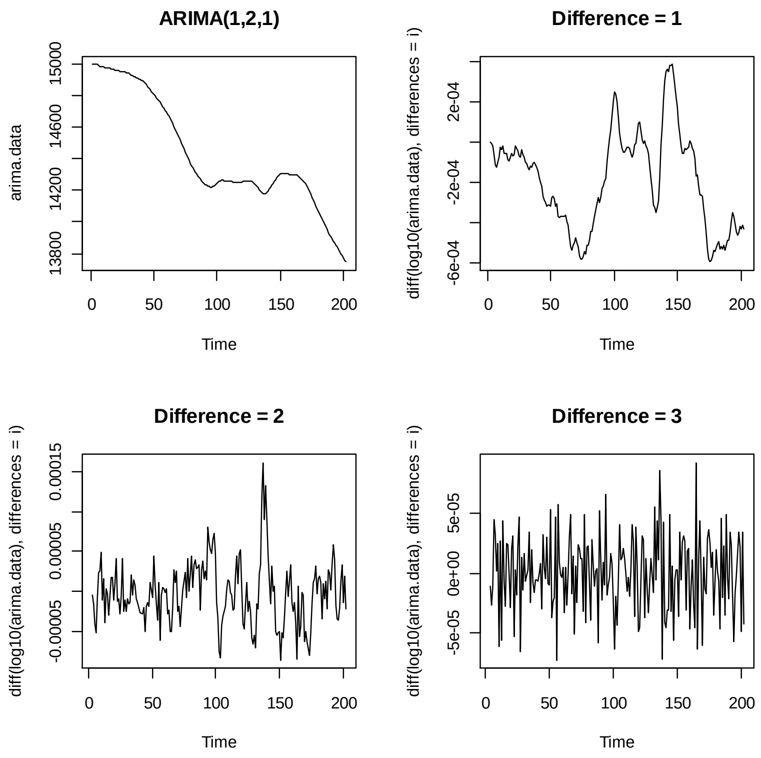
```
demo.arima.sim121.log.diff = function(ar.val = 0.8, ma.val = -0.2)
{
  par(mfrow=c(2,2))
  arima.data = 15000+arima.sim(list(order=c(1,2,1), ar=ar.val, ma=ma.val),
                               200)
  plot(arima.data, type="l", main="ARIMA(1,2,1)")
  for (i in 1:3)
  plot(diff(log10(arima.data), differences=i), type="l",
       main=paste("Difference =", as.character(i)))
  par(mfrow=c(1,1))
  return (TRUE)
}
```

Execute the following command to use the above function and draw graphs for the ARIMA(1,2,1) and the first three logarithmic differences.

```
> demo.arima.sim121.log.diff()
```

Figure 16 shows the graphs generated by the above command. The graphs for the third logarithmic difference show clear stationary behavior, while the graph for the second logarithmic difference shows a perhaps debatable stationary trend. The graph for the first logarithmic difference is not stationary.

Figure 16. Graphs for ARIMA(1,2,1) model and three logarithmic differences.



Using the ACF and PACF Curves

The ACF and PACF plots offer a good indication for the appropriate ARIMA model. The R functions acf() and pacf() plots the ACF and PACF graphs and also return the autocorrelation factors and partial autocorrelation factors.

The declarations for the acf() and pacf() functions are:

```
acf(x, lag.max = NULL,
    type = c("correlation", "covariance", "partial"),
    plot = TRUE, na.action = na.fail, demean = TRUE, ...)

pacf(x, lag.max = NULL, plot = TRUE, na.action = na.fail,    ...)
```

The parameter **x** is a numeric vector, numeric matrix, an acf object, or a time series object. The parameter **lag.max** specifies the maximum lag at which to calculate the acf. The default **value** is equal to 10*log10(N/m), where N is the number of observations and m the number of series. The function automatically limits to one less than the number of observations in the series. The parameter **type** is a string that specifies the type of acf to be computed. the function permits arguments to be "correlation" (which is the default), "covariance" or "partial". The Boolean parameter is logical. The function plots the acf data by default. The parameter **na.action** specifies which function to call in order to handle missing values. The Boolean parameter **demean** logical signals the removal of the mean from the data.

Here is a pair of functions that illustrate the acf() and pafc() functions:

```
demo.acf.sim101 = function(ar.val = 0.9, ma.val = -0.9)
{
  par(mfcol=c(2,2))
  arima.data = arima.sim(list(order=c(1,0,1), ar=ar.val, ma=ma.val), 200)
  acf(arima.data)
  pacf(arima.data)
  diff1 = diff(arima.data)
  acf(diff1)
  pacf(diff1)
  par(mfrow=c(1,1))
  return (TRUE)
}

demo.acf.sim111 = function(ar.val = 0.9, ma.val = -0.9)
{
  par(mfcol=c(2,2))
  arima.data = arima.sim(list(order=c(1,1,1), ar=ar.val, ma=ma.val), 200)
  acf(arima.data)
  pacf(arima.data)
  diff1 = diff(arima.data)
  acf(diff1)
  pacf(diff1)
  par(mfrow=c(1,1))
  return (TRUE)
}
```

Each function performs the following tasks:

