Time Series Analysis in R

Boni

9/4/2020

Load Nile dataset

```r
data("Nile")
```

Explore Nile

```r
# Print the Nile dataset
print(Nile)
```

```
## Time Series:
## Start = 1871
## End = 1970
## Frequency = 1
##   [1] 1120 1160  963 1210 1160 1160  813 1230 1370 1140  995  935 1110  994 1020
##  [16]  960 1180  799  958 1140 1100 1210 1150 1250 1260 1220 1030 1100  774  840
##  [31]  874  694  940  833  701  916  692 1020 1050  969  831  726  456  824  702
##  [46] 1120 1100  832  764  821  768  845  864  862  698  845  744  796 1040  759
##  [61]  781  865  845  944  984  897  822 1010  771  676  649  846  812  742  801
##  [76] 1040  860  874  848  890  744  749  838 1050  918  986  797  923  975  815
##  [91] 1020  906  901 1170  912  746  919  718  714  740
```

```r
# List the number of observations in the Nile dataset
length(Nile)
```

```
## [1] 100
```

```r
# Display the first 10 elements of the Nile dataset
head(Nile, 10)
```

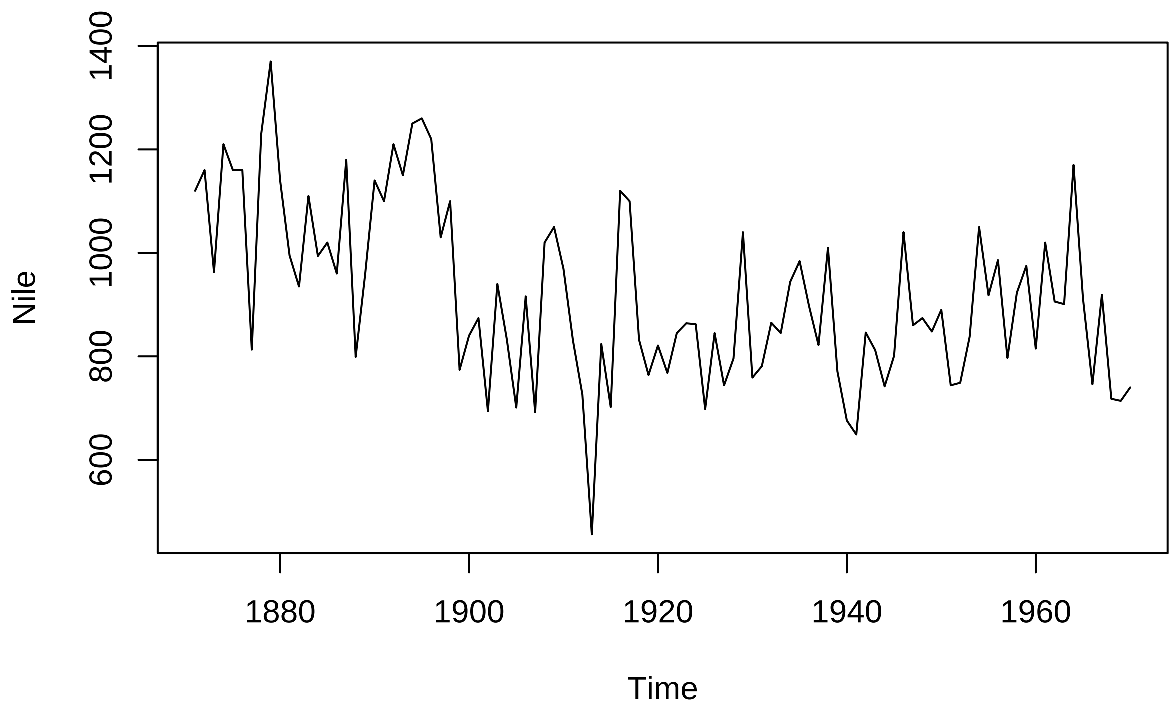
```
##  [1] 1120 1160  963 1210 1160 1160  813 1230 1370 1140
```

```r
# Display the last 12 elements of the Nile dataset
tail(Nile, 12)
```

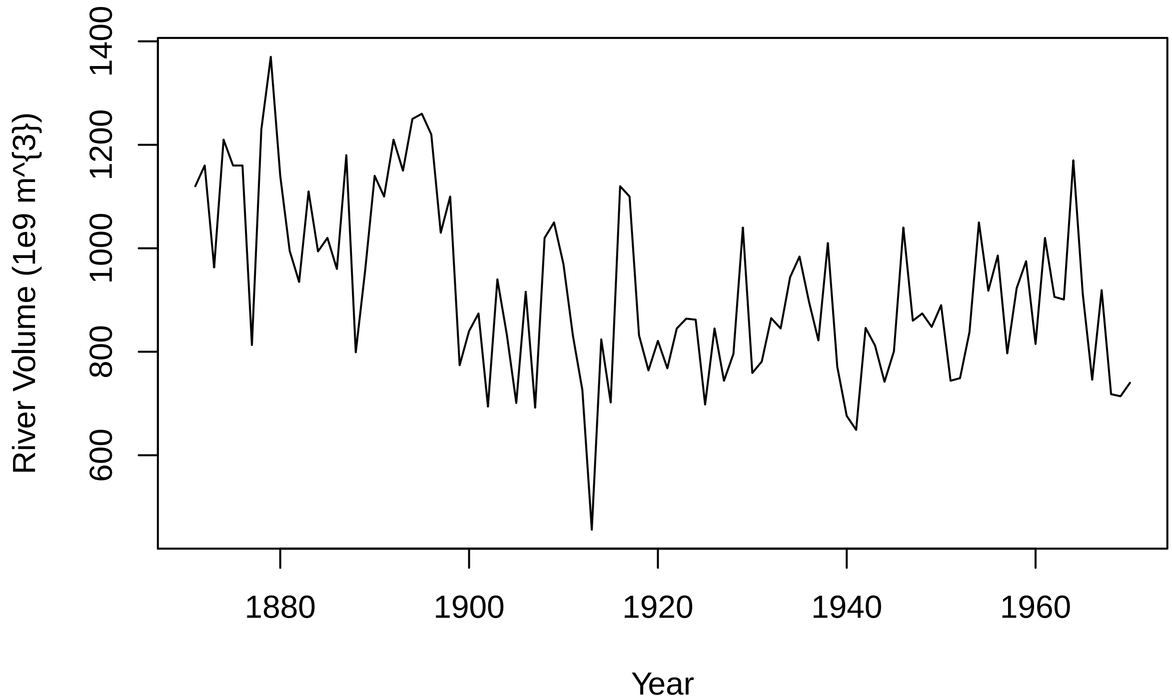
```
##  [1]  975  815 1020  906  901 1170  912  746  919  718  714  740
```

Plot time series data

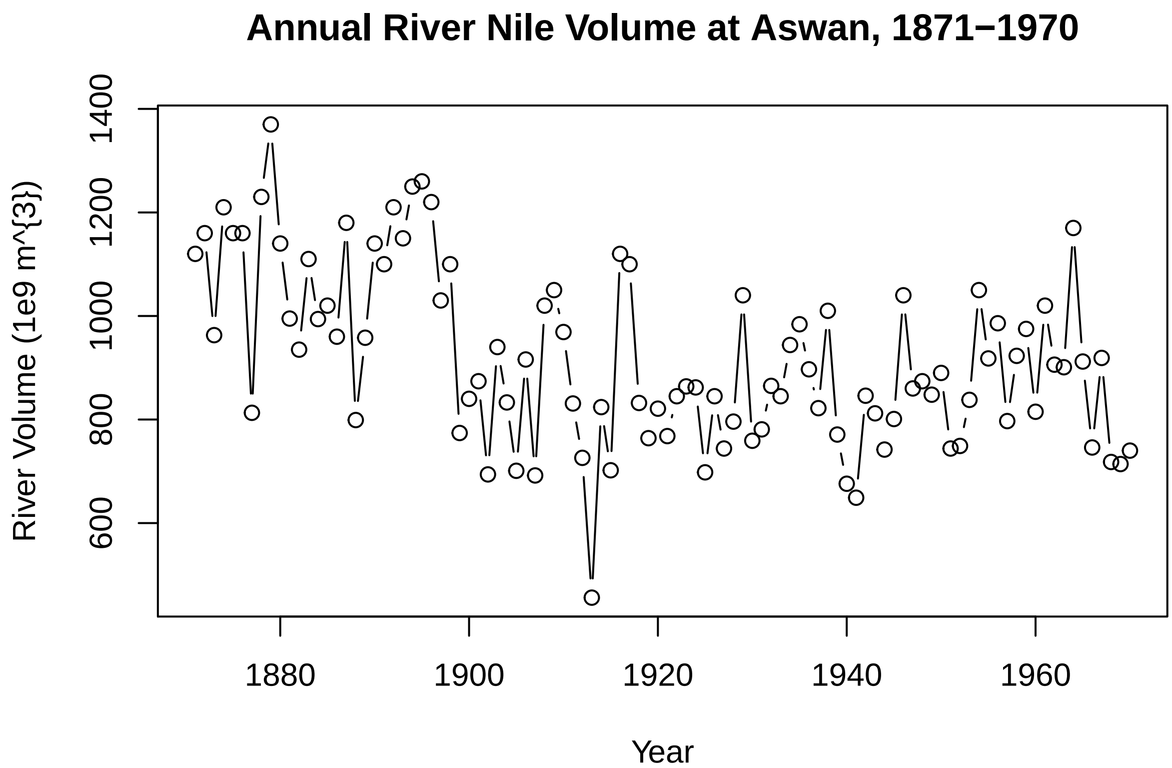
```r
# Plot the Nile data
plot(Nile)
```




```r
# Plot the Nile data with xlab and ylab arguments
plot(Nile, xlab = "Year", ylab = "River Volume (1e9 m^{3})")
```




```r
# Plot the Nile data with xlab, ylab, main, and type arguments
plot(Nile, xlab = "Year", ylab = "River Volume (1e9 m^{3})", main = "Annual River Nile Volume at Aswan,
```

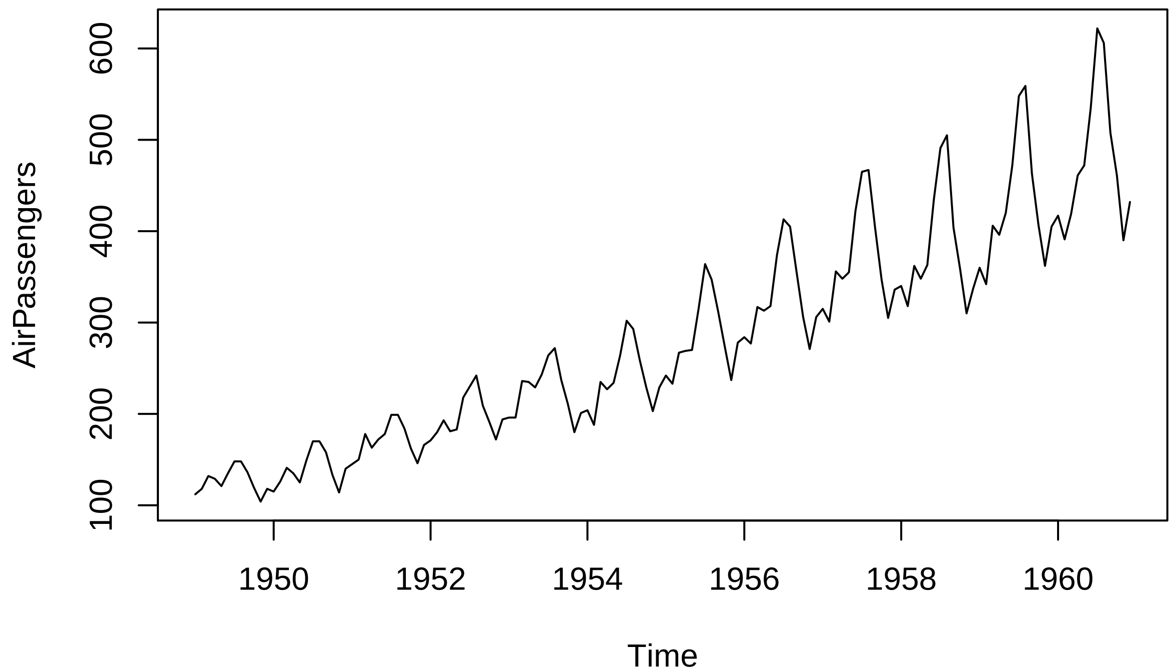


Identifying time series frequency

The start() and end() functions return the time index of the first and last observations, respectively. The time() function calculates a vector of time indices, with one element for each time index on which the series was observed.

The deltat() function returns the fixed time interval between observations and the frequency() function returns the number of observations per unit time. Finally, the cycle() function returns the position in the cycle of each observation.

```
# Plot AirPassengers
plot(AirPassengers)
```




```r
# View the start and end dates of AirPassengers
start(AirPassengers)
```

```
## [1] 1949    1
```

```r
end(AirPassengers)
```

```
## [1] 1960   12
```

```r
# Use time(), deltat(), frequency(), and cycle() with AirPassengers
time(AirPassengers)
```

```
##           Jan      Feb      Mar      Apr      May      Jun      Jul      Aug
## 1949 1949.000 1949.083 1949.167 1949.250 1949.333 1949.417 1949.500 1949.583
## 1950 1950.000 1950.083 1950.167 1950.250 1950.333 1950.417 1950.500 1950.583
## 1951 1951.000 1951.083 1951.167 1951.250 1951.333 1951.417 1951.500 1951.583
## 1952 1952.000 1952.083 1952.167 1952.250 1952.333 1952.417 1952.500 1952.583
## 1953 1953.000 1953.083 1953.167 1953.250 1953.333 1953.417 1953.500 1953.583
## 1954 1954.000 1954.083 1954.167 1954.250 1954.333 1954.417 1954.500 1954.583
## 1955 1955.000 1955.083 1955.167 1955.250 1955.333 1955.417 1955.500 1955.583
## 1956 1956.000 1956.083 1956.167 1956.250 1956.333 1956.417 1956.500 1956.583
## 1957 1957.000 1957.083 1957.167 1957.250 1957.333 1957.417 1957.500 1957.583
## 1958 1958.000 1958.083 1958.167 1958.250 1958.333 1958.417 1958.500 1958.583
## 1959 1959.000 1959.083 1959.167 1959.250 1959.333 1959.417 1959.500 1959.583
```

```
## 1960 1960.000 1960.083 1960.167 1960.250 1960.333 1960.417 1960.500 1960.583
##           Sep      Oct      Nov      Dec
## 1949 1949.667 1949.750 1949.833 1949.917
## 1950 1950.667 1950.750 1950.833 1950.917
## 1951 1951.667 1951.750 1951.833 1951.917
## 1952 1952.667 1952.750 1952.833 1952.917
## 1953 1953.667 1953.750 1953.833 1953.917
## 1954 1954.667 1954.750 1954.833 1954.917
## 1955 1955.667 1955.750 1955.833 1955.917
## 1956 1956.667 1956.750 1956.833 1956.917
## 1957 1957.667 1957.750 1957.833 1957.917
## 1958 1958.667 1958.750 1958.833 1958.917
## 1959 1959.667 1959.750 1959.833 1959.917
## 1960 1960.667 1960.750 1960.833 1960.917
```

```
deltat(AirPassengers)
```

```
## [1] 0.08333333
```

```
frequency(AirPassengers)
```

```
## [1] 12
```

```
cycle(AirPassengers)
```

```
##      Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
## 1949   1   2   3   4   5   6   7   8   9  10  11  12
## 1950   1   2   3   4   5   6   7   8   9  10  11  12
## 1951   1   2   3   4   5   6   7   8   9  10  11  12
## 1952   1   2   3   4   5   6   7   8   9  10  11  12
## 1953   1   2   3   4   5   6   7   8   9  10  11  12
## 1954   1   2   3   4   5   6   7   8   9  10  11  12
## 1955   1   2   3   4   5   6   7   8   9  10  11  12
## 1956   1   2   3   4   5   6   7   8   9  10  11  12
## 1957   1   2   3   4   5   6   7   8   9  10  11  12
## 1958   1   2   3   4   5   6   7   8   9  10  11  12
## 1959   1   2   3   4   5   6   7   8   9  10  11  12
## 1960   1   2   3   4   5   6   7   8   9  10  11  12
```

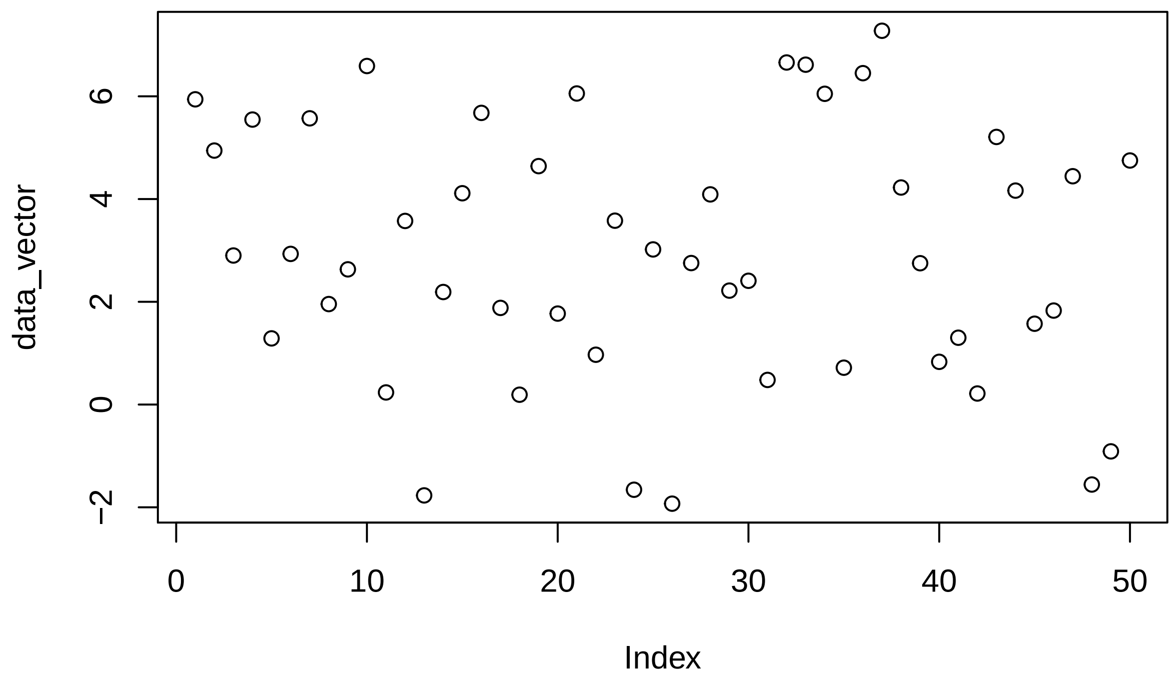
Creating ts object

```
data_vector <- rnorm(50, 3, 2.6)
# Use print() and plot() to view data_vector
print(data_vector)
```

```
##  [1]  5.9415268  4.9438253  2.9013574  5.5471766  1.2879561  2.9324190
##  [7]  5.5712679  1.9559237  2.6318662  6.5897333  0.2354899  3.5739166
## [13] -1.7685905  2.1905787  4.1131288  5.6773031  1.8811917  0.1917611
```

```
## [19]  4.6412255  1.7706036  6.0549922  0.9697190  3.5789989 -1.6585750
## [25]  3.0196248 -1.9292188  2.7540420  4.0909909  2.2180470  2.4091036
## [31]  0.4791841  6.6582224  6.6156499  6.0487299  0.7174654  6.4507767
## [37]  7.2757952  4.2248151  2.7514062  0.8315938  1.3010152  0.2151594
## [43]  5.2088136  4.1648240  1.5732000  1.8294604  4.4443890 -1.5564813
## [49] -0.9114915  4.7508199
```

```r
plot(data_vector)
```

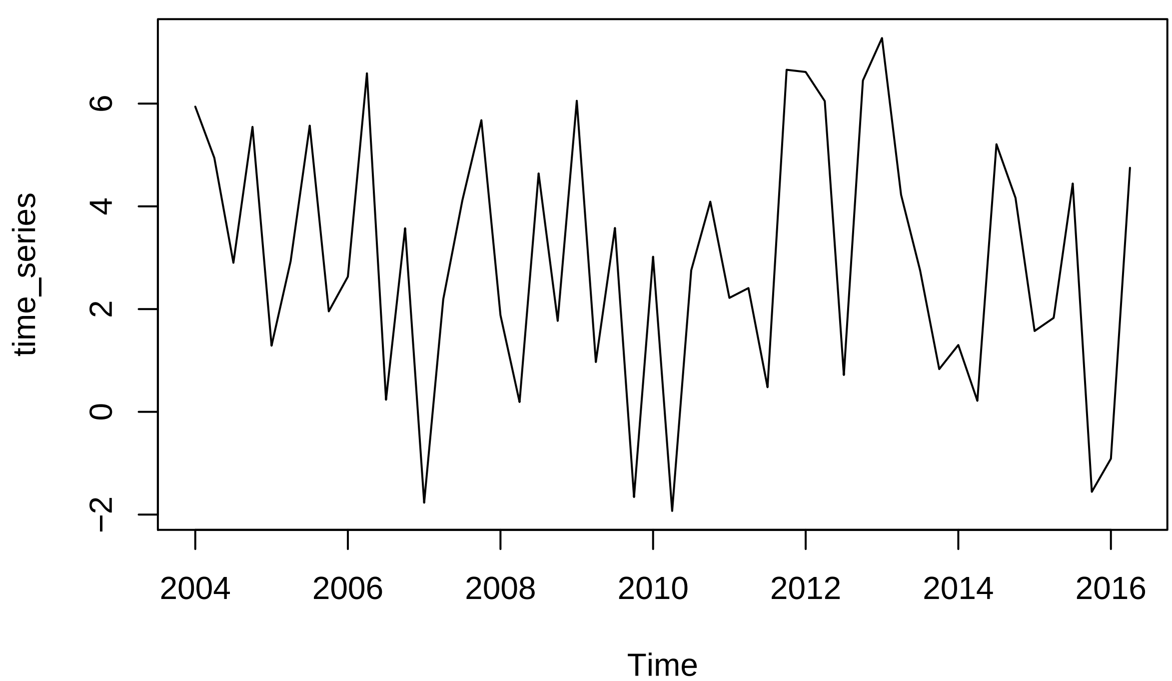



```r
# Convert data_vector to a ts object with start = 2004 and frequency = 4
time_series <- ts(data_vector, start = 2004, frequency = 4)

# Use print() and plot() to view time_series
print(time_series)
```

```
##            Qtr1       Qtr2       Qtr3       Qtr4
## 2004  5.9415268  4.9438253  2.9013574  5.5471766
## 2005  1.2879561  2.9324190  5.5712679  1.9559237
## 2006  2.6318662  6.5897333  0.2354899  3.5739166
## 2007 -1.7685905  2.1905787  4.1131288  5.6773031
## 2008  1.8811917  0.1917611  4.6412255  1.7706036
## 2009  6.0549922  0.9697190  3.5789989 -1.6585750
## 2010  3.0196248 -1.9292188  2.7540420  4.0909909
## 2011  2.2180470  2.4091036  0.4791841  6.6582224
## 2012  6.6156499  6.0487299  0.7174654  6.4507767
```

```
## 2013  7.2757952  4.2248151  2.7514062  0.8315938
## 2014  1.3010152  0.2151594  5.2088136  4.1648240
## 2015  1.5732000  1.8294604  4.4443890 -1.5564813
## 2016 -0.9114915  4.7508199
```

```
plot(time_series)
```



Check wheter a variable is time series

```
# Check whether data_vector and time_series are ts objects
is.ts(data_vector)
```

```
## [1] FALSE
```

```
is.ts(time_series)
```

```
## [1] TRUE
```

```
# Check whether Nile is a ts object
is.ts(Nile)
```

```
## [1] TRUE
```

```r
# Check whether AirPassengers is a ts object
is.ts(AirPassengers)
```

```
## [1] TRUE
```

Plotting time series object

```r
eu_stocks <- EuStockMarkets
# Check whether eu_stocks is a ts object
is.ts(eu_stocks)
```

```
## [1] TRUE
```

```r
# View the start, end, and frequency of eu_stocks
start(eu_stocks)
```

```
## [1] 1991  130
```

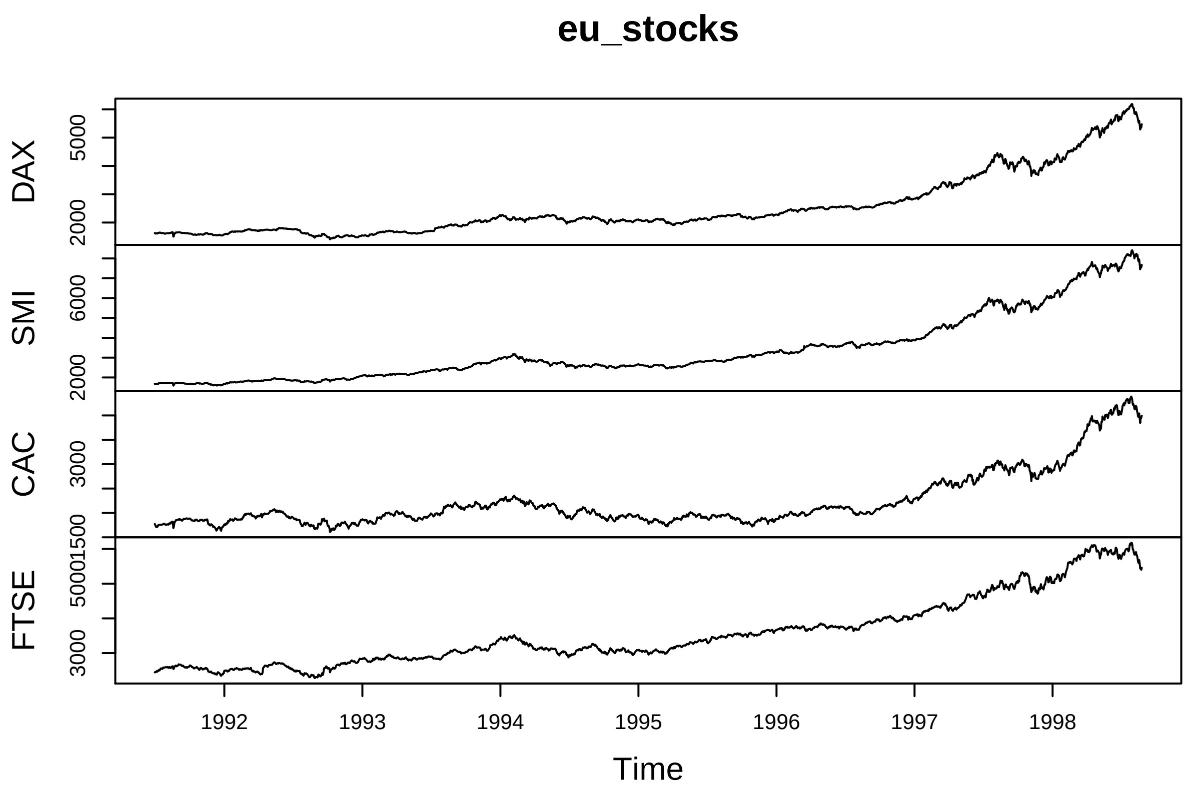
```r
end(eu_stocks)
```

```
## [1] 1998  169
```

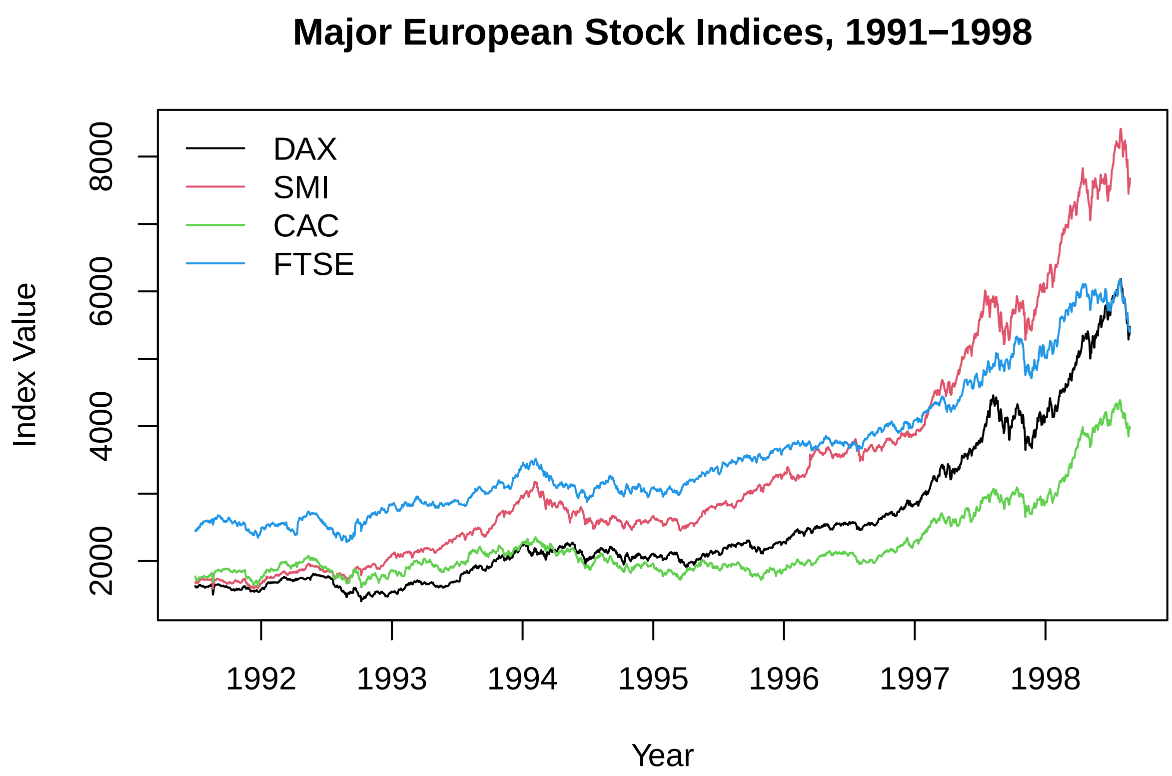
```r
frequency(eu_stocks)
```

```
## [1] 260
```

```r
# Generate a simple plot of eu_stocks
plot(eu_stocks)
```




```r
# Use ts.plot with eu_stocks
ts.plot(eu_stocks, col = 1:4, xlab = "Year", ylab = "Index Value", main = "Major European Stock Indices

# Add a legend to your ts.plot
legend("topleft", colnames(eu_stocks), lty = 1, col = 1:4, bty = "n")
```



Trend spotting

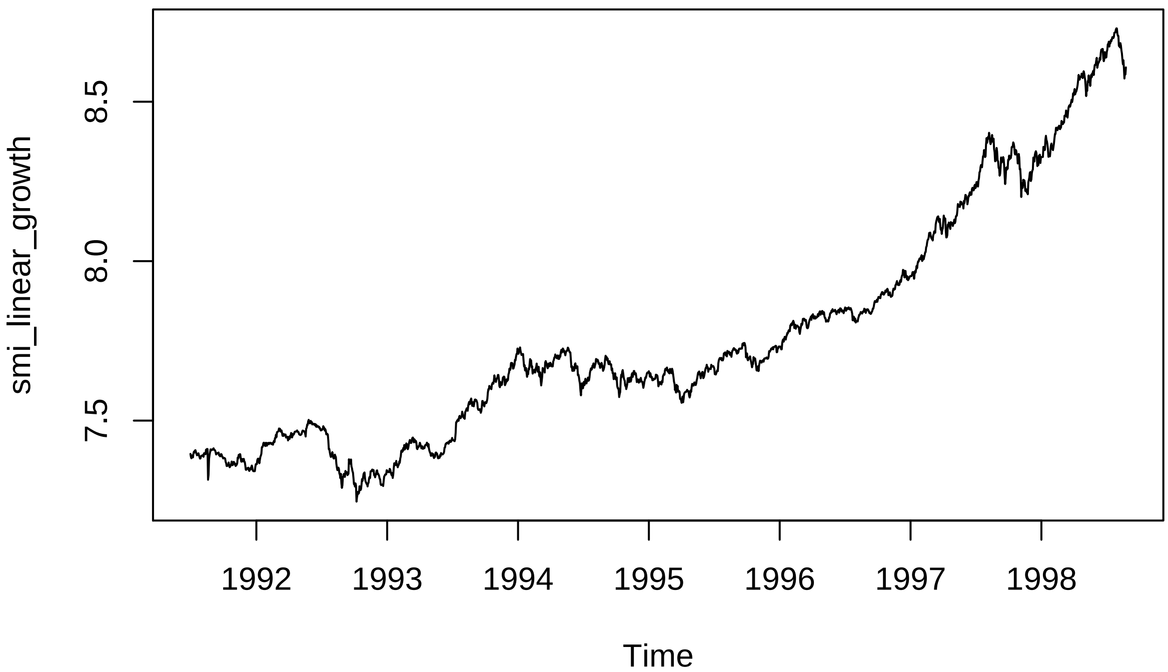
Removing trends in variability via the logarithmic transformation

The logarithmic function log() is a data transformation that can be applied to positively valued time series data. It slightly shrinks observations that are greater than one towards zero, while greatly shrinking very large observations. This property can stabilize variability when a series exhibits increasing variability over time. It may also be used to linearize a rapid growth pattern over time.

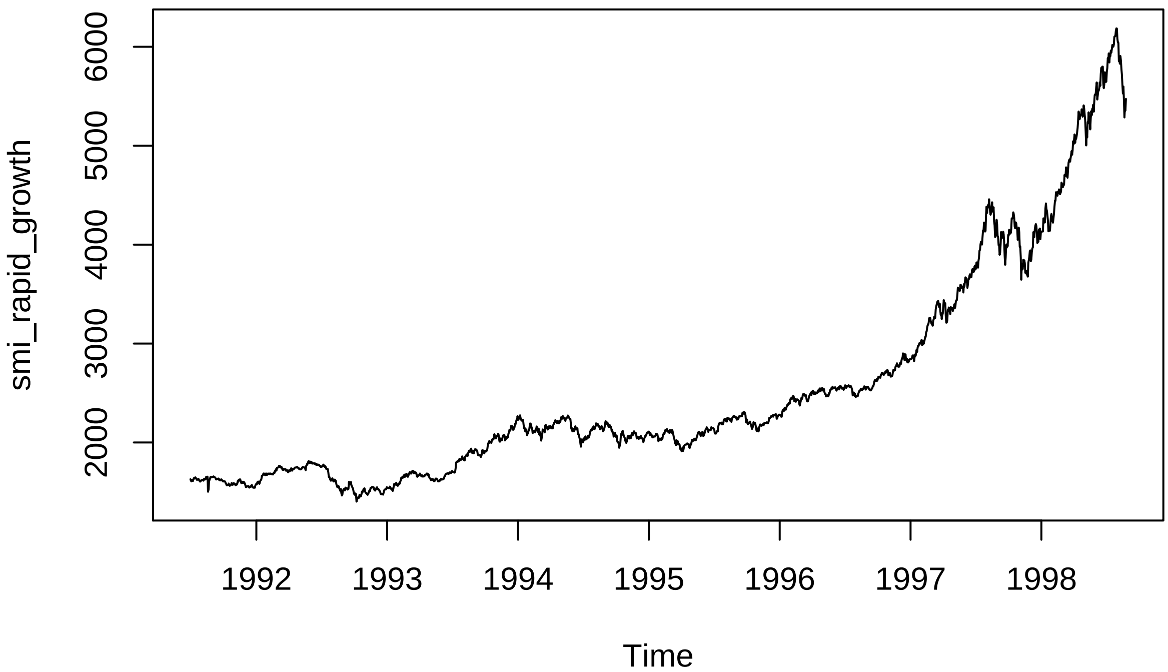
```
colnames(eu_stocks)
```

```
## [1] "DAX"  "SMI"  "CAC"  "FTSE"
```

```
smi_rapid_growth <- eu_stocks[,1]
smi_linear_growth <- log(smi_rapid_growth)
ts.plot(smi_linear_growth)
```




```
ts.plot(smi_rapid_growth)
```



Removing trends in level by differencing

The first difference transformation of a time series z[t] consists of the differences (changes) between successive observations over time, that is z[t] - z[t-1].

Differencing a time series can remove a time trend. The function diff() will calculate the first difference or change series. A difference series lets us examine the increments or changes in a given time series. It always has one fewer observations than the original series.

By removing the long-term time trend, we can view the amount of change from one observation to the next.

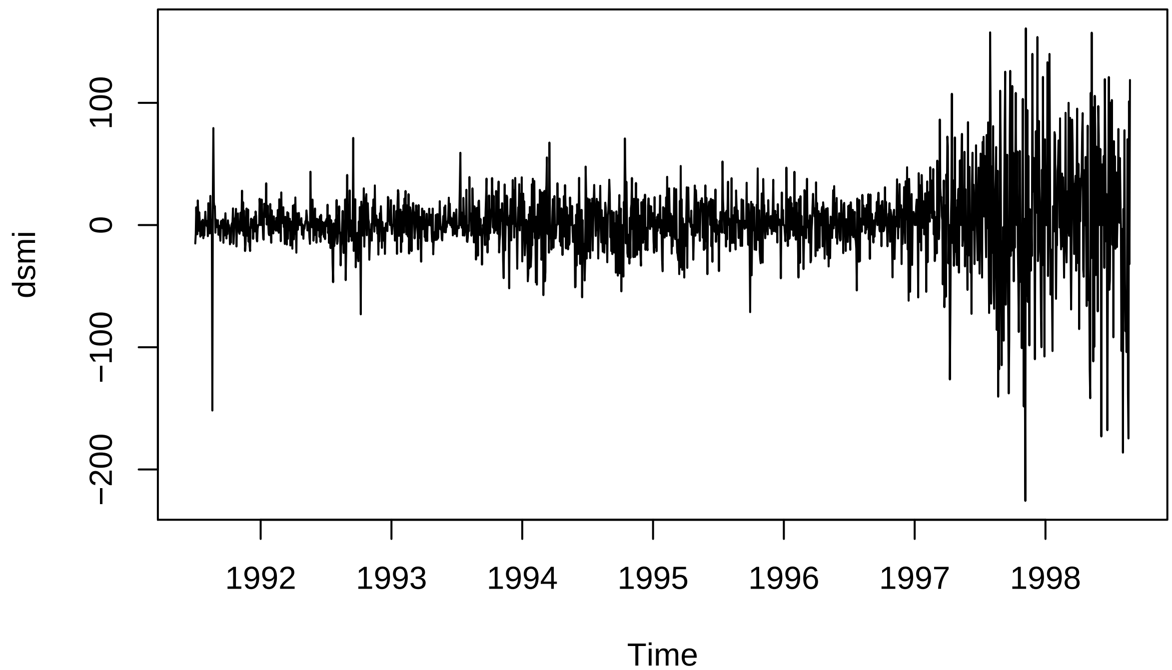
```
dsmi <- diff(smi_rapid_growth)
length(smi_rapid_growth)
```

```
## [1] 1860
```

```
length(dsmi)
```

```
## [1] 1859
```

```
ts.plot(dsmi)
```

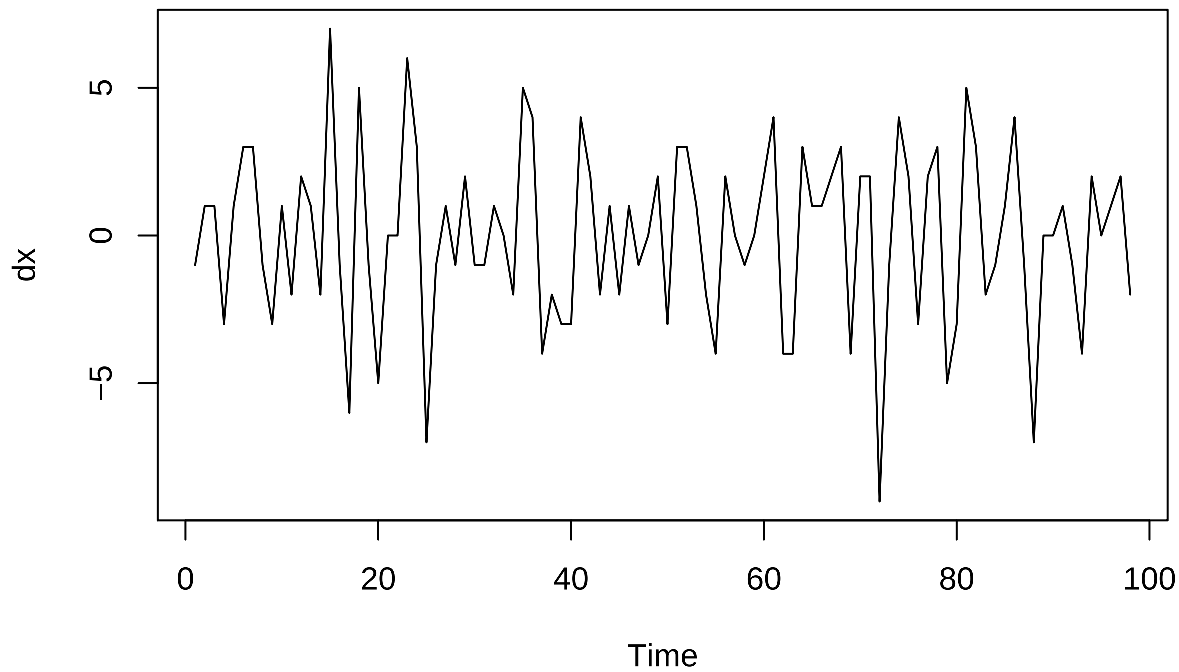


Removing seasonal trends with seasonal differencing

For time series exhibiting seasonal trends, seasonal differencing can be applied to remove these periodic patterns. For example, monthly data may exhibit a strong twelve month pattern. In such situations, changes in behavior from year to year may be of more interest than changes from month to month, which may largely follow the overall seasonal pattern.