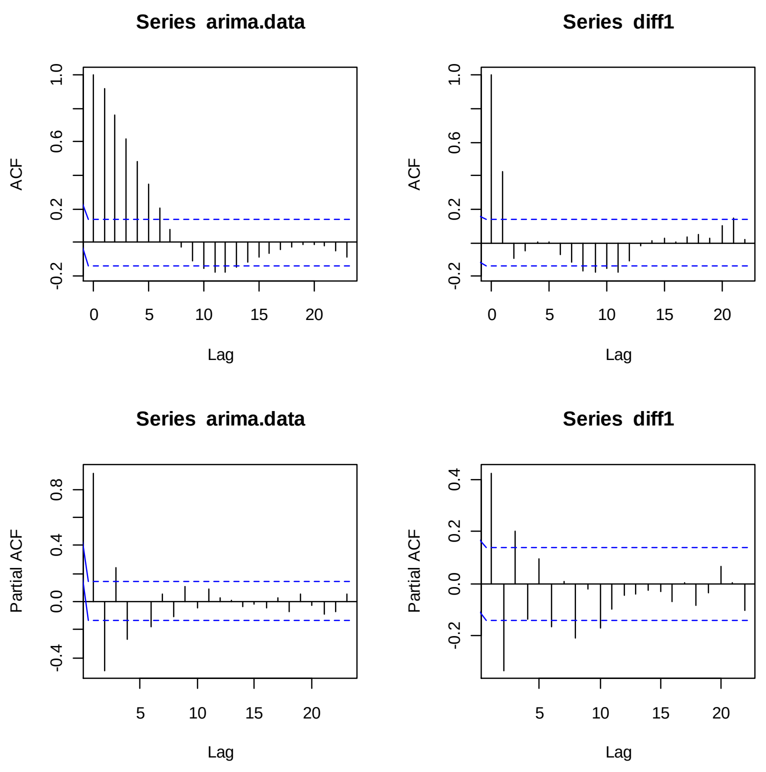
- Creates a simulated ARIMA time series of certain order.
- Calls function acf() to plot the ACF graph for the ARIMA model.
- Calls function pacf() to plot the PACF graph for the ARIMA model.
- Calculates the first difference in the time series and stores it in the variable diff1.
- Calls function acf() to plot the ACF graph for the first difference.
- Calls function pacf() to plot the PACF graph for the first difference.
- Returns TRUE as the function result.

Execute the following commands to call function demo.acf.sim101() and pass different values for the AR and MA coefficients:

```
> par(ask=TRUE)
> demo.acf.sim101(0.9,0.8)
> demo.acf.sim101(-0.9,0.8)
> demo.acf.sim101(0.9,-0.8)
> demo.acf.sim101(-0.9,-0.8)
> par(ask=FALSE)
```

Figure 17 shows the first graph that is generated by the call to function demo.acf.sim101(0.9,0.8). The above commands generate four graphs in all.

Figure 17. Sample graph showing acf and pacf graphs for ARIMA(1,0,1) time series and first differences for that series.

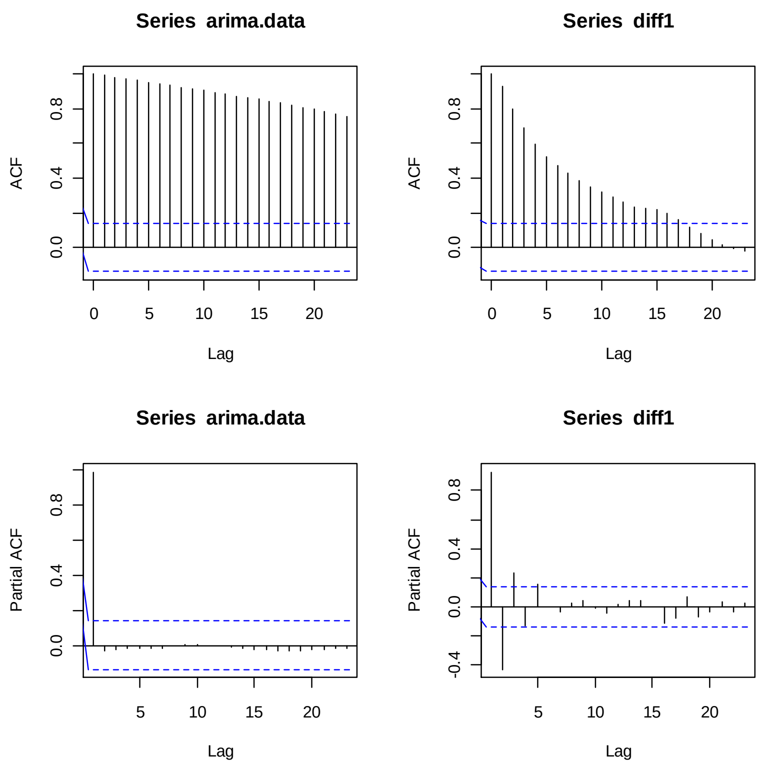


Execute the following commands to call function demo.acf.sim111() and pass different values for the AR and MA coefficients:

```
> par(ask=TRUE)
> demo.acf.sim111(0.9,0.8)
> demo.acf.sim111(-0.9,0.8)
> demo.acf.sim111(0.9,-0.8)
> demo.acf.sim111(-0.9,-0.8)
> par(ask=FALSE)
```

Figure 18 shows the first graph that is generated by the call to function demo.acf.sim111(0.9,0.8). The above commands generate four graphs in all.

Figure 18. . Sample graph showing acf and pacf graphs for ARIMA(1,1,1) time series and first differences for that series.



Manual ARIMA Model Selection

The function arima() performs the calculations on a time series to obtain the ARIMA coefficients. The declaration for this important function is:

```
arima(x, order = c(0, 0, 0),
      seasonal = list(order = c(0, 0, 0), period = NA),
      xreg = NULL, include.mean = TRUE,
      transform.pars = TRUE,
      fixed = NULL, init = NULL,
      method = c("CSS-ML", "ML", "CSS"),
      n.cond, optim.method = "BFGS",
      optim.control = list(), kappa = 1e6)
```

Where parameter **x** is univariate time series. The parameter **order** is the order in the ARIMA(p,d,q) model. The parameter **seasonal** defines the seasonal part of the ARIMA model and also the period (which defaults to frequency(x)). The parameter **xreg** is a vector or matrix of external regressors, which must have the same number of rows as parameter x. The Boolean parameter **include.mean** tells the function if the ARMA model should include a mean/intercept term. The default parameter value is TRUE for un-differenced time series. The function ignores the value for this parameter for ARIMA models with differencing. The Boolean parameter **transform.pars** tells the function, when TRUE, to transform the AR parameters to ensure that they remain in the region of stationarity. The function ignores this parameter when the parameter, method is "CSS". The parameter **fixed** is a numerical vector of the same length as the total number of parameters. If supplied, only NA entries in fixed will be varied. transform.pars = TRUE will be overridden (with a warning) if any AR parameters are fixed. It may be wise to set transform.pars = FALSE when fixing MA parameters, especially near non-invertibility. The parameter **init** is a numeric vector containing the initial parameter values. The function fills missing values with zeroes except for the regression coefficients. The function ignores the values already specified in parameter fixed. The parameter **method** specifies the fitting method, which can be the maximum likelihood or minimize conditional sum-of-squares. By default (unless there are missing values) the function employs the conditional-sum-of-squares to find starting values, and then switches to using the maximum likelihood. The parameter **n.cond** should be only used when fitting by the conditional-sum-of-squares method. This parameter specifies the number of initial observations to ignore. The function ignores this parameter if less than the maximum lag of an AR term. The parameter **optim.method** specifies the value passed as the method argument to function optim(). The default value is "BFGS" which refers to the best optimization method. The parameter **optim.control** provides the list of control parameters for function optim(). The parameter **kappa** is the prior variance (as a multiple of the innovations variance) for the past observations in a differenced model.