The function diff(. . . , lag = s) will calculate the lag s difference or length s seasonal change series. For monthly or quarterly data, an appropriate value of s would be 12 or 4, respectively. The diff() function has lag = 1 as its default for first differencing. Similar to before, a seasonally differenced series will have s fewer observations than the original series.

```
# create x
x <- rpois(100, 3)
# Generate a diff of x with lag = 4. Save this to dx
dx <- diff(x, lag = 2)

# Plot dx
ts.plot(dx)
```




```r
# View the length of x and dx, respectively
length(x)
```

```
## [1] 100
```

```r
length(dx)
```

```
## [1] 98
```

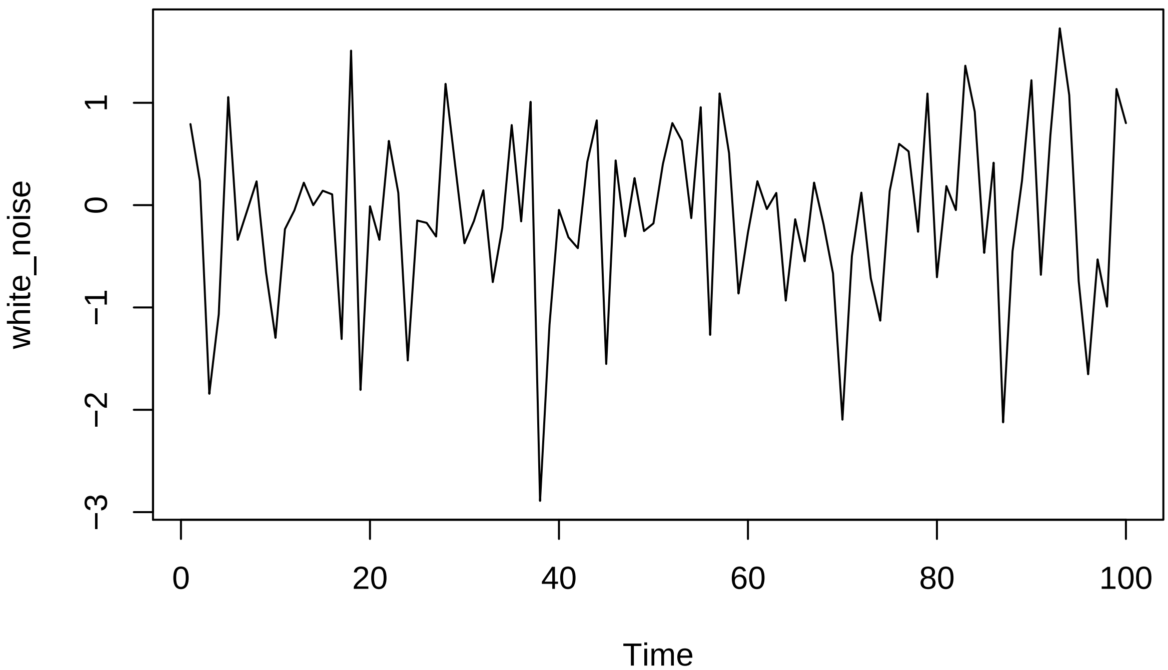
Simulate the white noise model

The white noise (WN) model is a basic time series model. We will focus on the simplest form of WN, independent and identically distributed data.

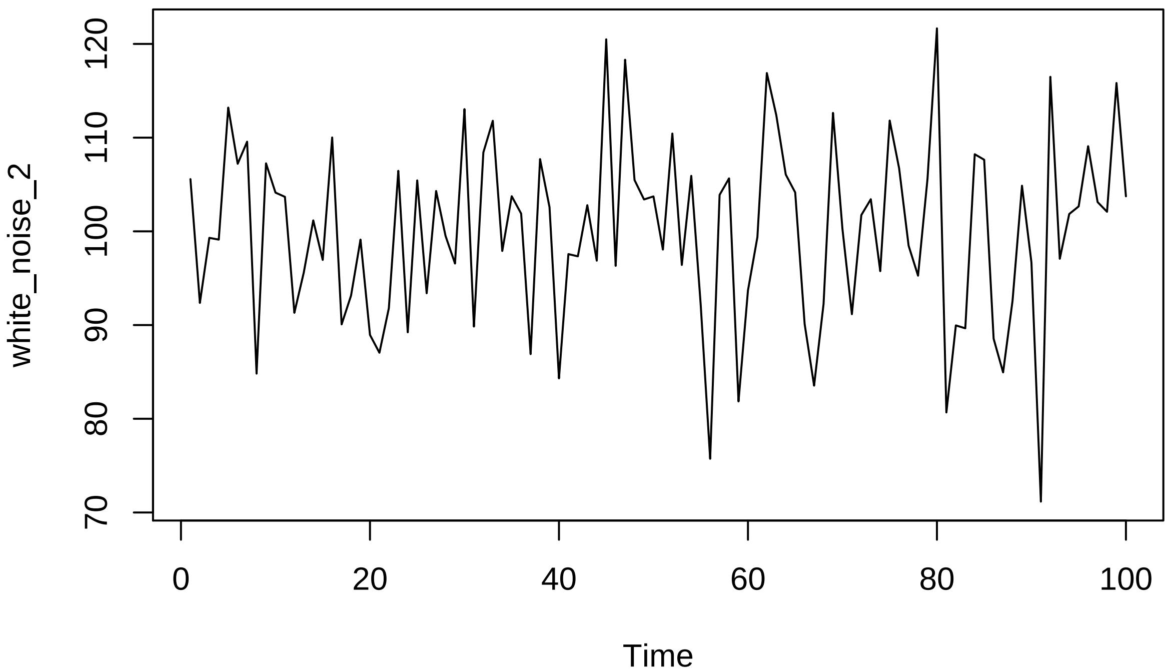
The arima.sim() function can be used to simulate data from a variety of time series models. ARIMA is an abbreviation for the autoregressive integrated moving average class of models.

An ARIMA(p, d, q) model has three parts, the autoregressive order p, the order of integration (or differencing) d, and the moving average order q.

```r
# Simulate a WN model with list(order = c(0, 0, 0))
white_noise <- arima.sim(model = list(order = c(0,0,0)), n = 100)

# Plot your white_noise data
ts.plot(white_noise)
```




```r
# Simulate from the WN model with: mean = 100, sd = 10
white_noise_2 <- arima.sim(model = list(order = c(0,0,0)), n = 100, mean = 100, sd = 10)

# Plot your white_noise_2 data
ts.plot(white_noise_2)
```



Estimate the white noise model

For a given time series y we can fit the white noise (WN) model using the arima(..., order = c(0, 0, 0)) function. Recall that the WN model is an ARIMA(0,0,0) model. Applying the arima() function returns information or output about the estimated model. For the WN model this includes the estimated mean, labeled intercept, and the estimated variance, labeled sigma ˆ 2.

```
y <- white_noise
# Fit the WN model to y using the arima command
arima(y, order = c(0, 0, 0))
```

```
##
## Call:
## arima(x = y, order = c(0, 0, 0))
##
## Coefficients:
##       intercept
##         -0.1037
## s.e.     0.0856
##
## sigma^2 estimated as 0.732:  log likelihood = -126.29,  aic = 256.59
```

```
# Calculate the sample mean and sample variance of y
mean(y)
```

```
## [1] -0.1037405
```

```r
var(y)
```

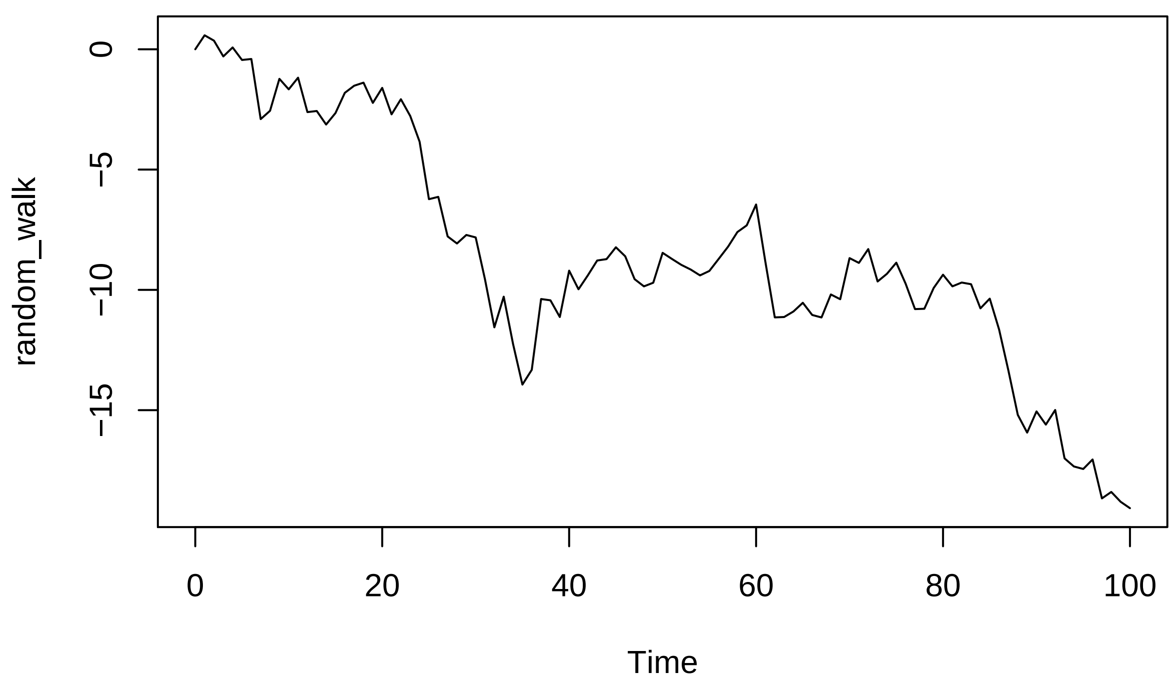
```
## [1] 0.7393648
```

Simulate the random walk model

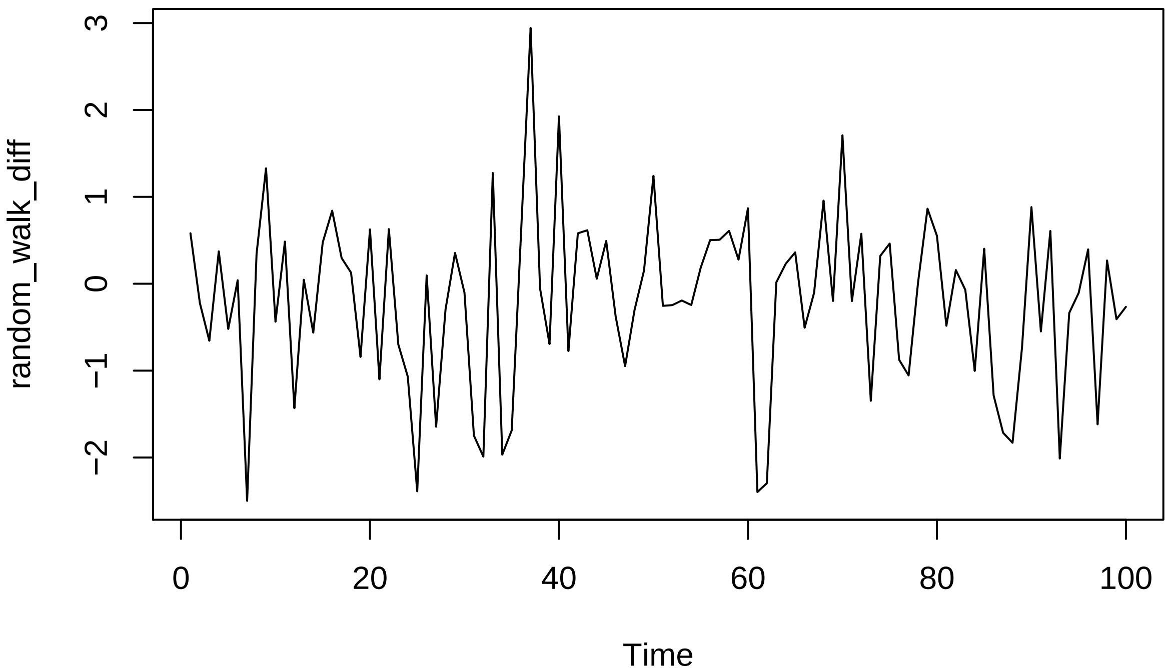
The random walk (RW) model is also a basic time series model. It is the cumulative sum (or integration) of a mean zero white noise (WN) series, such that the first difference series of a RW is a WN series. Note for reference that the RW model is an ARIMA(0, 1, 0) model, in which the middle entry of 1 indicates that the model's order of integration is 1.

The arima.sim() function can be used to simulate data from the RW by including the model = list(order = c(0, 1, 0)) argument. We also need to specify a series length n. Finally, you can specify a sd for the series (increments), where the default value is 1.

```r
# Generate a RW model using arima.sim
random_walk <- arima.sim(model = list(order = c(0, 1, 0)), n = 100)

# Plot random_walk
ts.plot(random_walk)
```



```r
# Calculate the first difference series
random_walk_diff <- diff(random_walk)

# Plot random_walk_diff
ts.plot(random_walk_diff)
```



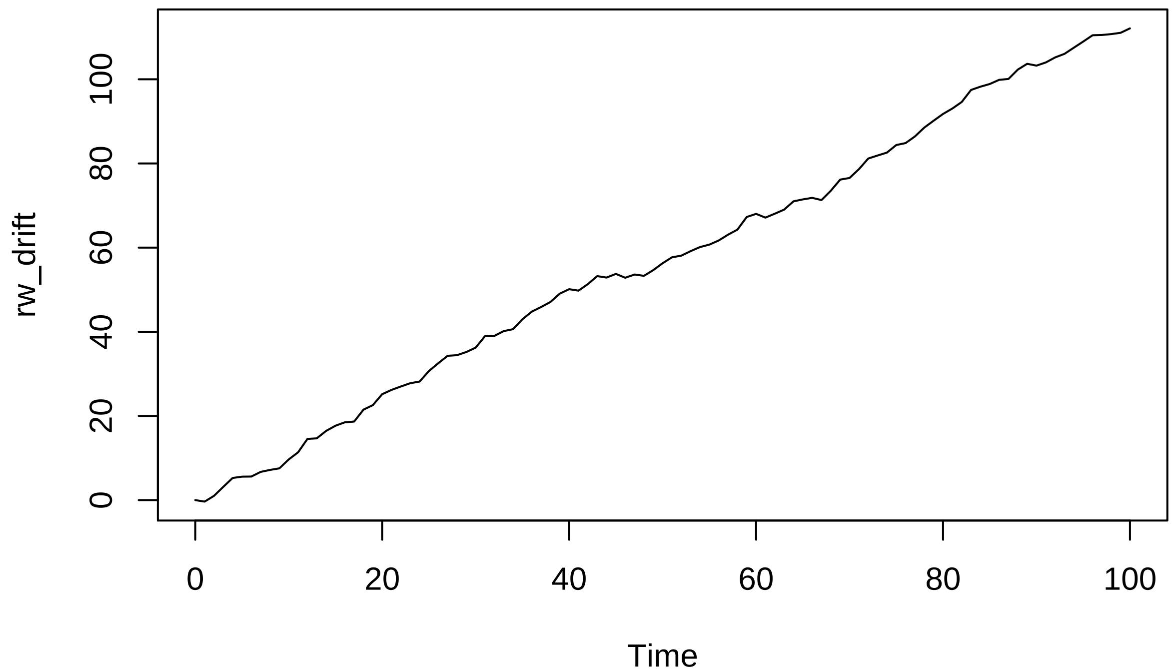
Simulate the random walk model with a drift

A random walk (RW) need not wander about zero, it can have an upward or downward trajectory, i.e., a drift or time trend. This is done by including an intercept in the RW model, which corresponds to the slope of the RW time trend.

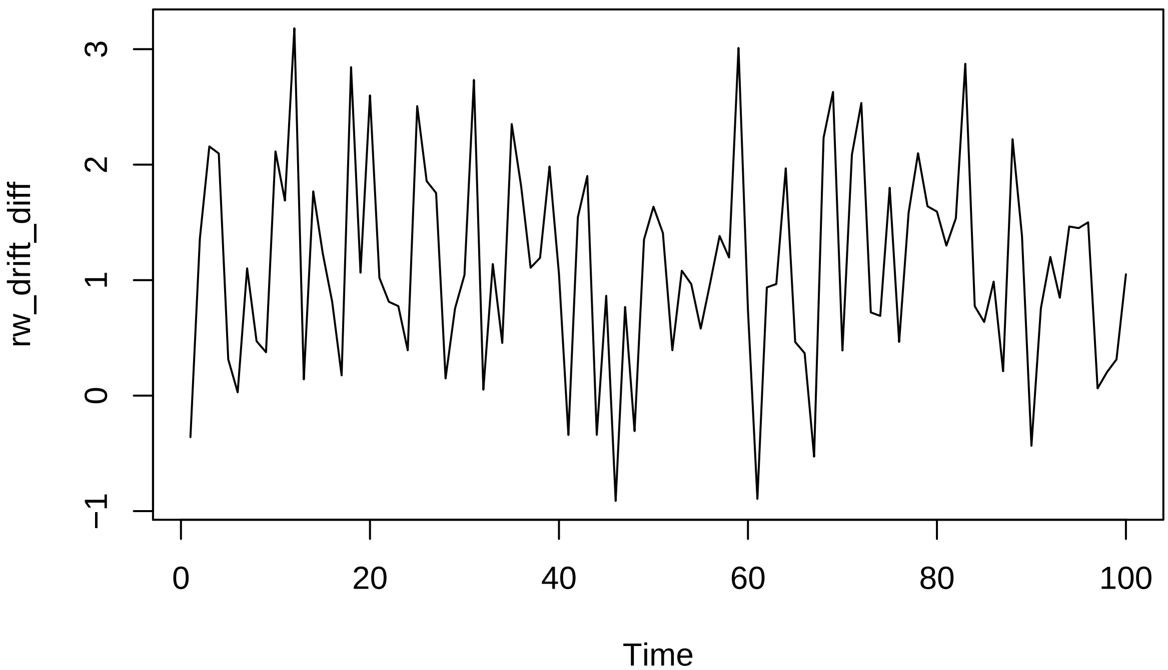
For an alternative formulation, you can take the cumulative sum of a constant mean white noise (WN) series, such that the mean corresponds to the slope of the RW time trend.

To simulate data from the RW model with a drift you again use the arima.sim() function with the model = list(order = c(0, 1, 0)) argument. This time, you should add the additional argument mean = … to specify the drift variable, or the intercept.

```r
# Generate a RW model with a drift uing arima.sim
rw_drift <- arima.sim(model = list(order = c(0, 1, 0)), n = 100, mean = 1)

# Plot rw_drift
ts.plot(rw_drift)
```




```
# Calculate the first difference series
rw_drift_diff <- diff(rw_drift)

# Plot rw_drift_diff
ts.plot(rw_drift_diff)
```

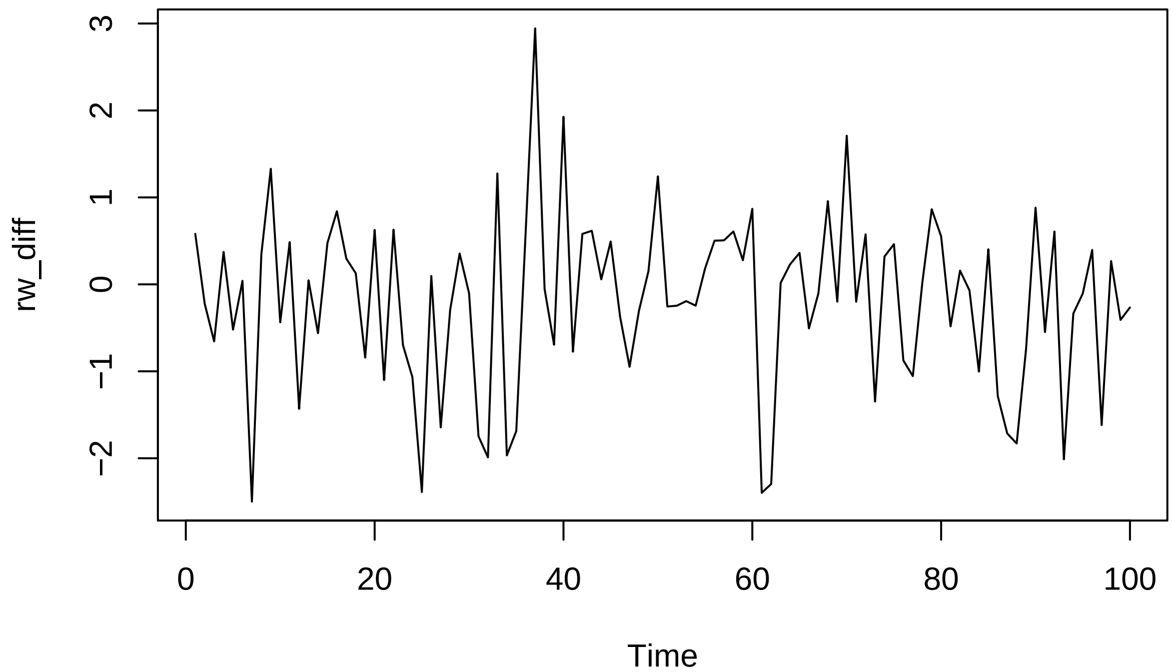


Estimate the random walk model

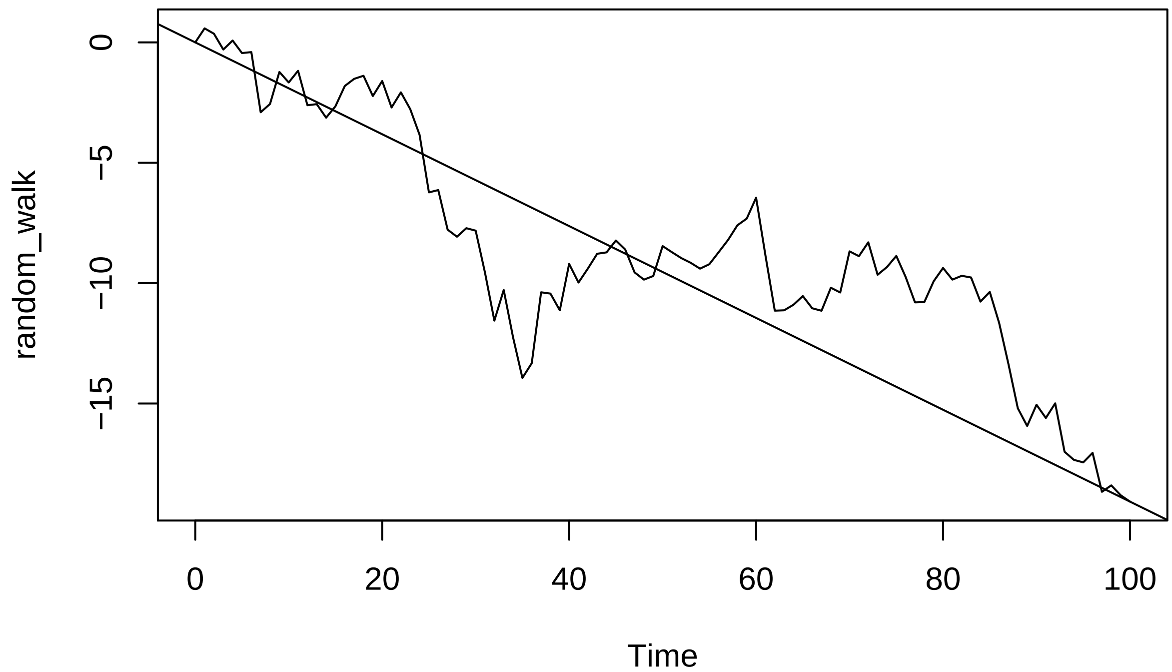
For a given time series y we can fit the random walk model with a drift by first differencing the data, then fitting the white noise (WN) model to the differenced data using the arima() command with the order = c(0, 0, 0)) argument.

The arima() command displays information or output about the fitted model. Under the Coefficients: heading is the estimated drift variable, named the intercept. Its approximate standard error (or s.e.) is provided directly below it. The variance of the WN part of the model is also estimated under the label sigma ^ 2.

```
# Difference your random_walk data
rw_diff <- diff(random_walk)

# Plot rw_diff
ts.plot(rw_diff)
```




```r
# Now fit the WN model to the differenced data
model_wn <-arima(rw_diff, order = c(0, 0, 0))

# Store the value of the estimated time trend (intercept)
int_wn <- model_wn$coef

# Plot the original random_walk data
ts.plot(random_walk)

# Use abline(0, ...) to add time trend to the figure
abline(0, int_wn)
```

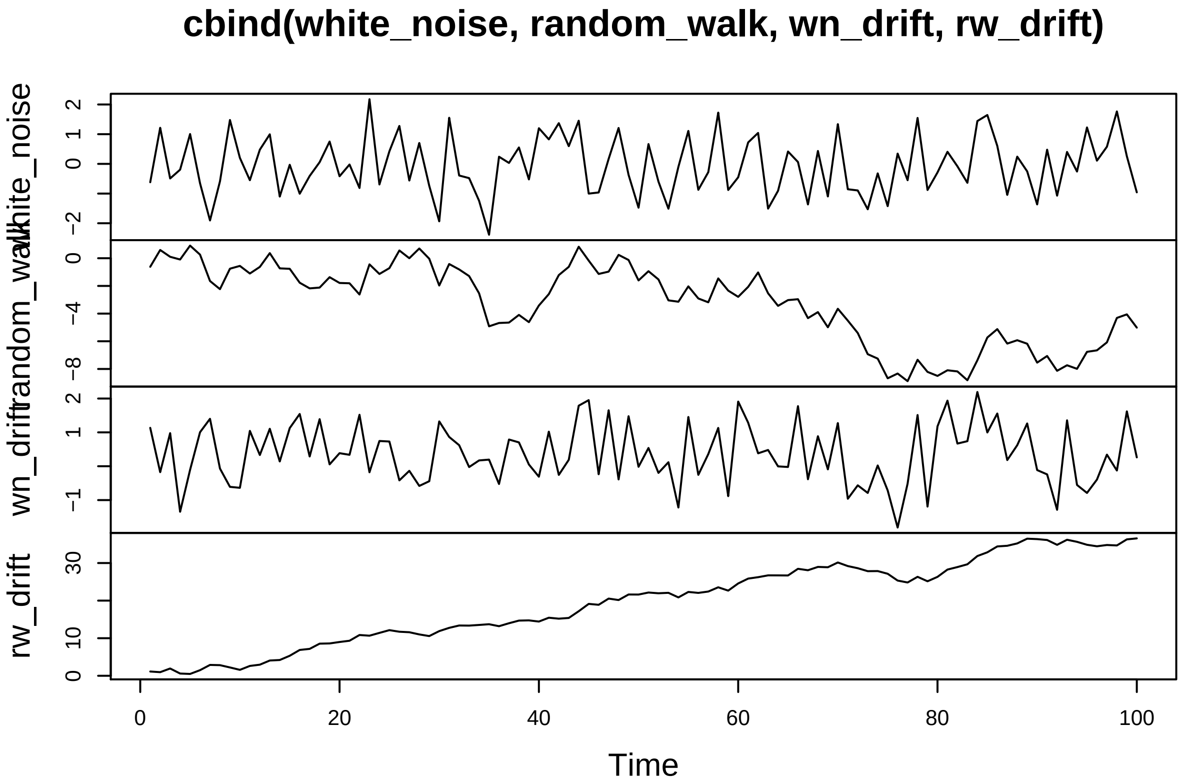


Are the white noise model or the random walk model stationary?

The white noise (WN) and random walk (RW) models are very closely related. However, only the RW is always non-stationary, both with and without a drift term. This is a simulation exercise to highlight the differences.

Recall that if we start with a mean zero WN process and compute its running or cumulative sum, the result is a RW process. The cumsum() function will make this transformation for you. Similarly, if we create a WN process, but change its mean from zero, and then compute its cumulative sum, the result is a RW process with a drift.

```
# Use arima.sim() to generate WN data
white_noise <- arima.sim(model = list(order = c(0, 0, 0)), n = 100)

# Use cumsum() to convert your WN data to RW
random_walk <- cumsum(white_noise)

# Use arima.sim() to generate WN drift data
wn_drift <- arima.sim(model = list(order = c(0, 0, 0)), n = 100, mean = 0.4)

# Use cumsum() to convert your WN drift data to RW
rw_drift <- cumsum(wn_drift)

# Plot all four data objects
plot.ts(cbind(white_noise, random_walk, wn_drift, rw_drift))
```



Asset prices vs. asset returns

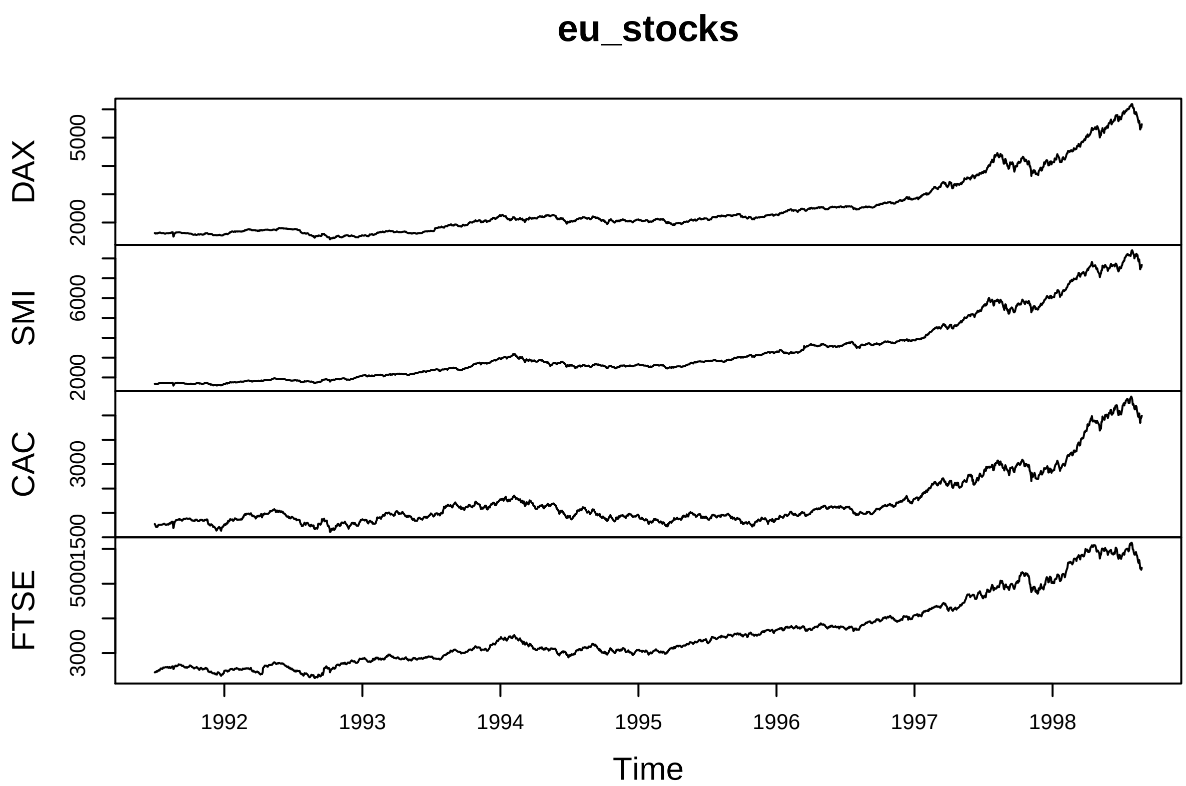
The goal of investing is to make a profit. The revenue or loss from investing depends on the amount invested and changes in prices, and high revenue relative to the size of an investment is of central interest. This is what financial asset returns measure, changes in price as a fraction of the initial price over a given time horizon, for example, one business day.

Let's again consider the eu_stocks dataset. This dataset reports index values, which we can regard as prices. The indices are not investable assets themselves, but there are many investable financial assets that closely track major market indices, including mutual funds and exchange traded funds.

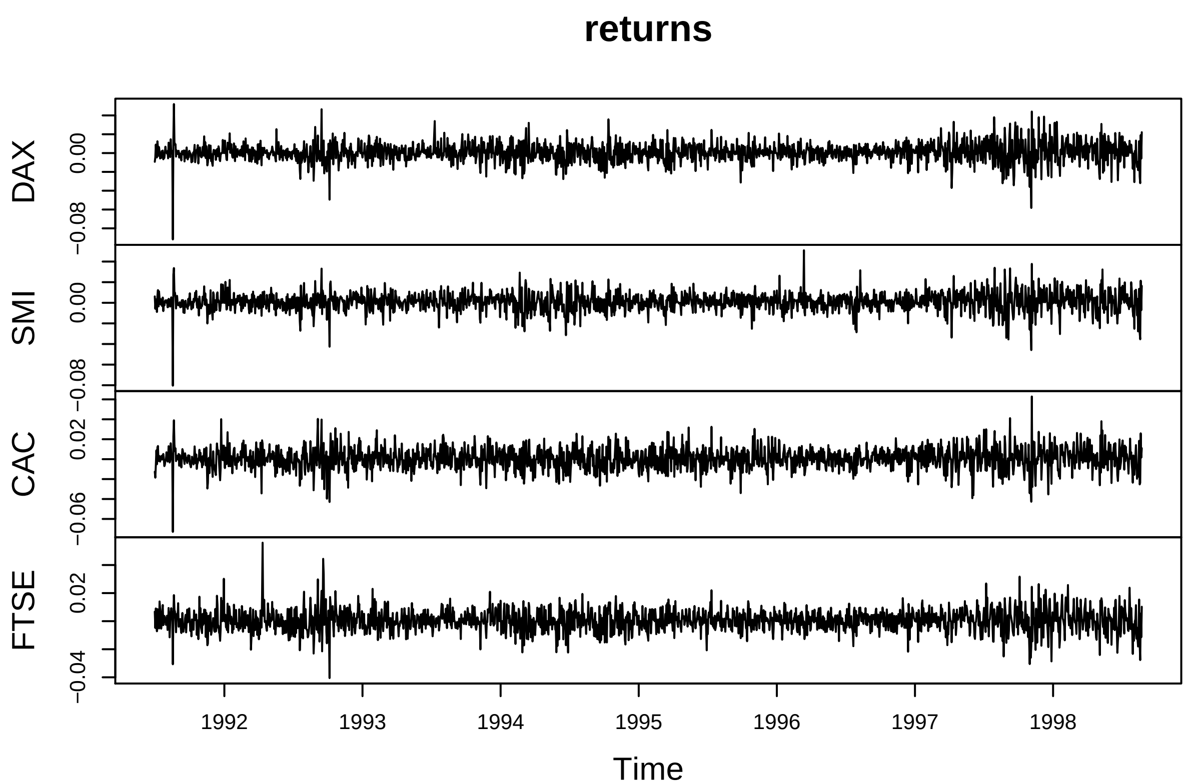
Log returns, also called continuously compounded returns, are also commonly used in financial time series analysis. They are the log of gross returns, or equivalently, the changes (or first differences) in the logarithm of prices.

The change in appearance between daily prices and daily returns is typically substantial, while the difference between daily returns and log returns is usually small. As you'll see later, one advantage of using log returns is that calculating multi-period returns from individual periods is greatly simplified - you just add them together!

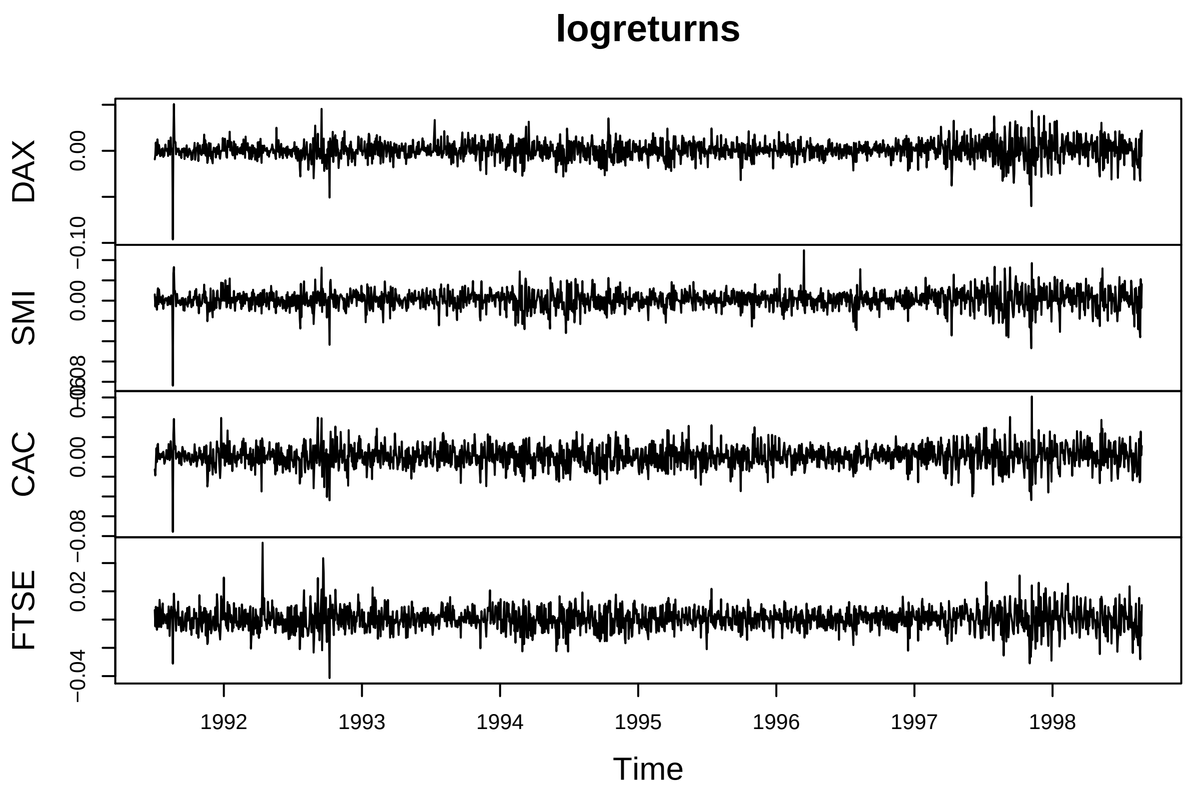
```
# Plot eu_stocks
plot(eu_stocks)
```




```
# Use this code to convert prices to returns
returns <- eu_stocks[-1,] / eu_stocks[-1860,] - 1

# Convert returns to ts
returns <- ts(returns, start = c(1991, 130), frequency = 260)

# Plot returns
plot(returns)
```




```
# Use this code to convert prices to log returns
logreturns <- diff(log(eu_stocks))

# Plot logreturns
plot(logreturns)
```



Characteristics of financial time series

Daily financial asset returns typically share many characteristics. Returns over one day are typically small, and their average is close to zero. At the same time, their variances and standard deviations can be relatively large. Over the course of a few years, several very large returns (in magnitude) are typically observed. These relative outliers happen on only a handful of days, but they account for the most substantial movements in asset prices. Because of these extreme returns, the distribution of daily asset returns is not normal, but heavy-tailed, and sometimes skewed. In general, individual stock returns typically have even greater variability and more extreme observations than index returns.

```
eu_percentreturns <- diff(eu_stocks)[,] / eu_stocks[1:length(diff(eu_stocks))] * 100
head(eu_percentreturns)
```

```
##             DAX        SMI        CAC       FTSE
## [1,] -0.9283193  0.1899988 -0.2952626  0.4207112
## [2,] -0.4412412 -0.5899529 -0.4233811 -0.3036668
## [3,]  0.9044450  0.3257329 -0.5584386  0.5556946
## [4,] -0.1776637  0.1489336  0.8568980  0.5852022
## [5,] -0.4665793 -0.8906835 -0.5122235 -0.7275831
## [6,]  1.2504579  0.6699870  1.1826006  0.8618577
```

```
# Generate means from eu_percentreturns
colMeans(eu_percentreturns)
```

```
##        DAX        SMI        CAC       FTSE
## 0.07052174 0.08634147 0.05152295 0.04674874
```

```r
# Use apply to calculate sample variance from eu_percentreturns
apply(eu_percentreturns, MARGIN = 2, FUN = var)
```

```
##       DAX       SMI       CAC      FTSE
## 1.0569648 0.8492982 1.2081988 0.6337395
```

```r
# Use apply to calculate standard deviation from eu_percentreturns
apply(eu_percentreturns, MARGIN = 2, FUN = sd)
```

```
##       DAX       SMI       CAC      FTSE
## 1.0280879 0.9215737 1.0991810 0.7960775
```