Despite the long list of parameters in function arima(), most calls to this function often involve the first two parameters.

To illustrate using function arima() with some of the other functions, here the the function arima.test():

```
arima.test = function()
{
  arima.data = 100 + arima.sim(n = 200, list(order=c(1,1,1),
               ar = 0.1, ma = 0.7))
  diff1 = diff(arima.data, differences=1)
  diff2 = diff(arima.data, differences=2)
  par(ask=TRUE)
  par(mfcol=c(3,1))
  plot(arima.data)
  plot(diff1)
  plot(diff2)
  par(mfcol=c(2,3))
  acf(arima.data)
  pacf(arima.data)
  diff1 = diff(arima.data, differences=1)
  acf(diff1)
  pacf(diff1)
  acf(diff2)
  pacf(diff2)
  par(mfrow=c(1,1))
  par(ask=FALSE)
  arima.data.res011 = arima(arima.data, order=c(0,1,1))
  arima.data.res110 = arima(arima.data, order=c(1,1,0))
  arima.data.res111 = arima(arima.data, order=c(1,1,1))
  arima.data.res012 = arima(arima.data, order=c(0,2,1))
  arima.data.res210 = arima(arima.data, order=c(1,2,0))
  arima.data.res212 = arima(arima.data, order=c(1,2,1))
  show(arima.data.res011)
  show(arima.data.res110)
  show(arima.data.res111)
  show(arima.data.res012)
  show(arima.data.res210)
  show(arima.data.res212)
  return (TRUE)
}
```

The above function performs the following tasks:

- Creates a simulated ARIMA(1,1,1) time series.
- Calculates the first and second differences for the time series.
- Plot the time series and its two differences.
- Plot the acf and pacf for the time series and its two differences.
- Calculate the coefficients for models ARIMA(0,1,1), ARIMA(1,1,0), ARIMA(1,1,1), ARIMA(1,2,0), ARIMA(0,2,1), and ARIMA(1,2,1). The choice for selecting this limited set of models is based on knowing what the simulated ARIMA model is and viewing several sets of the graphs for the time series, their differences, and their acf and pacf plots.
- Display the results for the coefficients for the above ARIMA models.
- Returns TRUE.

To take the function arima.test() for a spin, execute the following command:

```
> arima.test()
Waiting to confirm page change...
Waiting to confirm page change...

Call:
arima(x = arima.data, order = c(0, 1, 1))

Coefficients:
         ma1
      0.8407
s.e.  0.0364

sigma^2 estimated as 1.130:  log likelihood = -296.61,  aic = 597.21

Call:
arima(x = arima.data, order = c(1, 1, 0))

Coefficients:
         ar1
      0.6428
s.e.  0.0549

sigma^2 estimated as 1.317:  log likelihood = -311.6,  aic = 627.21
```

```
Call:
arima(x = arima.data, order = c(1, 1, 1))

Coefficients:
         ar1     ma1
      0.2366  0.7563
s.e.  0.0851  0.0616

sigma^2 estimated as 1.088:  log likelihood = -292.88,  aic = 591.75

Call:
arima(x = arima.data, order = c(0, 2, 1))

Coefficients:
         ma1
      0.1718
s.e.  0.1517

sigma^2 estimated as 1.583:  log likelihood = -328.11,  aic = 660.21

Call:
arima(x = arima.data, order = c(1, 2, 0))

Coefficients:
         ar1
      0.0413
s.e.  0.0707

sigma^2 estimated as 1.592:  log likelihood = -328.62,  aic = 661.24

Call:
arima(x = arima.data, order = c(1, 2, 1))

Coefficients:
          ar1     ma1
      -0.5713  0.8849
s.e.   0.0950  0.0559

sigma^2 estimated as 1.389:  log likelihood = -315.31,  aic = 636.62
[1] TRUE
```

The function displays the graphs in Figure 19 and Figure 20. Figure 19 displays the graphs for the time series and its first two differences. Figure 20 displays the acf and pacf for the time series and its first two differences.

Examine the above output and look for the results with the least AIC value. The model ARIMA(1,1,1) has the least value of 591.75. The calculations estimated the AR and MA coefficients to be 0.2366 and 0.7563, respectively. The values are kind of close to the those of 0.1 and 0.7 used in calling function arima.sim(). The value of sigma^2 for the ARIMA(1,1,1) is 1.088 and is the least one in among the set of results. Keep in mind that when you rerun such calculations you may get results that favor different ARIMA models.

Figure 19. The plot for a time series and its first two differences.

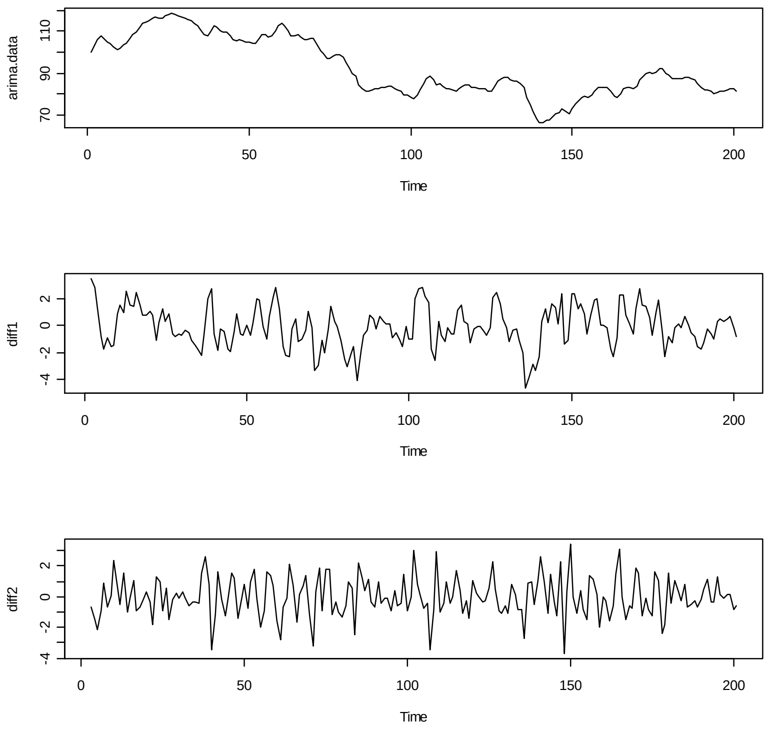
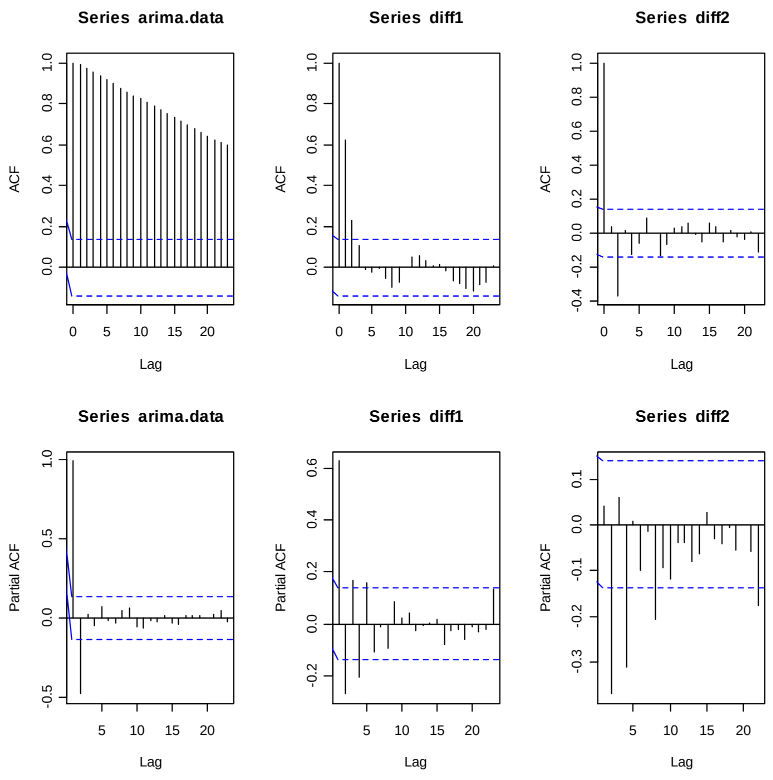


Figure 20. The acf and pacf plot for the time series and its first two differences.



Examining the Results and Residuals

The function tsdiag() performs graphical diagnostics on an ARIMA model.

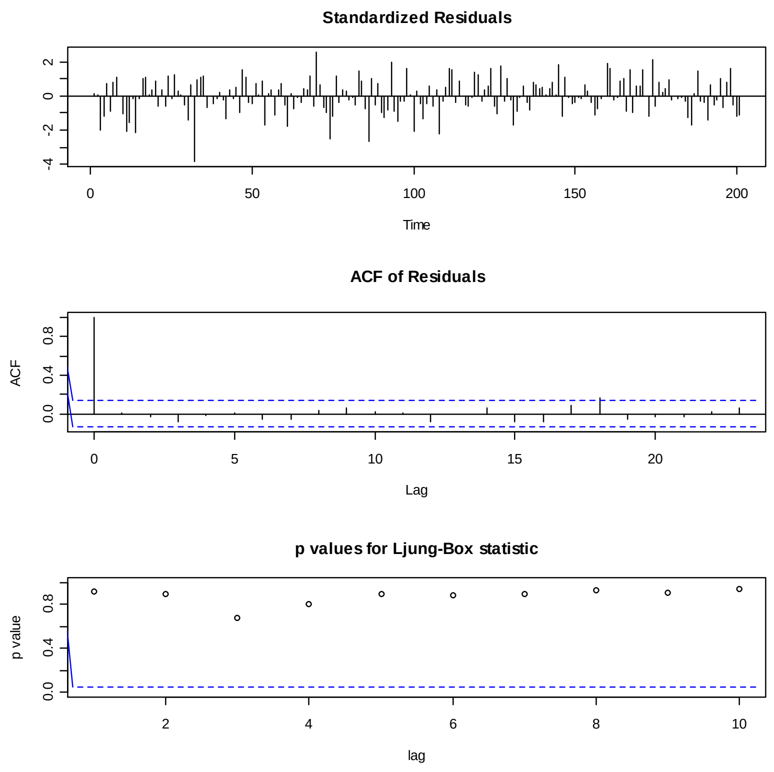
Execute the following commands to test the tsdiag() function:

```
> arima.data = 100 + arima.sim(n = 200, list(order=c(1,1,1), ar = 0.1, ma =
0.7))
> (arima.data.res111 = arima(arima.data, order=c(1,1,1)))

Call:
arima(x = arima.data, order = c(1, 1, 1))

Coefficients:
         ar1     ma1
      0.1211  0.6745
s.e.  0.0950  0.0703

sigma^2 estimated as 1.03:  log likelihood = -287.13,  aic = 580.27
> tsdiag(arima.data.res111)
```

Figure 21 shows the graph created by the tsdiag() function. The graphs include the standardized residuals, ACF for residuals, and the p values for the Ljung-Box statistic. The latte statistics tests the null hypothesis of the independently distributed residuals. High p values indicate that the residuals are distributed independently.