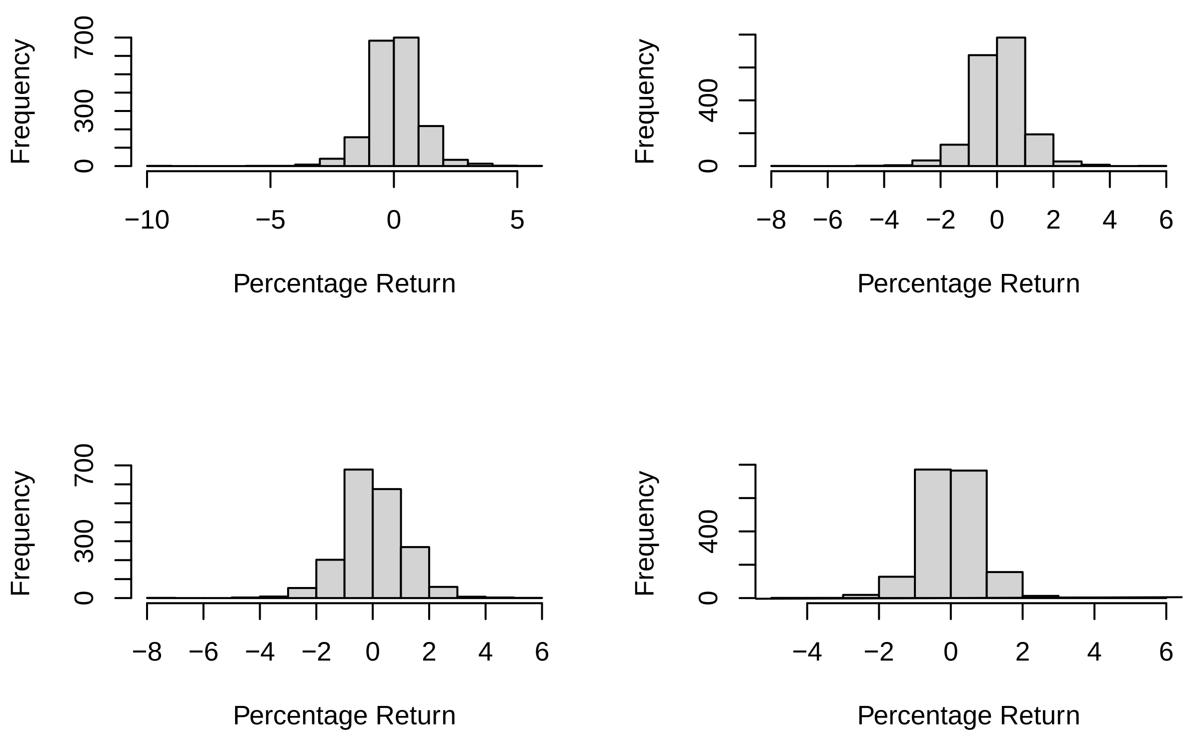
```r
# Display a histogram of percent returns for each index
par(mfrow = c(2,2))
apply(eu_percentreturns, MARGIN = 2, FUN = hist, main = "", xlab = "Percentage Return")
```

```
## $DAX
## $breaks
##  [1] -10  -9  -8  -7  -6  -5  -4  -3  -2  -1   0   1   2   3   4   5   6
##
## $counts
##  [1]   1   0   0   0   1   1   8  40 157 683 700 218  34  13   2   1
##
## $density
##  [1] 0.0005379236 0.0000000000 0.0000000000 0.0000000000 0.0005379236
##  [6] 0.0005379236 0.0043033889 0.0215169446 0.0844540075 0.3674018289
## [11] 0.3765465304 0.1172673480 0.0182894029 0.0069930070 0.0010758472
## [16] 0.0005379236
##
## $mids
##  [1] -9.5 -8.5 -7.5 -6.5 -5.5 -4.5 -3.5 -2.5 -1.5 -0.5  0.5  1.5  2.5  3.5  4.5
## [16]  5.5
##
## $xname
## [1] "newX[, i]"
##
## $equidist
## [1] TRUE
##
## attr(,"class")
## [1] "histogram"
##
## $SMI
## $breaks
##  [1] -8 -7 -6 -5 -4 -3 -2 -1  0  1  2  3  4  5  6
##
## $counts
##  [1]   1   0   0   2   5  34 130 675 782 193  28   8   0   1
##
```

```
## $density
##  [1] 0.0005379236 0.0000000000 0.0000000000 0.0010758472 0.0026896181
##  [6] 0.0182894029 0.0699300699 0.3630984400 0.4206562668 0.1038192577
## [11] 0.0150618612 0.0043033889 0.0000000000 0.0005379236
##
## $mids
##  [1] -7.5 -6.5 -5.5 -4.5 -3.5 -2.5 -1.5 -0.5  0.5  1.5  2.5  3.5  4.5  5.5
##
## $xname
## [1] "newX[, i]"
##
## $equidist
## [1] TRUE
##
## attr(,"class")
## [1] "histogram"
##
## $CAC
## $breaks
##  [1] -8 -7 -6 -5 -4 -3 -2 -1  0  1  2  3  4  5  6
##
## $counts
##  [1]   1   0   0   3   8  53 202 678 575 269  59   7   3   1
##
## $density
##  [1] 0.0005379236 0.0000000000 0.0000000000 0.0016137708 0.0043033889
##  [6] 0.0285099516 0.1086605702 0.3647122109 0.3093060785 0.1447014524
## [11] 0.0317374933 0.0037654653 0.0016137708 0.0005379236
##
## $mids
##  [1] -7.5 -6.5 -5.5 -4.5 -3.5 -2.5 -1.5 -0.5  0.5  1.5  2.5  3.5  4.5  5.5
##
## $xname
## [1] "newX[, i]"
##
## $equidist
## [1] TRUE
##
## attr(,"class")
## [1] "histogram"
##
## $FTSE
## $breaks
##  [1] -5 -4 -3 -2 -1  0  1  2  3  4  5  6
##
## $counts
##  [1]   1   1  19 128 771 765 156  13   3   1   1
##
## $density
##  [1] 0.0005379236 0.0005379236 0.0102205487 0.0688542227 0.4147391070
##  [6] 0.4115115654 0.0839160839 0.0069930070 0.0016137708 0.0005379236
## [11] 0.0005379236
##
## $mids
```

```
##  [1] -4.5 -3.5 -2.5 -1.5 -0.5  0.5  1.5  2.5  3.5  4.5  5.5
##
## $xname
## [1] "newX[, i]"
##
## $equidist
## [1] TRUE
##
## attr(,"class")
## [1] "histogram"
```

```r
# Display normal quantile plots of percent returns for each index
par(mfrow = c(2,2))
# apply(eu_percentreturns, MARGIN = 2, FUN = qqnorm, main = "")
qqline(eu_percentreturns)
```



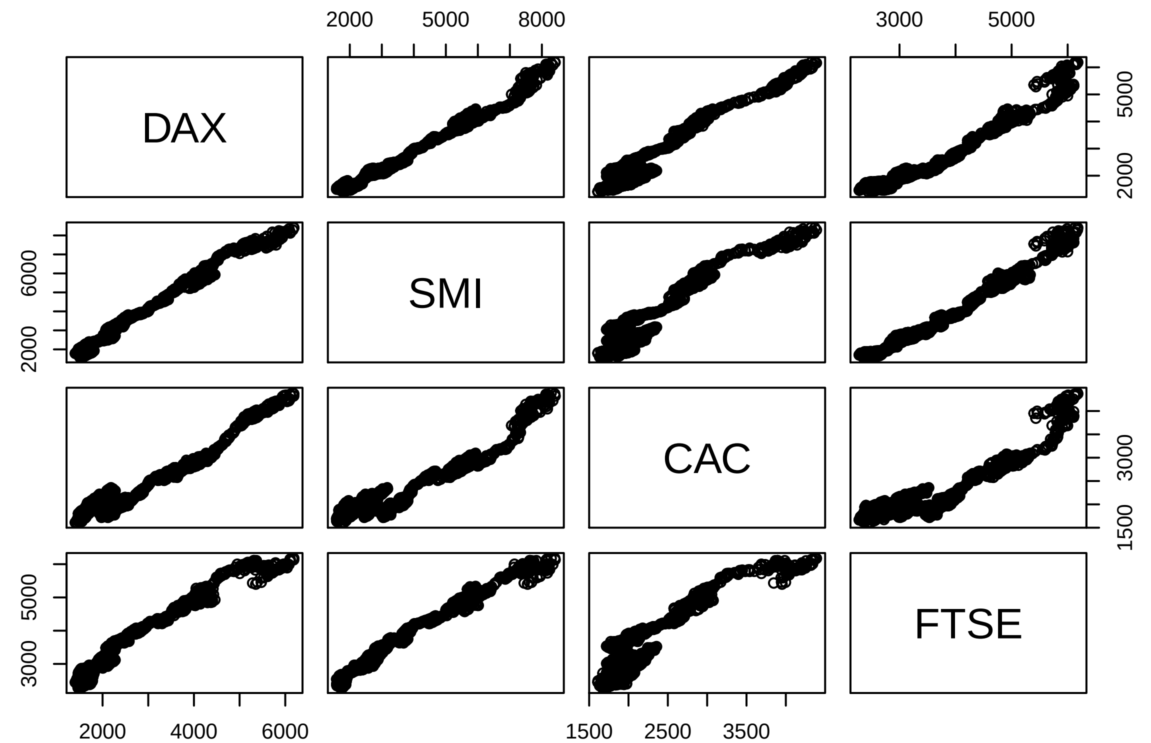
Plotting pairs of data

Time series data is often presented in a time series plot. For example, the index values from the eu_stocks dataset are shown in the adjoining figure. Recall, eu_stocks contains daily closing prices from 1991-1998 for the major stock indices in Germany (DAX), Switzerland (SMI), France (CAC), and the UK (FTSE).

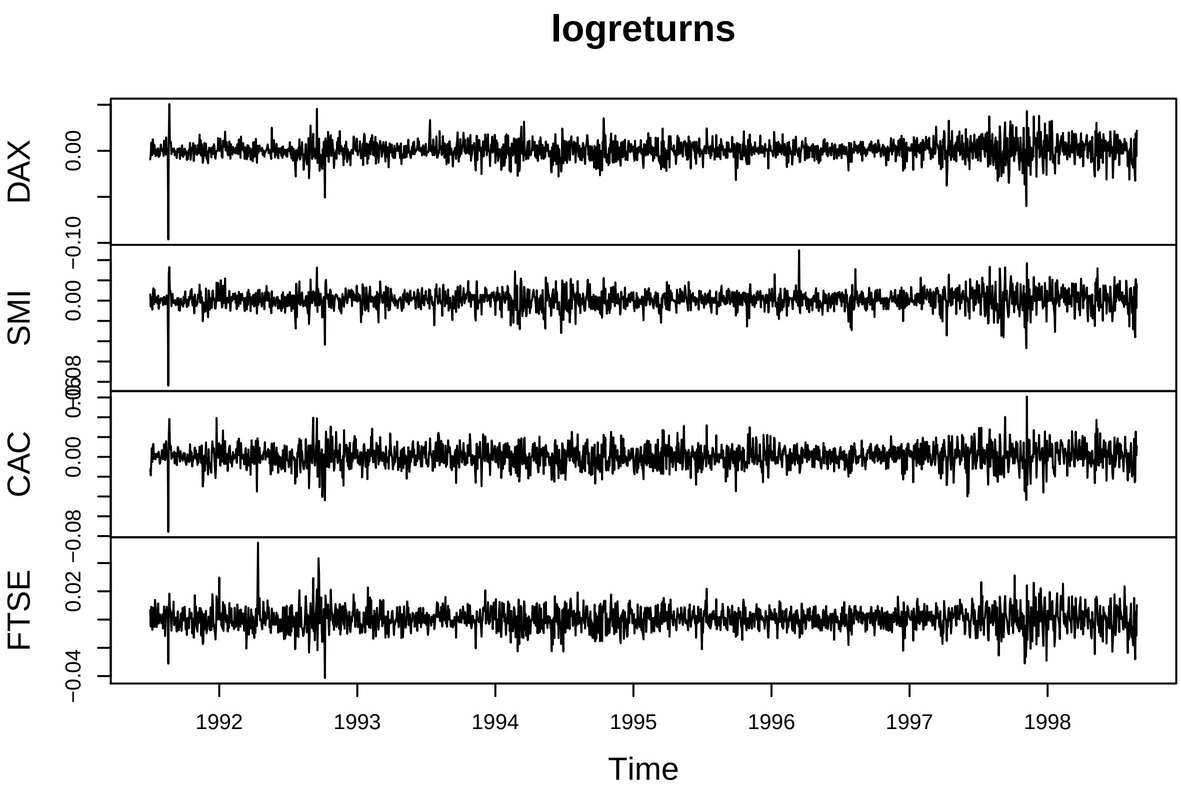
It is also useful to examine the bivariate relationship between pairs of time series. In this exercise we will consider the contemporaneous relationship, that is matching observations that occur at the same time, between pairs of index values as well as their log returns. The plot(a, b) function will produce a scatterplot when two time series names a and b are given as input.

To simultaneously make scatterplots for all pairs of several assets the pairs() function can be applied to produce a scatterplot matrix. When shared time trends are present in prices or index values it is common to instead compare their returns or log returns.

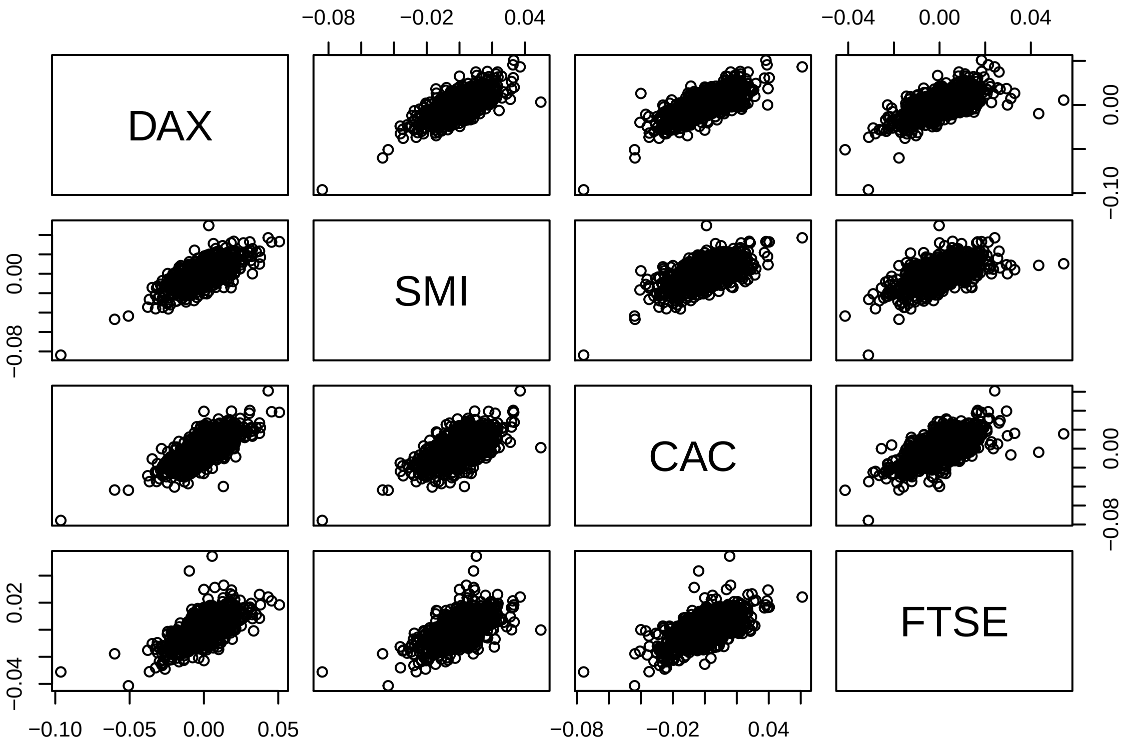
```
# Make a scatterplot matrix of eu_stocks
pairs(eu_stocks)
```




```
# Convert eu_stocks to log returns
logreturns <- diff(log(eu_stocks))

# Plot logreturns
plot(logreturns)
```




```r
# Make a scatterplot matrix of logreturns
pairs(logreturns)
```



Calculating autocorrelations

Autocorrelations or lagged correlations are used to assess whether a time series is dependent on its past. For a time series x of length n we consider the n-1 pairs of observations one time unit apart. The first such pair is (x[2],x[1]), and the next is (x[3],x[2]). Each such pair is of the form (x[t],x[t-1]) where t is the observation index, which we vary from 2 to n in this case. The lag-1 autocorrelation of x can be estimated as the sample correlation of these (x[t], x[t-1]) pairs.

In general, we can manually create these pairs of observations. First, create two vectors, x_t0 and x_t1, each with length n-1, such that the rows correspond to (x[t], x[t-1]) pairs. Then apply the cor() function to estimate the lag-1 autocorrelation.

Luckily, the acf() command provides a shortcut. Applying acf(. . . , lag.max = 1, plot = FALSE) to a series x automatically calculates the lag-1 autocorrelation.

Finally, note that the two estimates differ slightly as they use slightly different scalings in their calculation of sample covariance, 1/(n-1) versus 1/n. Although the latter would provide a biased estimate, it is preferred in time series analysis, and the resulting autocorrelation estimates only differ by a factor of (n-1)/n.

```
library(quantmod)
```

```
## Loading required package: xts
```

```
## Loading required package: zoo
```

```
##
## Attaching package: 'zoo'

## The following objects are masked from 'package:base':
##
##     as.Date, as.Date.numeric

## Loading required package: TTR

## Registered S3 method overwritten by 'quantmod':
##   method            from
##   as.zoo.data.frame zoo

## Version 0.4-0 included new data defaults. See ?getSymbols.
```

```r
getSymbols("TSLA", src = "yahoo")
```

```
## 'getSymbols' currently uses auto.assign=TRUE by default, but will
## use auto.assign=FALSE in 0.5-0. You will still be able to use
## 'loadSymbols' to automatically load data. getOption("getSymbols.env")
## and getOption("getSymbols.auto.assign") will still be checked for
## alternate defaults.
##
## This message is shown once per session and may be disabled by setting
## options("getSymbols.warning4.0"=FALSE). See ?getSymbols for details.

## [1] "TSLA"
```

```r
acf(TSLA$TSLA.Adjusted, lag.max = 10, plot = FALSE)
```

```
##
## Autocorrelations of series 'TSLA$TSLA.Adjusted', by lag
##
##     0     1     2     3     4     5     6     7     8     9    10
## 1.000 0.988 0.976 0.962 0.946 0.929 0.915 0.900 0.886 0.874 0.861
```

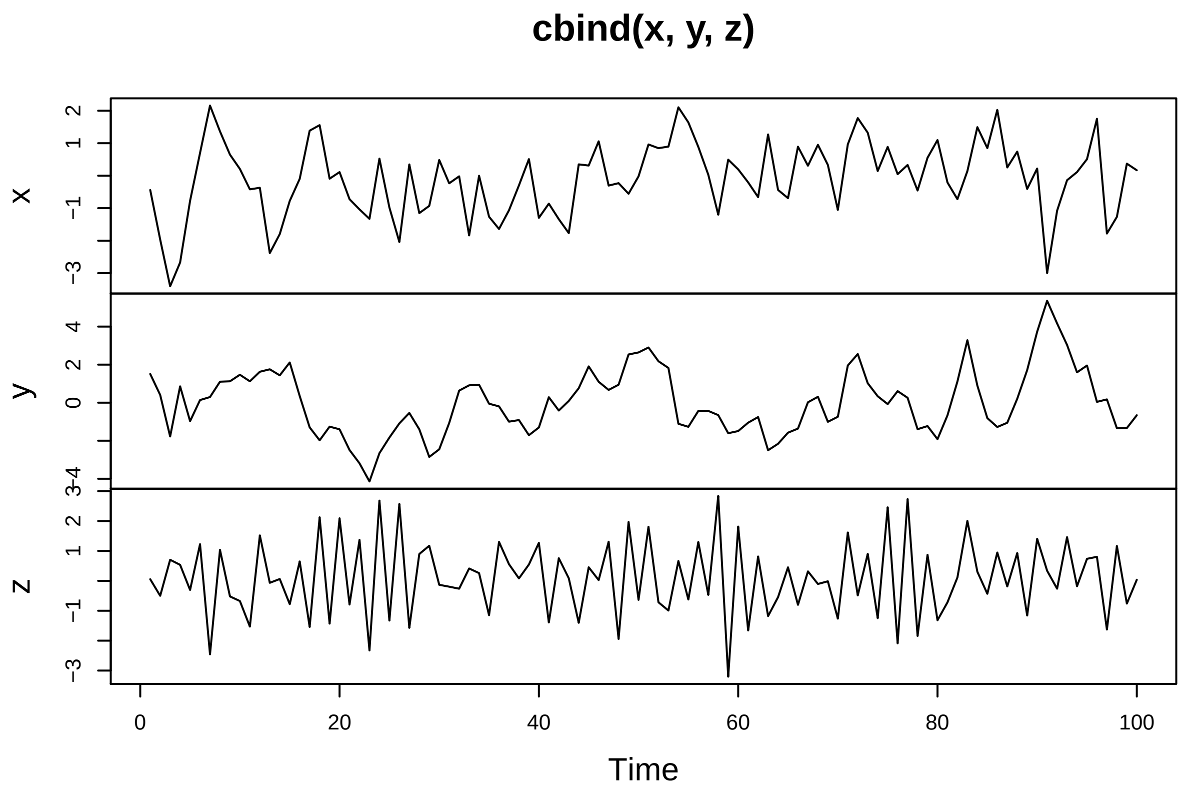
Autoregressive model.