All of these graphs assure us that the ARIMA(1,1,1) is a good model (and why shouldn't it be?).

Figure 21. The graphs generated by function tsdiag().



ARIMA Forecasting

The function predict(arima.res, n.ahead) allows you to predict n.ahead time units for the result of function arima().

The following commands allow you to calculate the predicted values and their confidence intervals for 20 days in the future of the ARIMA(1,1,1) model in variable :

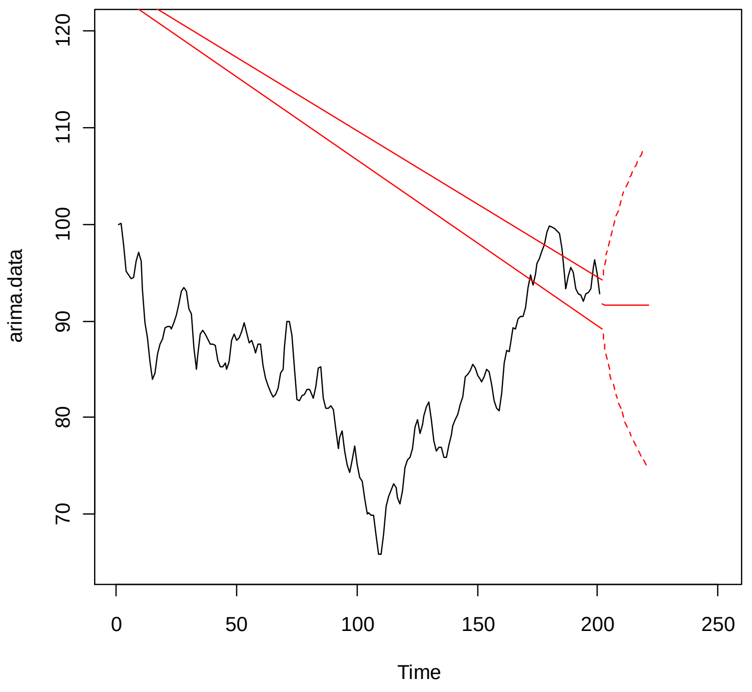
```
> plot(arima.data, type="l", xlim=c(1,250), ylim=c(65,120))
> fut=predict(arima.data.res111, 20)
> lines(fut$pred, col="red")
> lines(fut$pred + 2*fut$se, col="red", lty=2)
> lines(fut$pred - 2*fut$se, col="red", lty=2)
```

The above command plot the time series using function plot(). This function call uses the parameter xlim and ylim to expand on the ranges for the x and y values to accommodate the predictions. The command set also calls function predict() and passes the arguments arima.data.res111 and 20 to that function. The command set then calls three times the function lines() to plot the predicted value in variable fut and its confidence interval. The code uses the expressions fut$pred + 2*fut$se and fut$pred - 2*fut$se to calculate the confidence interval. The number 2 is an approximation for the inverse Student-t distribution. A slightly more accurate version of the above command set is:

```
> plot(arima.data, type="l", xlim=c(1,250), ylim=c(65,120))
> fut=predict(arima.data.res111, 20)
> lines(fut$pred, col="red")
> lines(fut$pred + qt(0.025,200-3)*fut$se, col="red", lty=2)
> lines(fut$pred - qt(0.025,200-3)*fut$se, col="red", lty=2)
```

The above set of commands uses the function qt() to calculate the inverse Student-t distribution. Either set of commands generates the graph in Figure 22.

Figure 22. Predicted values and associated confidence interval for a time series.



Best ARIMA Model Search

This subsection presents an function that performs a best-ARIMA model search. The function displays graphs, writes the same graphs to a pdf file, and writes the results to both the R Console window and a text file. The function selects the best ARIMA model using the corrected AIC statistic.

The Function best.arima.aicc()

Here is the source code for the function best.arima.aicc():

```
best.arima.aicc = function(x, memo="", max.ar=4, max.ma=4, max.diff=3,
                    show.graphs=TRUE, to.pdf=TRUE,
                    out.file.basename="best.arima",
                    target.dir="", trace=FALSE, warning.messages=FALSE)
{
  # Find best ARIMA model

  # x is the time series
  # memo is a short memory that is written to the output txt file
  # max.ar is the highest p value in ARIMA(p,d,q)
  # max.ma is the highest q value in ARIMA(p,d,q)
  # max.diff is the highest f value in ARIMA(p,d,q)
  # show.graphs is a flag to display graphs
  # to.pdf is a flag to write graphs to a pdf file
  # out.file.basename is the base name for the output pdf file
  # target.dir is the target directory for output pdf file. The default
  #   indicates that the current working directory should be used.
  # trace is a flag to display intermediate values for AICC, q, d, and p
  # warning,messages is a flag to display warning messages

  n=length(x) # size of sample
  # select ARIMA(1,0,0) model arbitrarily
  if (warning.messages) {
    m = try(arima(x, order=c(1,0,0), method="ML"), silent=TRUE)
  }
  else {
    m = suppressWarnings(try(arima(x, order=c(1,0,0), method="ML"),
      silent=TRUE))
  }
  k.m = length(m$coef) # number of parameters in arima model
  s.m= m$sigma2        # mle of sigma^2
  best.arima.aicc = m$aic + 2 * k.m * (k.m + 1) / (n - k.m - 1)
  best.diff=1
  best.arima=m
  best.arima.name = "ARIMA(1,0,0)"
  num.res = 4
  row.vals = rep(0, num.res)
  max.elems = (max.ar + 1) * (max.ma + 1) * (max.diff + 1)
  aicc.mat = matrix(rep(0, num.res * max.elems),
                 nrow = max.elems, ncol=num.res)
  # output counter
  count = 0
  for (p in 0:max.ar) {
    for (q in 0:max.ma) {
      for (d in 0:max.diff) {
        # trigger artificial error
        try(sum(dummy), silent=TRUE)
        # store the error message for the above error
        last.err.msg = geterrmessage()
        # get the arima model in a try() function
        if (warning.messages) {
          m = try(arima(x, order=c(p,d,q), method="ML"), silent=TRUE)
        }
        else {
          m = suppressWarnings(try(arima(x, order=c(p,d,q), method="ML"),
             silent=TRUE))
        }
        # arima() function did not generate an error?
        if (last.err.msg == geterrmessage()) {
          k.m = length(m$coef) # number of parameters in m model
          s.m= m$sigma2        # mle of sigma^2
          aicc = m$aic + 2 * k.m * (k.m + 1) / (n - k.m - 1)
          row.vals = c(aicc, p, d, q)
          if (trace) cat(count, row.vals, "\n")
          if (length(row.vals) == num.res) {
            count = count + 1 # increment solution counter
            # store results in a row of matrix aicc.mat
            aicc.mat[count,] = row.vals
            # found a better ARIMA model?
            if (best.arima.aicc > aicc) {
              # store results for the better ARIMA model
              best.diff = d
              best.arima = m
              best.arima.name = paste("ARIMA(", as.character(p), ",",
                 as.character(d), ",", as.character(q), ")", sep="")
              best.arima.aicc = aicc
            }
          }
        }
      }
    }
  }

  # sort the matrix aicc.mat
  # get the rank vector of column 1
  sort.order = order(aicc.mat[,1])
  # sort the matrix columns
  for (i in 1:ncol(aicc.mat))
    aicc.mat[,i] = aicc.mat[sort.order,i]

  # remove zeros from sorted matrix
  nr = nrow(aicc.mat)
  repeat {
    # first row has a zero aicc?
    if (aicc.mat[1,1] == 0) {
      # delete the row
      aicc.mat = aicc.mat[-1,]
      nr = nr - 1 # decrement row counter
      # just one row left?
      if (nr < 2) break
    }
    else  # exit if aicc.mat[1,1] != 0
      break
  }


  # List of AIC and values of (p,d,q) (for  double checking with the optimum)
  list.res = list(aicc.mat = aicc.mat,
                 best.arima.name = best.arima.name,
                 best.arima = best.arima,
                 best.arima.aicc = best.arima.aicc)
  max.lags = 25
  # display graphs?
  if (show.graphs) {
    par(ask=TRUE)
    par(mfrow=c(1,1))
    plot(x, type="l", main="Time Series")
    par(mfrow=c(2,1))
    # difference for best arima is positive?
    if (best.diff > 0) {
      # show acf and pacf of differences
      acf(diff(x, differences=best.diff), max.lags)
      pacf(diff(x,differences=best.diff), max.lags)
    }
    else {
      # show acf and pacf of un-differenced data
      acf(x, 25)
      pacf(x, 25)
    }
    par(mfrow=c(1,1))
    tsdiag(best.arima)
    par(ask=TRUE)
  }

  # now send output to pdf file
  if (to.pdf) {
    # get the target directory
    if (target.dir == "") target.dir = paste(getwd(), "/", sep="")
    # get the system time
    st = Sys.time()
    # repalce colons with underscores
    st = gsub(":", "_", st)
    # assemble pdf filename
    pdf.file = paste(target.dir, out.file.basename, " ", st, ".pdf", sep="")
    # start dumping graphs to pdf file
    pdf(pdf.file)
    par(mfrow=c(1,1))
    plot(x, type="l", main="Time Series")
    par(mfrow=c(2,1))
    # difference for best arima is positive?
    if (best.diff > 0) {
      # show acf and pacf of differences
      acf(diff(x, differences=best.diff), max.lags)
      pacf(diff(x, differences=best.diff), max.lags)
    }
    else {
      # show acf and pacf of un-differenced data
      acf(x, 25)
      pacf(x, 25)
    }
    par(mfrow=c(1,1))
    tsdiag(best.arima)
    dev.off()
  }

  # assemble txt filename
  txt.file = paste(target.dir, out.file.basename, " ", st, ".txt", sep="")
  # dump output using sink() function
  sink(txt.file)  # start sending output
  show(paste("Date/Time: ", Sys.time(), sep=""))
  if (memo != "") show(paste("Memo: ", memo, sep=""))
  show(list.res) # display list of results
  sink() # stop sending output to text file

  # and finally return the function's result
  return (list.res)
}
```

The Function's Tasks

The parameter **x** is the time series. The parameter **memo** is a short memo that is written to the output txt file. The parameter **max.ar** is the highest p value in ARIMA(p,d,q). The parameter **max.ma** is the highest q value in ARIMA(p,d,q). The parameter **max.diff** is the highest f value in ARIMA(p,d,q). The Boolean parameter **show.graphs** is a flag to display graphs. The Boolean parameter **to.pdf** is a flag to write graphs to a pdf file. The parameter **out.file.basename** specifies the base name for the output pdf file (no file extension should be included). The parameter **target.dir** is the target directory for output pdf file. The default value indicates that the current working directory should be used. The Boolean parameter **trace** is a flag to display intermediate values for AICC, q, d, and p. The Boolean **warning,messages** is a flag to display warning messages. The default value of FALSE makes the function run as quiet as possible!

The function best.arima.aicc() performs the following tasks:

1. Stores in the variable n the number of data points in the time series parameter.
2. Performs an initial ARIMA calculation for model ARIMA(1,0,0). The choice for this model is arbitrary. This task uses the parameter warning,messages to determine how to call the arima() function. When the parameter warning,messages is TRUE, the function does not use the function suppressWarnings() to suppress warning messages, and vice versa. In either case, the call to function arima() is an argument to function try(). This function traps the error and with its parameter silent set to TRUE, the runtime system displays no error messages.
3. Stores the some results from the last call to function arima().
4. Initializes the variables that track the best ARIMA model.
5. Initializes the variables that track the list of ARIMA(p, d, q) models' AICC, p, d, and q values.
6. Initializes the variable count which stores the number of results.
7. Starts three nested loops that iterate over the ranges of ARIMA model parameters p, d, and q. The nested loops perform the following subtasks:
   7.1. Trigger an artificial error and store the message for that error in variable last.err.msg.
   7.2. Invoke the arima() function to calculate the coefficients for the ARIMA(p,d,q) model. The function passes the call to arima() as an argument to function try() to catch any runtime error. In addition, the code may pass the result of function try() as an argument to function suppressWarnings(), when the parameter warning.messages is TRUE.