Simulate the autoregressive model

The autoregressive (AR) model is arguably the most widely used time series model. It shares the very familiar interpretation of a simple linear regression, but here each observation is regressed on the previous observation. The AR model also includes the white noise (WN) and random walk (RW) models examined in earlier chapters as special cases.

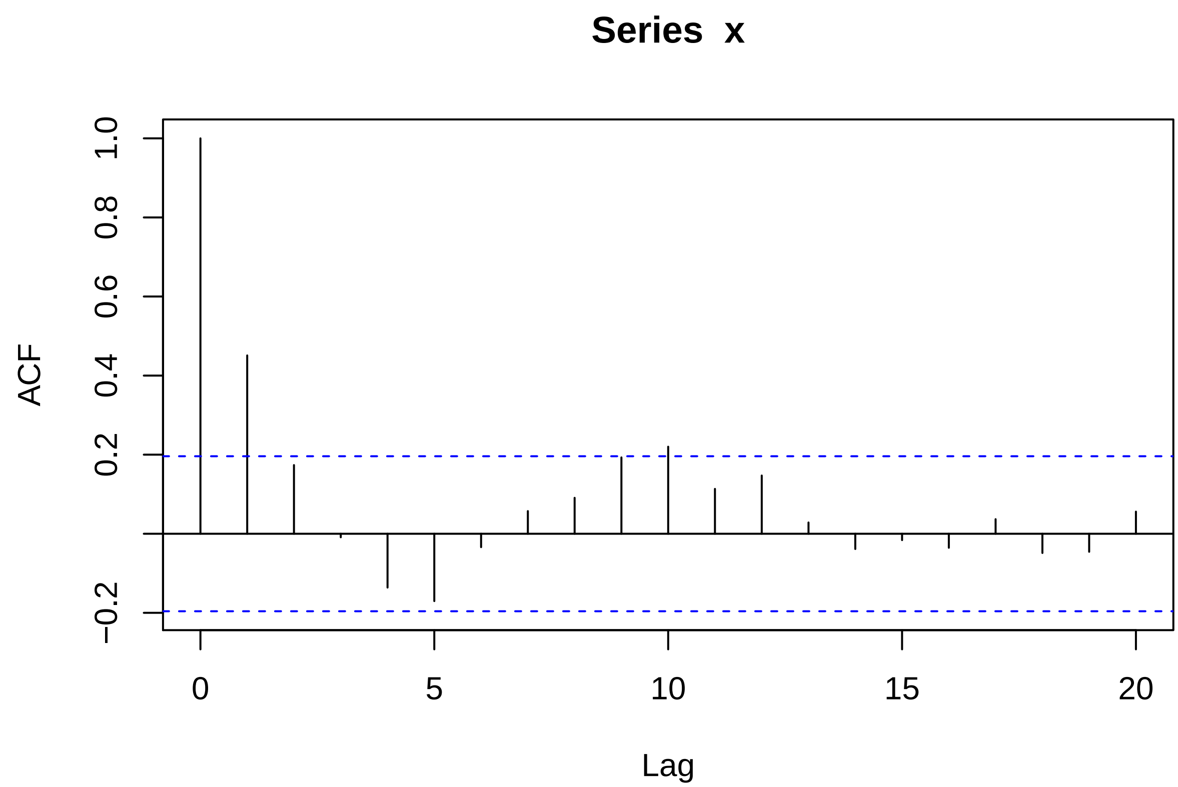
The versatile arima.sim() function used in previous chapters can also be used to simulate data from an AR model by setting the model argument equal to list(ar = phi) , in which phi is a slope parameter from the interval (-1, 1). We also need to specify a series length n.

```r
# Simulate an AR model with 0.5 slope
x <- arima.sim(model = list(ar = 0.5), n = 100)

# Simulate an AR model with 0.9 slope
y <- arima.sim(model = list(ar = 0.9), n = 100)

# Simulate an AR model with -0.75 slope
z <- arima.sim(model = list(ar = -0.75), n = 100)

# Plot your simulated data
plot.ts(cbind(x, y, z))
```

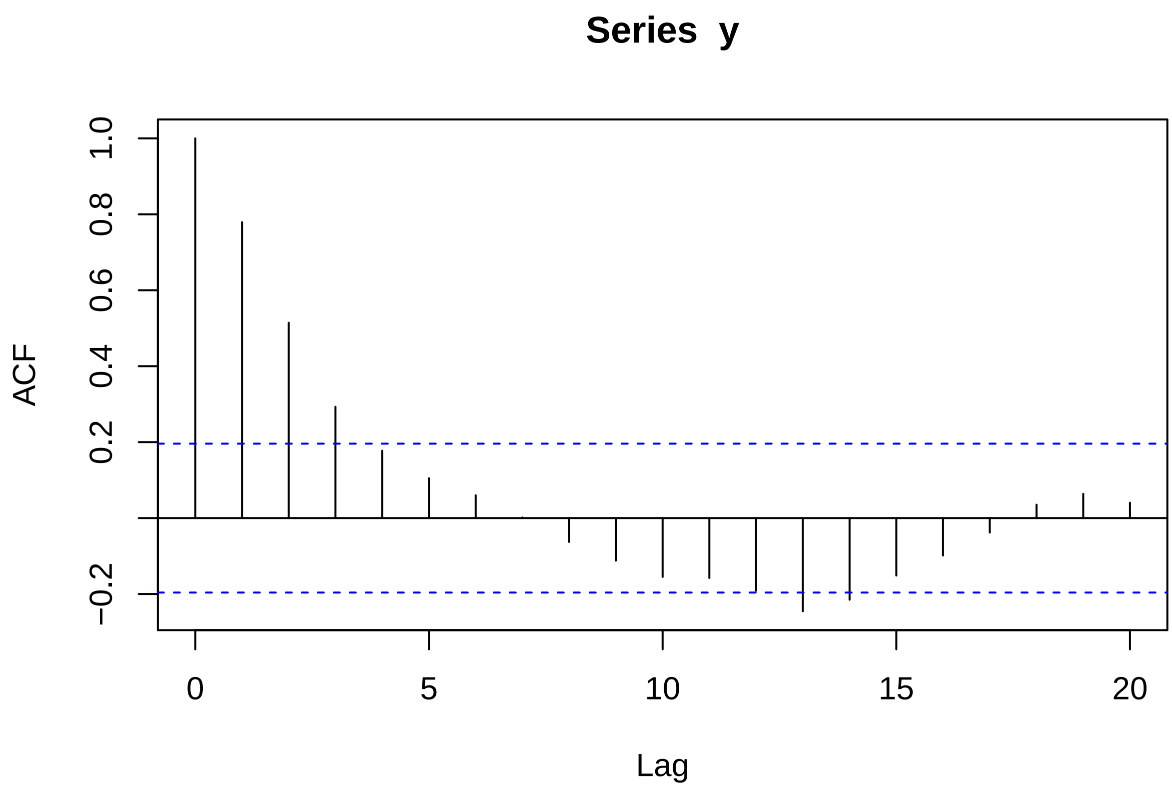


Estimate the autocorelation function (ACF) from an autoregression

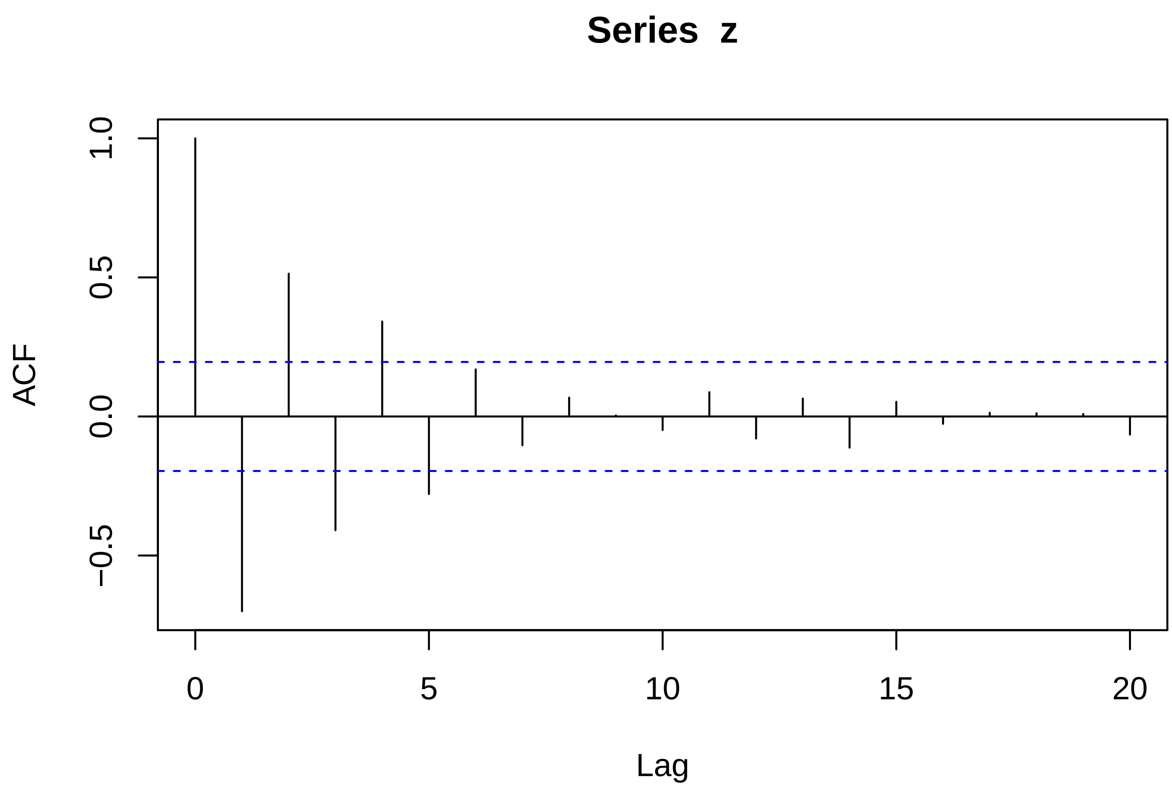
```r
# Calculate the ACF for x
acf(x)
```




```
# Calculate the ACF for y
acf(y)
```




```r
# Calculate the ACF for z
acf(z)
```

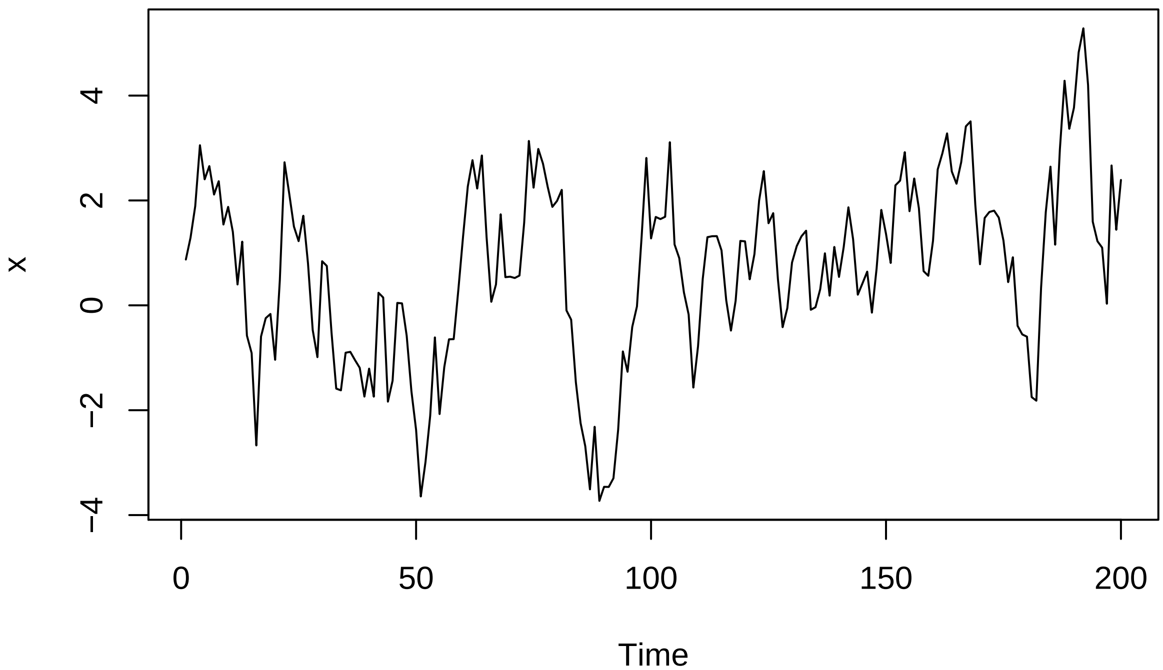


Compare the random walk (RW) and autoregressive (AR) models

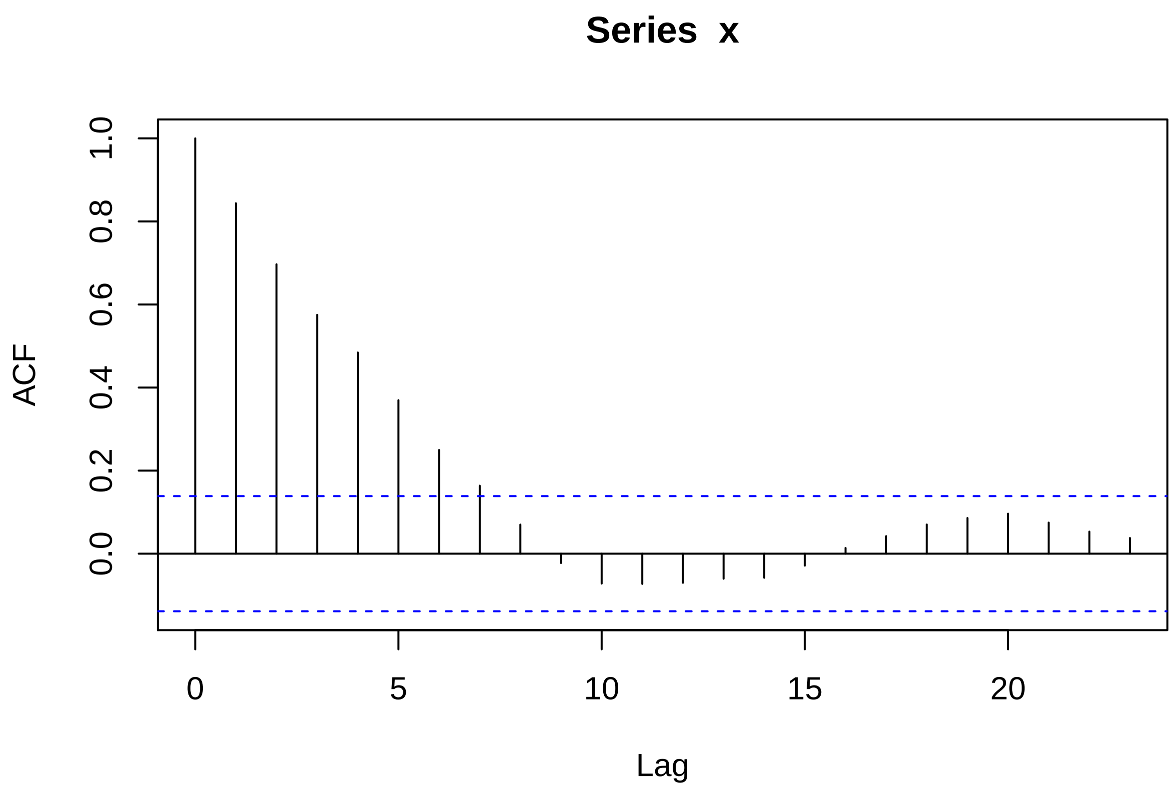
The random walk (RW) model is a special case of the autoregressive (AR) model, in which the slope parameter is equal to 1. Recall from previous chapters that the RW model is not stationary and exhibits very strong persistence. Its sample autocovariance function (ACF) also decays to zero very slowly, meaning past values have a long lasting impact on current values.

The stationary AR model has a slope parameter between -1 and 1. The AR model exhibits higher persistence when its slope parameter is closer to 1, but the process reverts to its mean fairly quickly. Its sample ACF also decays to zero at a quick (geometric) rate, indicating that values far in the past have little impact on future values of the process.