7.3. Determine if the call to function arima() generates an error. This task compares the string in variable last.err.msg with the result of calling function geterrmessage(). If there is an error, the program flow skips to the end of the inner most for loop.
7.4. Increment the result counter.
7.5. Store the values for AICC, p, d, and q in the matrix aicc.mat.
7.6. Determine if the AICC value for the current model is smaller than the AICC value of the best model. If this condition is true, the function has found a better ARIMA model. Consequently, the function updates the variables that store the information for the best ARIMA model.

8. Sorts the columns of matrix aicc.mat using the values of the first column as the keys for sorting each column.
9. Removes entries in matrix aicc.mat that have AICC values (in column 1) equal to zero.
10. Assembles the results in the list list.res.
11. Determines if the user requested to display graphs by checking the value of parameter show.graphs. If this parameter is TRUE, the function displays the time series, the acf and pacf graphs for the best ARIMA model, and the time series diagnostic graphs obtained by calling function tsdiag().
12. Determines if the user requested to write the graphs to a pdf file by checking the value of parameter to.pfd. If this parameter is TRUE, the function writes the same graphs in step 11 to a pdf file. The name of the output pdf file is a combination of the specified base name and the current date/time stamp. If the target.dir parameter is an empty string, the function uses the current working directory as the location to write the output pdf file.
13. Writes the results in the list list.res to a text file. This task uses the sink() function to redirect the output to an text file. The name of the text file is a combination of the specified base name and the current date/time stamp. Once the function is done writing the selected output to the text file, the function then calls function sink() with no parameters to resume normal output the R Console window.
14. Returns the function's result which is stored in variable list.res.

Sample Run

Here is a sample run with the function best.arima.aicc():

```
arima.data = 200 + arima.sim(n = 500, list(order=c(2,1,1),
                 ar = c(0.95, -0.1), ma = -0.1))
(baa = best.arima.aicc(arima.data))
```

The R Console displays the following data (which are also echoed in the output text file "**best.arima <*date*> <*time*>.txt**"):

```
[1] "Date/Time: 2009-10-22 10:33:31"
[1] "Memo: Test for simulated ARIMA(2,1,1) model"
$aicc.mat
          [,1] [,2] [,3] [,4]
 [1,] 1408.131    4    1    2
 [2,] 1408.409    1    1    0
 [3,] 1409.768    2    1    0
 [4,] 1409.800    1    1    1
 [5,] 1410.319    4    2    3
 [6,] 1410.423    3    2    4
 [7,] 1410.597    1    2    1
 [8,] 1411.361    2    1    1
 [9,] 1411.571    3    1    0
[10,] 1411.596    3    1    2
[11,] 1411.604    1    1    2
[12,] 1411.921    2    2    1
[13,] 1411.954    1    2    2
[14,] 1413.068    3    2    3
[15,] 1413.130    4    1    3
[16,] 1413.161    4    1    4
[17,] 1413.372    2    1    2
[18,] 1413.374    3    1    1
[19,] 1413.536    2    2    2
[20,] 1413.589    4    1    0
[21,] 1413.635    1    1    3
[22,] 1413.755    3    2    1
[23,] 1413.787    1    2    3
[24,] 1413.793    3    1    4
[25,] 1413.799    1    1    4
[26,] 1415.345    2    1    3
[27,] 1415.353    4    1    1
[28,] 1415.563    2    2    3
[29,] 1415.564    3    2    2
[30,] 1415.699    2    1    4
[31,] 1415.776    4    2    1
[32,] 1415.823    1    2    4
[33,] 1416.154    4    2    4
[34,] 1416.725    3    1    3
[35,] 1417.540    4    2    2
[36,] 1417.875    2    2    4
[37,] 1422.698    3    0    0
[38,] 1423.376    3    0    1
[39,] 1423.502    1    3    2
[40,] 1424.230    4    0    2
[41,] 1424.570    4    0    0
[42,] 1425.515    2    3    2
[43,] 1426.033    3    0    2
[44,] 1426.378    4    0    1
[45,] 1426.695    3    3    4
[46,] 1426.807    2    3    3
[47,] 1427.209    1    3    4
[48,] 1427.218    3    3    2
[49,] 1427.437    4    3    4
[50,] 1427.473    3    3    3
[51,] 1428.565    2    3    4
[52,] 1430.013    1    3    3
[53,] 1430.336    4    3    3
[54,] 1434.438    3    0    3
[55,] 1435.532    0    2    3
[56,] 1436.784    0    2    4
[57,] 1437.041    0    2    1
```

```
[58,] 1437.419    0    2    2
[59,] 1437.866    1    2    0
[60,] 1437.896    3    2    0
[61,] 1438.532    2    2    0
[62,] 1439.849    4    2    0
[63,] 1442.492    0    3    4
[64,] 1442.719    0    2    0
[65,] 1443.568    0    3    2
[66,] 1444.084    0    3    3
[67,] 1444.338    1    3    1
[68,] 1444.605    3    3    1
[69,] 1445.102    2    3    1
[70,] 1446.587    4    3    1
[71,] 1448.687    4    3    2
[72,] 1448.856    4    0    4
[73,] 1448.974    0    3    1
[74,] 1452.553    0    1    4
[75,] 1497.707    0    1    3
[76,] 1501.109    1    0    4
[77,] 1505.756    2    0    4
[78,] 1514.101    1    0    3
[79,] 1537.641    0    1    2
[80,] 1551.387    3    0    4
[81,] 1559.013    4    3    0
[82,] 1572.587    3    3    0
[83,] 1616.669    2    3    0
[84,] 1625.738    2    0    3
[85,] 1626.474    1    0    2
[86,] 1674.907    1    3    0
[87,] 1686.346    0    1    1
[88,] 1700.822    1    0    1
[89,] 1840.550    0    3    0
[90,] 2007.385    0    1    0
[91,] 2021.906    1    0    0
[92,] 2634.645    2    0    0
[93,] 3023.640    0    0    4
[94,] 3409.204    0    0    3
[95,] 3920.416    0    0    2
[96,] 4543.627    0    0    1
[97,] 5219.272    0    0    0

$best.arima.name
[1] "ARIMA(4,1,2)"

$best.arima

Call:
arima(x = x, order = c(p, d, q), method = "ML")

Coefficients:
          ar1     ar2     ar3     ar4     ma1     ma2
      -0.4589  0.0901  0.8159  0.0591  1.2872  1.0000
s.e.   0.0451  0.0354  0.0342  0.0456  0.0092  0.0092

sigma^2 estimated as 0.9408:  log likelihood = -696.98,  aic = 1407.96
```

```
$best.arima.aicc
[1] 1408.131
```