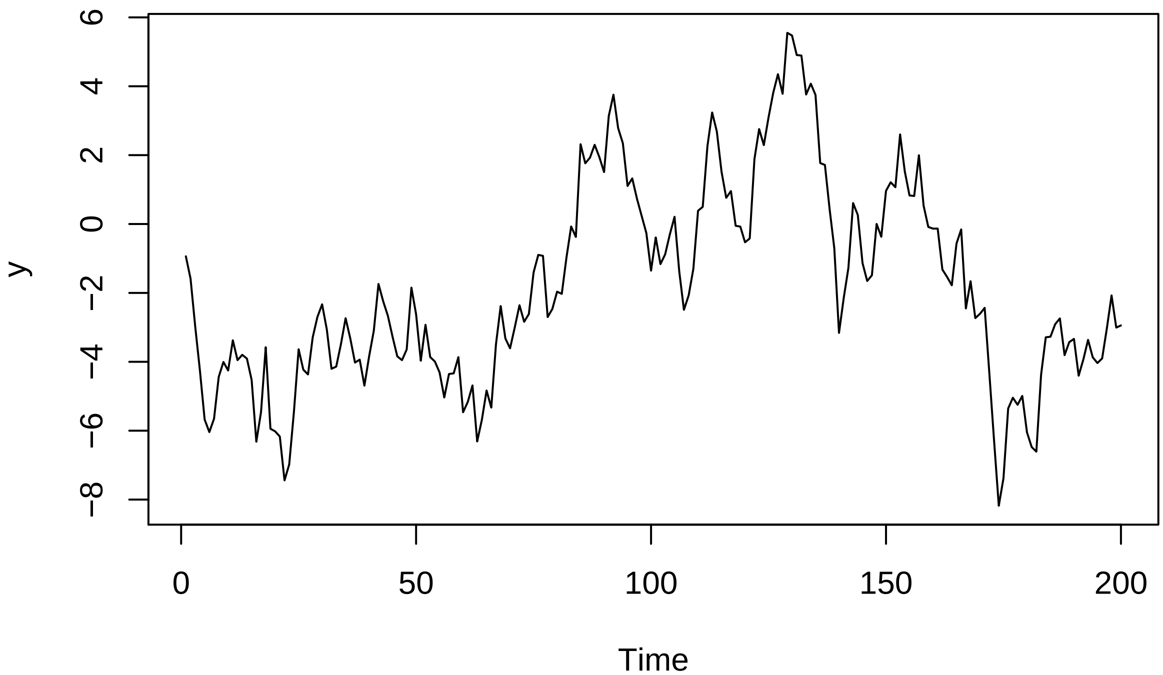
```
# Simulate and plot AR model with slope 0.9
x <- arima.sim(model = list(ar = 0.9), n = 200)
ts.plot(x)
```



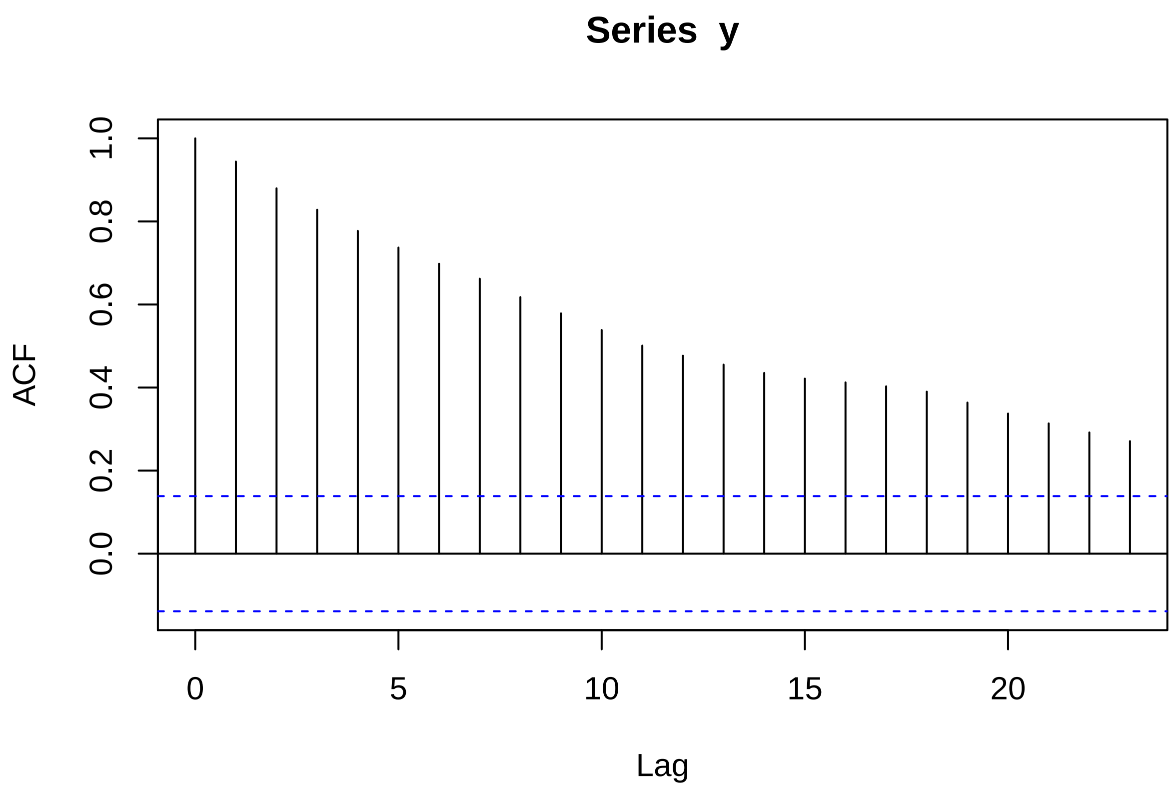

```
acf(x)
```



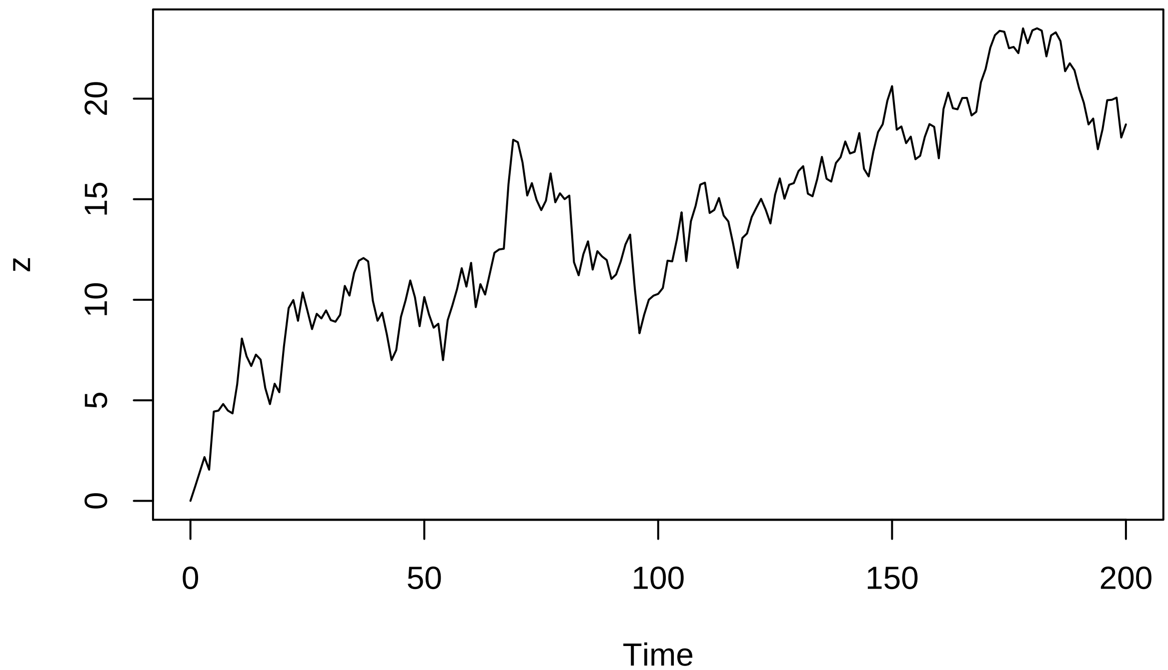

```r
# Simulate and plot AR model with slope 0.98
y <- arima.sim(model = list(ar = 0.98), n = 200)
ts.plot(y)
```



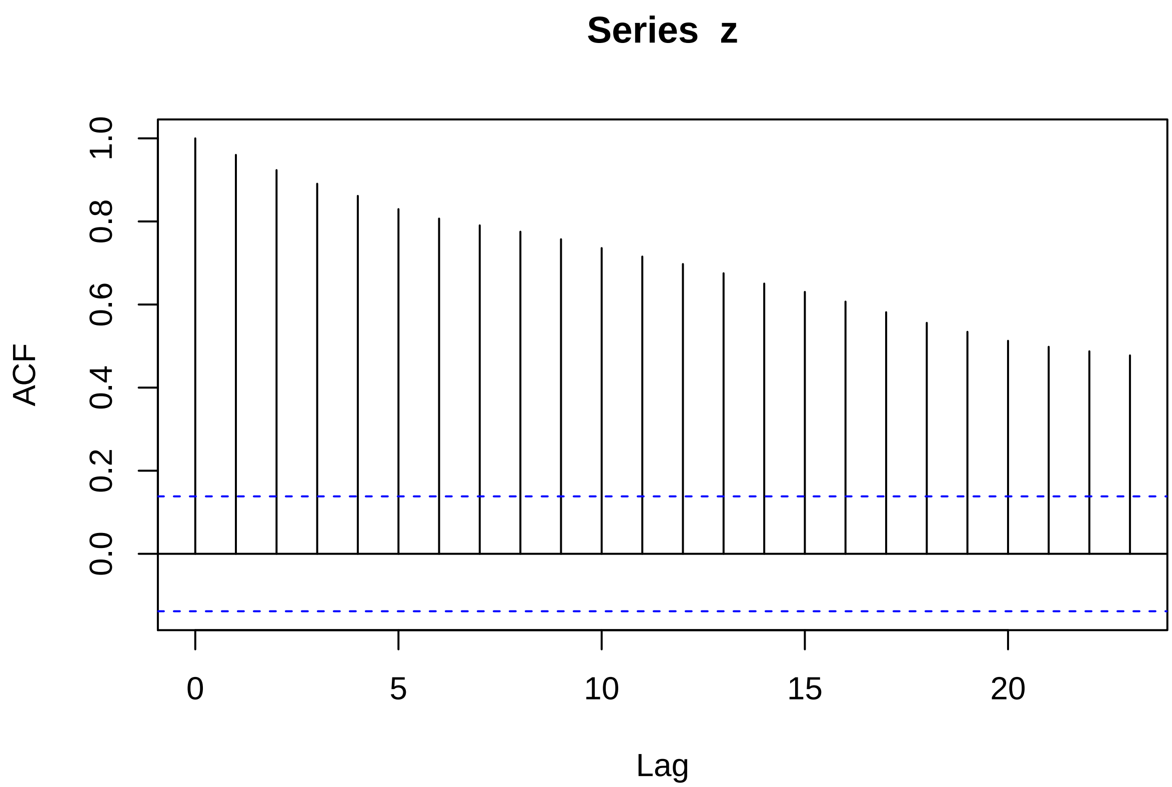

```
acf(y)
```




```r
# Simulate and plot RW model
z <- arima.sim(model = list(order=c(0,1,0)), n = 200)
ts.plot(z)
```




```
acf(z)
```



Estimate the autoregressive (AR) model

For a given time series x we can fit the autoregressive (AR) model using the arima() command and setting order equal to c(1, 0, 0). Note for reference that an AR model is an ARIMA(1, 0, 0) model.

```
# Fit the AR model to x
arima(x, order = c(1, 0, 0))
```

```
##
## Call:
## arima(x = x, order = c(1, 0, 0))
##
## Coefficients:
##          ar1  intercept
##       0.8438     0.7499
## s.e.  0.0372     0.4028
##
## sigma^2 estimated as 0.8335:  log likelihood = -266.19,  aic = 538.38
```

```
# Copy and paste the slope (ar1) estimate
0.8575
```

```
## [1] 0.8575
```

```r
# Copy and paste the slope mean (intercept) estimate
-0.0948
```

```
## [1] -0.0948
```

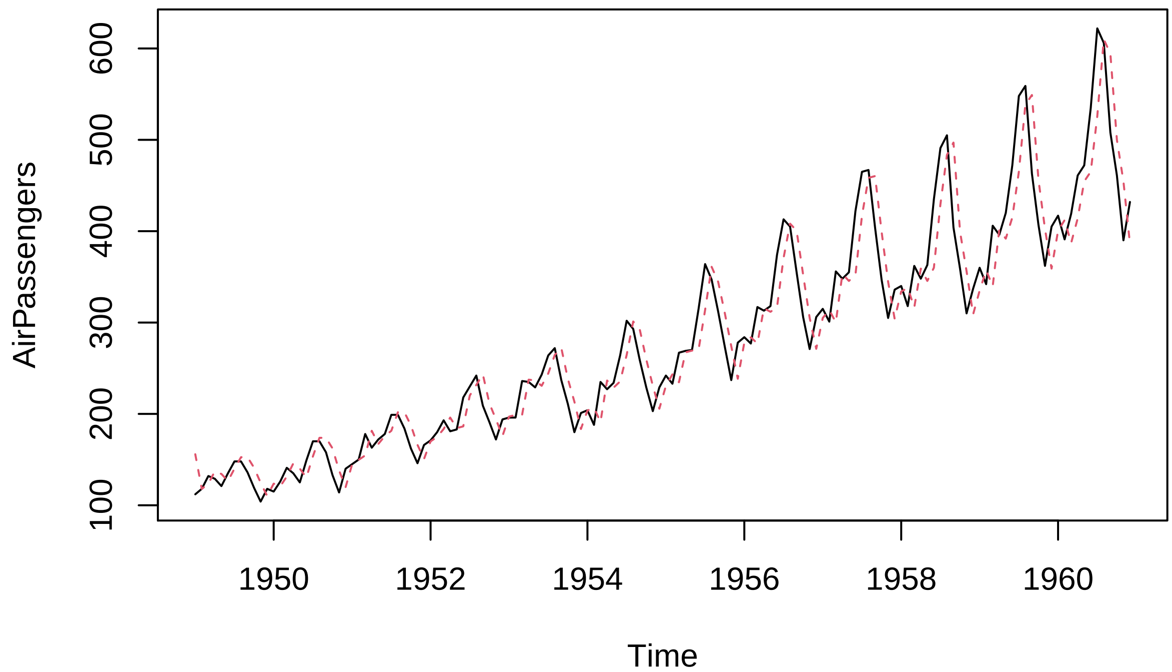
```r
# Copy and paste the innovation variance (sigma^2) estimate
1.022
```

```
## [1] 1.022
```

```r
# Fit the AR model to AirPassengers
AR <- arima(AirPassengers, order = c(1, 0, 0))
print(AR)
```

```
##
## Call:
## arima(x = AirPassengers, order = c(1, 0, 0))
##
## Coefficients:
##          ar1  intercept
##       0.9646   278.4649
## s.e.  0.0214    67.1141
##
## sigma^2 estimated as 1119:  log likelihood = -711.09,  aic = 1428.18
```

```r
# Run the following commands to plot the series and fitted values
ts.plot(AirPassengers)
AR_fitted <- AirPassengers - residuals(AR)
points(AR_fitted, type = "l", col = 2, lty = 2)
```



Simple forecasts from an estimated AR model

Now that you've modeled your data using the arima() command, you are ready to make simple forecasts based on your model. The predict() function can be used to make forecasts from an estimated AR model. In the object generated by your predict() command, the $pred value is the forecast, and the $se value is the standard error for the forecast.

To make predictions for several periods beyond the last observations, you can use the n.ahead argument in your predict() command. This argument establishes the forecast horizon (h), or the number of periods being forecast. The forecasts are made recursively from 1 to h-steps ahead from the end of the observed time series.

```r
# Fit an AR model to Nile
AR_fit <- arima(Nile, order = c(1,0,0))
print(AR_fit)
```

```
##
## Call:
## arima(x = Nile, order = c(1, 0, 0))
##
## Coefficients:
##          ar1  intercept
##       0.5063   919.5685
## s.e.  0.0867    29.1410
##
```

```
## sigma^2 estimated as 21125:  log likelihood = -639.95,  aic = 1285.9
```

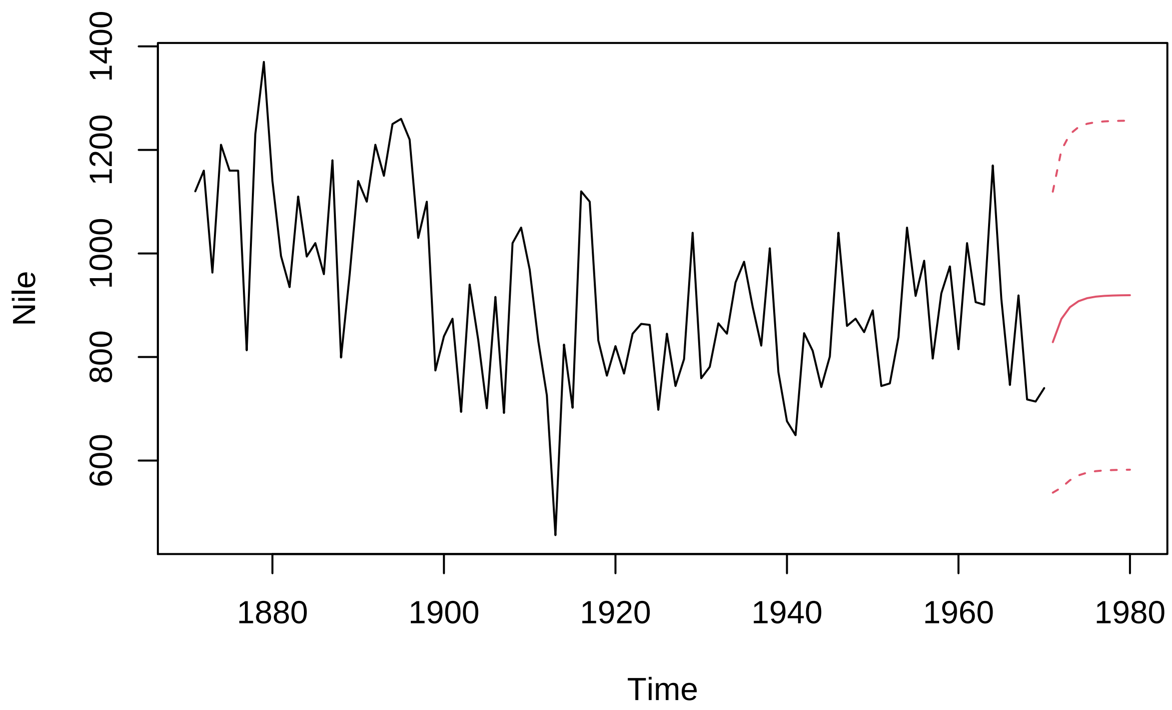
```
# Use predict() to make a 1-step forecast
predict_AR <- predict(AR_fit)

# Obtain the 1-step forecast using $pred[1]
predict_AR$pred[1]
```

```
## [1] 828.6576
```

```
# Use predict to make 1-step through 10-step forecasts
predict(AR_fit, n.ahead = 10)
```

```
## $pred
## Time Series:
## Start = 1971
## End = 1980
## Frequency = 1
##  [1] 828.6576 873.5426 896.2668 907.7715 913.5960 916.5448 918.0377 918.7935
##  [9] 919.1762 919.3699
##
## $se
## Time Series:
## Start = 1971
## End = 1980
## Frequency = 1
##  [1] 145.3439 162.9092 167.1145 168.1754 168.4463 168.5156 168.5334 168.5380
##  [9] 168.5391 168.5394
```

```
# Run to plot the Nile series plus the forecast and 95% prediction intervals
ts.plot(Nile, xlim = c(1871, 1980))
AR_forecast <- predict(AR_fit, n.ahead = 10)$pred
AR_forecast_se <- predict(AR_fit, n.ahead = 10)$se
points(AR_forecast, type = "l", col = 2)
points(AR_forecast - 2*AR_forecast_se, type = "l", col = 2, lty = 2)
points(AR_forecast + 2*AR_forecast_se, type = "l", col = 2, lty = 2)
```



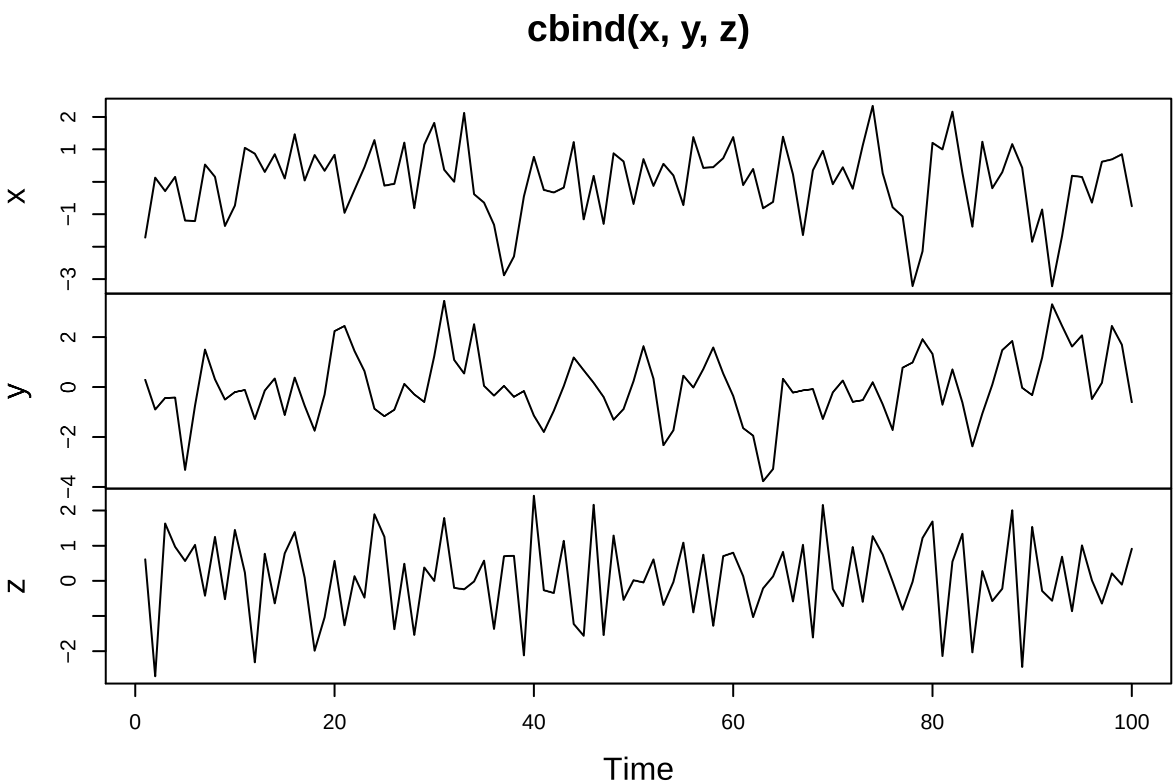
A simple moving average

Simulate the simple moving average model

The simple moving average (MA) model is a parsimonious time series model used to account for very short-run autocorrelation. It does have a regression like form, but here each observation is regressed on the previous innovation, which is not actually observed. Like the autoregressive (AR) model, the MA model includes the white noise (WN) model as special case.