Notice that the function best.arima.aicc() selects the model ARIMA(4,1,2) as the best. The sorted list of ARIMA models shows that the model ARIMA(2,1,1) ranks as number 8. If you rerun the above commands you will most likely see a different ARIMA model selected the best. Figure 23 shows the time series for the simulated ARIMA. Figure 24 shows the ACF and PACF for the first difference of the best ARIMA mode. Figure 25 shows the diagnostic graphs for the best ARIMA model. The pdf file "**best.arima <*date*> <*time*>.pdf"** also contains these graphs.

Figure 23. Time series plot.

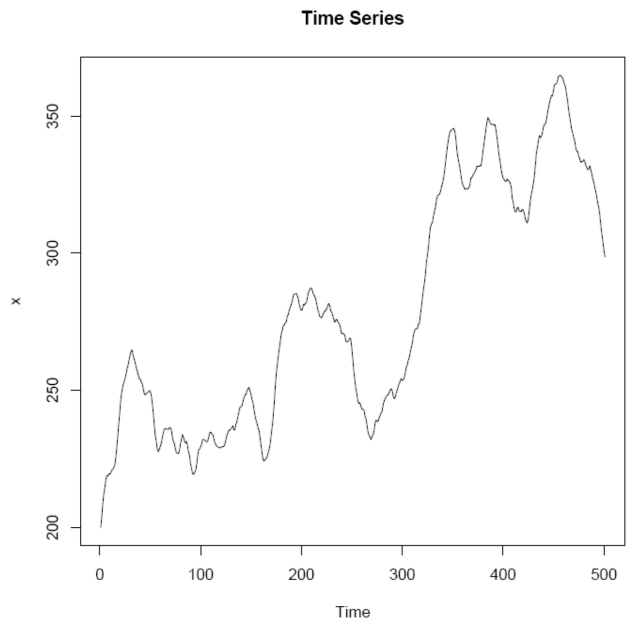


Figure 24. ACF and PACF plots.

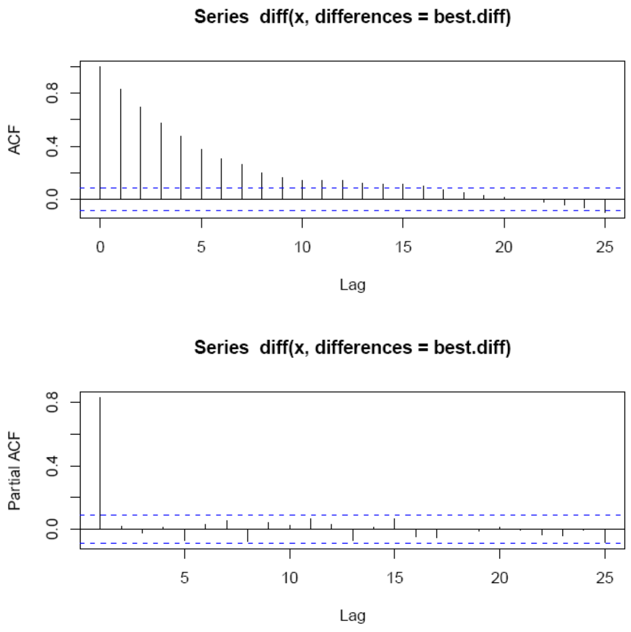
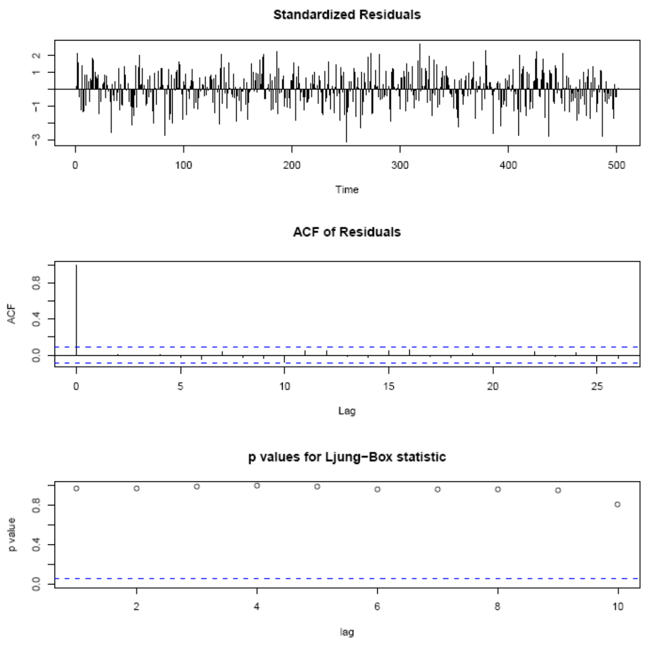


Figure 25. Time series diagnostic graphs.



I encourage you to try different commands to calculate the best ARIMA model by altering the following input:

- Changing the ARIMA model used in the function arima.sim().

- Changing the coefficients for the AR and MA terms.
- Changing the number of observations.
- Changing the range of p, d, and q by assigning different upper limits to these values through parameters max.p, max.diff, and max.q.

I was tempted to include a study based on changing the above variable to the simulated ARIMA model used in hunting for the best ARIMA model. I came to realize that the output from such model variations would easily double the size of this tutorial. In as much as the idea of experimenting with the different ARIMA values tempted my curiosity, I decided to give you the tool of function best.arima.aicc() and step aside.