As with previous models, the MA model can be simulated using the arima.sim() command by setting the model argument to list(ma = theta), where theta is a slope parameter from the interval (-1, 1). Once again, you also need to specifcy the series length using the n argument.

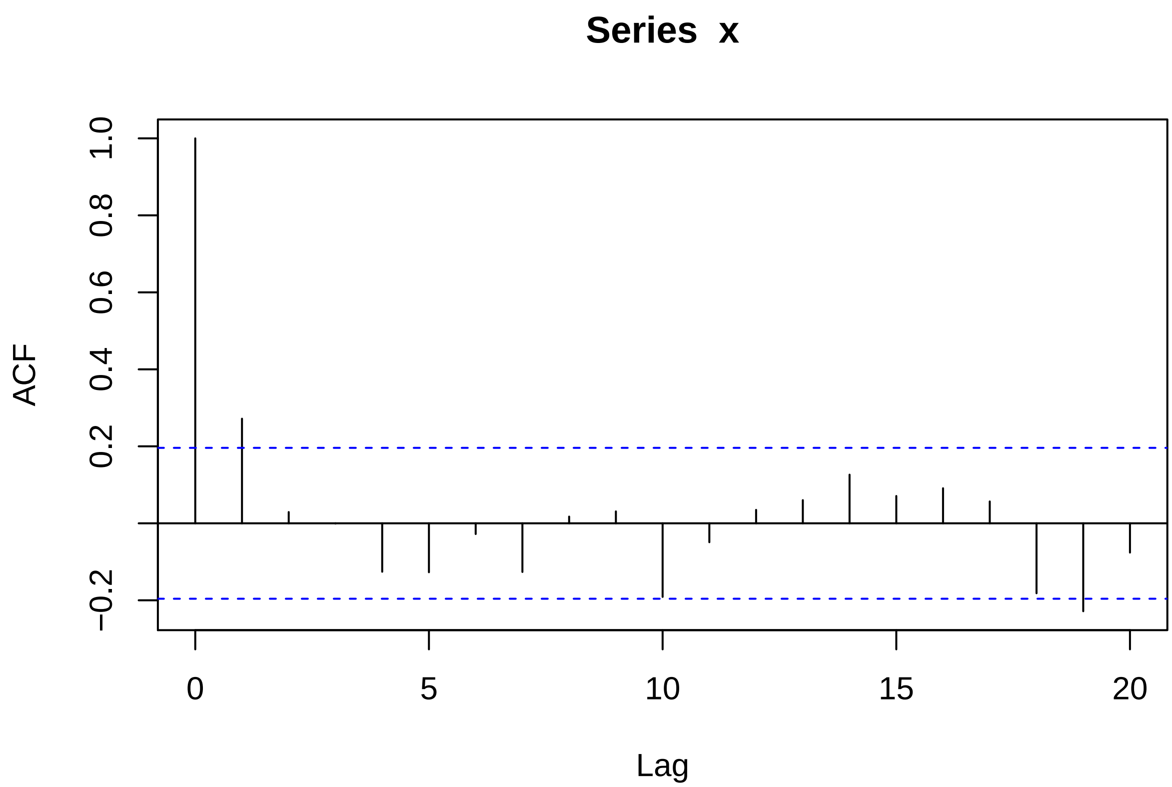
```
# Generate MA model with slope 0.5
x <- arima.sim(model = list(ma = 0.5), n = 100)

# Generate MA model with slope 0.9
y <- arima.sim(model = list(ma = 0.9), n = 100)

# Generate MA model with slope -0.5
z <- arima.sim(model = list(ma = -0.5), n = 100)

# Plot all three models together
plot.ts(cbind(x, y, z))
```



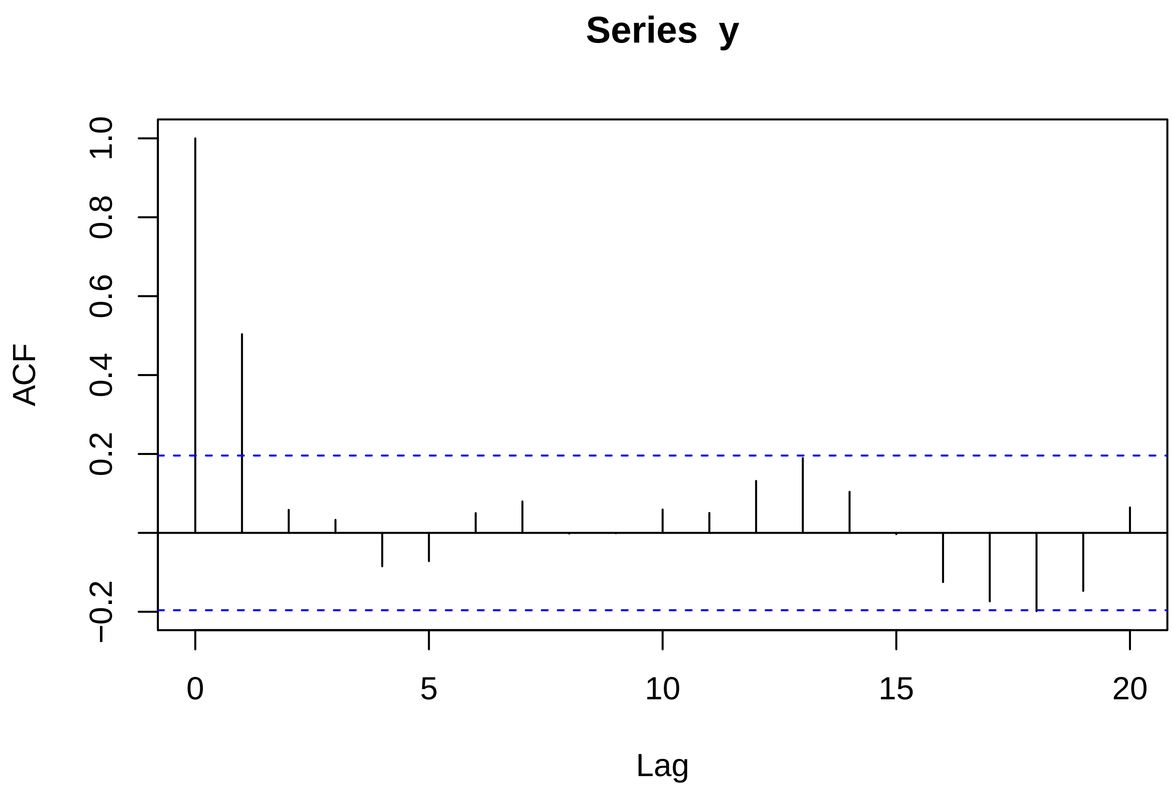
Estimate the autocorrelation function (ACF) for a moving average

Now that you've simulated some MA data using the arima.sim() command, you may want to estimate the autocorrelation functions (ACF) for your data. As in the previous chapter, you can use the acf() command to generate plots of the autocorrelation in your MA data.

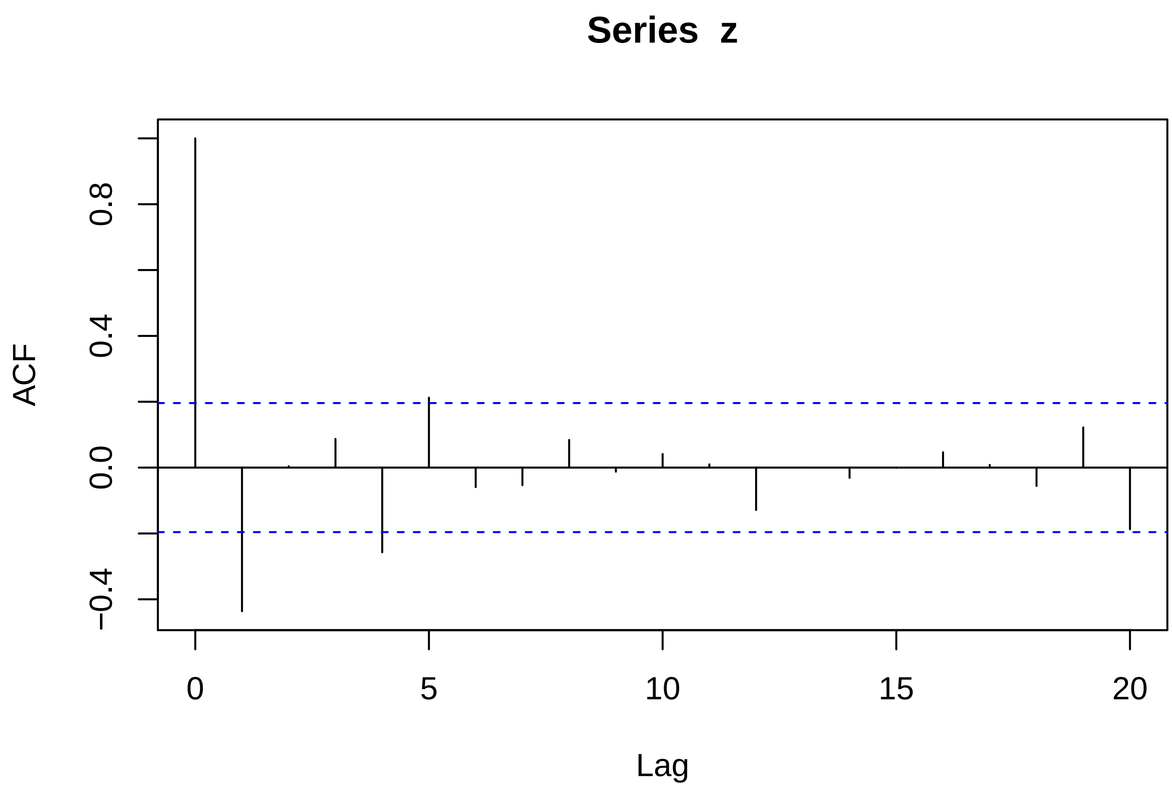
```
# Calculate ACF for x
acf(x)
```




```r
# Calculate ACF for y
acf(y)
```




```
# Calculate ACF for z
acf(z)
```



Estimate the simple moving average model

Now that you've simulated some MA models and calculated the ACF from these models, your next step is to fit the simple moving average (MA) model to some data using the arima() command. For a given time series x we can fit the simple moving average (MA) model using arima(..., order = c(0, 0, 1)). Note for reference that an MA model is an ARIMA(0, 0, 1) model.

```r
# Fit the MA model to `x`
arima(x, order = c(0, 0, 1))
```

```
##
## Call:
## arima(x = x, order = c(0, 0, 1))
##
## Coefficients:
##          ma1  intercept
##       0.2941    -0.0017
## s.e.  0.0980     0.1364
##
## sigma^2 estimated as 1.116:  log likelihood = -147.42,  aic = 300.83
```

```r
# Paste the slope (ma1) estimate below
0.7928
```

```
## [1] 0.7928
```

```r
# Paste the slope mean (intercept) estimate below
0.1589
```

```
## [1] 0.1589
```

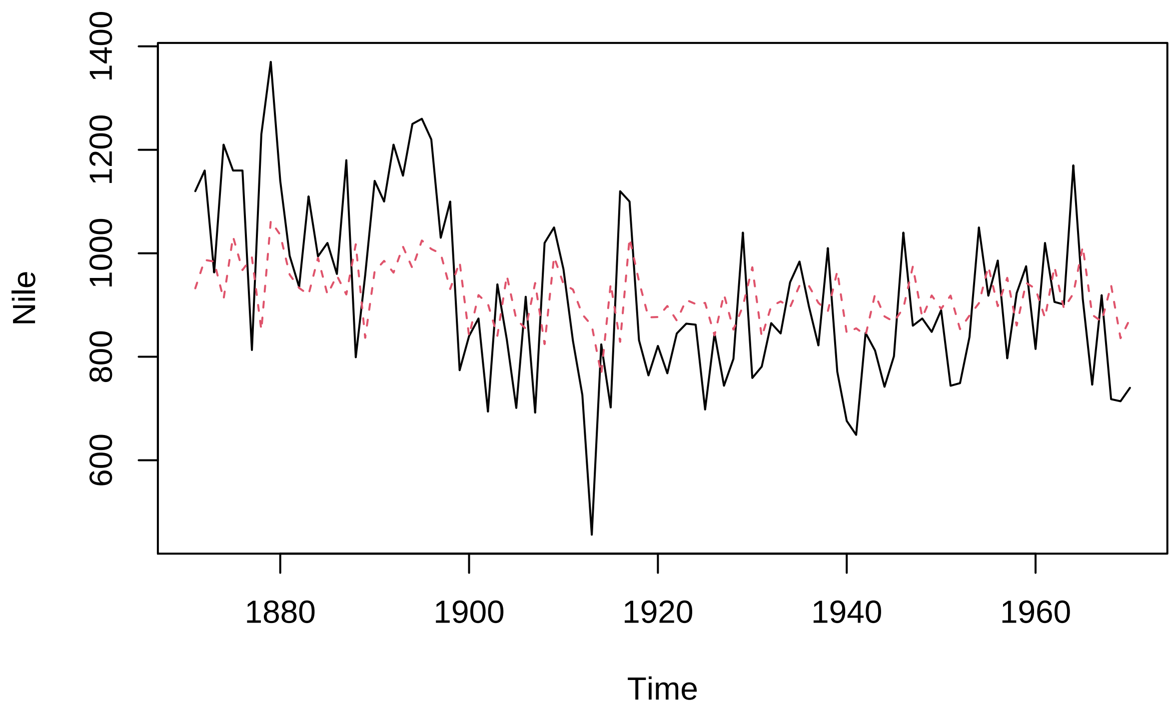
```r
# Paste the innovation variance (sigma^2) estimate below
0.9576
```

```
## [1] 0.9576
```

```r
# Fit the MA model to Nile
MA <- arima(Nile, order = c(0, 0, 1))
print(MA)
```

```
##
## Call:
## arima(x = Nile, order = c(0, 0, 1))
##
## Coefficients:
##          ma1  intercept
##       0.3783   919.2433
## s.e.  0.0791    20.9685
##
## sigma^2 estimated as 23272:  log likelihood = -644.72,  aic = 1295.44
```

```r
# Plot Nile and MA_fit
ts.plot(Nile)
MA_fit <- Nile - resid(MA)
points(MA_fit, type = "l", col = 2, lty = 2)
```



Simple forecasts from an estimated MA model

Now that you've estimated a MA model with your Nile data, the next step is to do some simple forecasting with your model. As with other types of models, you can use the predict() function to make simple forecasts from your estimated MA model. Recall that the $pred value is the forecast, while the $se value is a standard error for that forecast, each of which is based on the fitted MA model.

Once again, to make predictions for several periods beyond the last observation you can use the n.ahead = h argument in your call to predict(). The forecasts are made recursively from 1 to h-steps ahead from the end of the observed time series. However, note that except for the 1-step forecast, all forecasts from the MA model are equal to the estimated mean (intercept).

```
# Make a 1-step forecast based on MA
predict_MA <- predict(MA)

# Obtain the 1-step forecast using $pred[1]
predict_MA$pred[1]
```

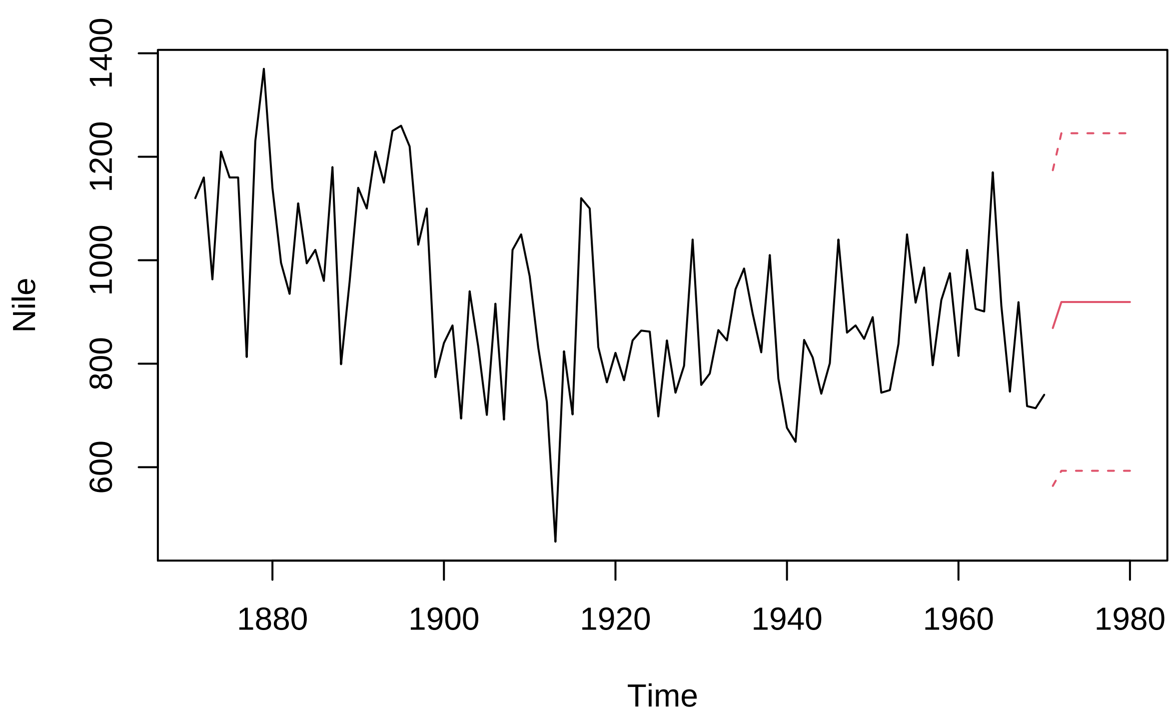
```
## [1] 868.8747
```

```
# Make a 1-step through 10-step forecast based on MA
predict(MA, n.ahead = 10)
```

```
## $pred
```

```
## Time Series:
## Start = 1971
## End = 1980
## Frequency = 1
##  [1] 868.8747 919.2433 919.2433 919.2433 919.2433 919.2433 919.2433 919.2433
##  [9] 919.2433 919.2433
##
## $se
## Time Series:
## Start = 1971
## End = 1980
## Frequency = 1
##  [1] 152.5508 163.1006 163.1006 163.1006 163.1006 163.1006 163.1006 163.1006
##  [9] 163.1006 163.1006
```

```
# Plot the Nile series plus the forecast and 95% prediction intervals
ts.plot(Nile, xlim = c(1871, 1980))
MA_forecasts <- predict(MA, n.ahead = 10)$pred
MA_forecast_se <- predict(MA, n.ahead = 10)$se
points(MA_forecasts, type = "l", col = 2)
points(MA_forecasts - 2*MA_forecast_se, type = "l", col = 2, lty = 2)
points(MA_forecasts + 2*MA_forecast_se, type = "l", col = 2, lty = 2)
```



AR vs MA models

As you've seen, autoregressive (AR) and simple moving average (MA) are two useful approaches to modeling time series. But how can you determine whether an AR or MA model is more appropriate in practice?

To determine model fit, you can measure the Akaike information criterion (AIC) and Bayesian information criterion (BIC) for each model. While the math underlying the AIC and BIC is beyond the scope of this course, for your purposes the main idea is these these indicators penalize models with more estimated parameters, to avoid overfitting, and smaller values are preferred. All factors being equal, a model that produces a lower AIC or BIC than another model is considered a better fit.

To estimate these indicators, you can use the AIC() and BIC() commands, both of which require a single argument to specify the model in question.

In this exercise, you'll return to the Nile data and the AR and MA models you fitted to this data. These models and their predictions for the 1970s (AR_fit) and (MA_fit) are depicted in the plot on the right.

```r
# Find correlation between AR_fit and MA_fit
# cor(AR_fit, MA_fit)

# Find AIC of AR
AIC(AR)
```

```
## [1] 1428.179
```

```r
# Find AIC of MA
AIC(MA)
```

```
## [1] 1295.442
```

```r
# Find BIC of AR
BIC(AR)
```

```
## [1] 1437.089
```

```r
# Find BIC of MA
BIC(MA)
```

```
## [1] 1303.257
```